Preprint of Grundy, J.C. and Hosking, J.G. Engineering plug-in software components to support collaborative work, Software – Practice and Experience, vol. 32, Wiley, pp. 983-1013, 2002.

Engineering plug-in software components to support collaborative work

JOHN GRUNDY

Department of Electrical and Electronic Engineering and Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
john-g@cs.auckland.ac.nz

JOHN HOSKING
Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand
john@ cs.auckland.ac.nz

ABSTRACT

Many software applications require co-operative work support, including collaborative editing, group awareness, versioning, messaging and automated notification and co-ordination agents. Most approaches hard-code such facilities into applications, with fixed functionality and limited ability to reuse groupware implementations. We describe our recent work in seamlessly adding such capabilities to component-based applications via a set of collaborative work-supporting plug-in software components. We describe a variety of applications of this technique, along with descriptions of the novel architecture, user interface adaptation and implementation techniques for the collaborative work-supporting components that we have developed. We report on our experiences to date with this method of supporting collaborative work enhancement of component-based systems, and discuss the advantages of our approach over conventional techniques.

KEYWORDS: software components, groupware, collaborative work tools, software architecture, user interface adaptation, aspect-oriented design

INTRODUCTION

Many software applications, such as CASE tools, programming environments, process and project management tools, and distributed Information Systems, require collaborative work facilities. Such facilities include support for both synchronous and asynchronous editing of documents; group awareness facilities; annotation and versioning of documents; messaging, email and chat; change notification and other task automation facilities, and process-based work co-ordination. Because of this need for collaborative work support, many frameworks, toolkits and application generators have been developed to help construct tools with collaborative work, or "groupware", facilities. Examples include the groupware toolkits GroupKit[1], Clockworks[2, 3], Rendezvous[4, 5], Suite[6], and COAST[7]; architectures like Clock[3], MetaMOOSE[8] and ALV[5]; and extensible groupware tools like Teamwave[9], CocoDoc[10], ConversationBuilder[11], and MS Netmeeting™[12]. Many groupware-enabled tools have also been developed using ad-hoc techniques, such as Grove[13], Mjolner[14], Oz[16] and SPADE[15].

While these approaches allow developers to build systems that support various kinds of collaborative work, they each have key disadvantages. Groupware toolkits typically support only limited forms of groupware facilities, such as real-time conferencing (e.g. GroupKit), or groupware must be built into custom user interfaces and architectures, resulting in portability, extensibility and performance problems (e.g. Rendezvous, Suite). Most "extensible" groupware systems restrict new tools to those built with the system's architecture (e.g. ConversationBuilder, Teamwave, COAST). A major problem with most systems is that groupware facilities are built-in and not dynamically deployable and extensible (e.g. Netmeeting and CocoDoc), resulting in unnecessary architectural overheads when some groupware facilities are not needed by end users, and applications with "fixed", non-expandable groupware functionality.

To solve these problems we have been developing plug-in software components to support the addition of groupware facilities to applications, even while they are in use. We have developed many such components and reused them in a number of component-based applications. Our approach is elegant, in that it does not require any modification to the code of the plug-in groupware components, nor the components of the application they are plugged into. Our approach differs from other component-based groupware systems, like COAST, CocoaDoc, Teamwave and Clock, in the variety of collaborative support possible, the openness of the architecture, the ability to add and remove groupware components dynamically, and the range of components and enhanced applications that we have developed.

In the following section we give some examples of component-based applications that require various plug-in groupware capabilities, and overview the requirements of such capabilities from application end user and application developer perspectives. We review related work strengths and weaknesses with respect to these groupware requirements. We then give examples component-based groupware user interfaces using some simple group work scenarios for different application domains, to demonstrate the versatility of application of our work. We then give an overview of the key kinds of groupware components we have identified and their basic architecture. We describe our own component-based framework we have used to build a number of groupware components and illustrate some groupware component architectures realised with this framework. We briefly discuss some of our groupware component designs and the implementation approaches we have used. We conclude by discussing the range of groupware components we have been able to build with this approach, results of reliability, performance and usability evaluations of some of these components, and areas for future work. We hope our experiences will be useful for others interested in developing their own component-based groupware applications.

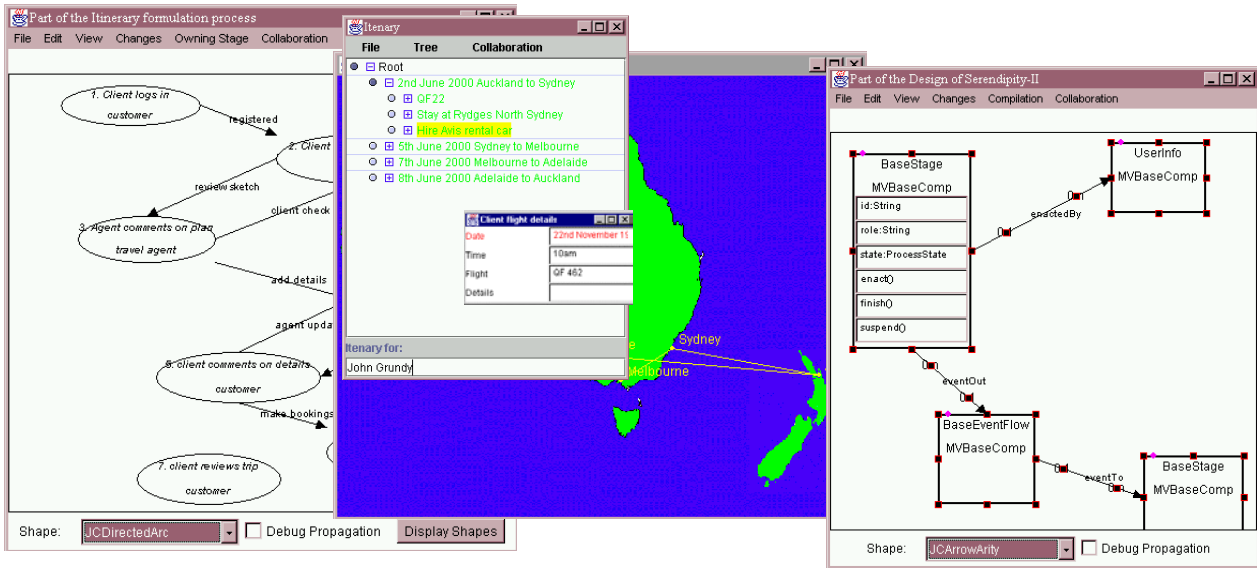
MOTIVATION

Figure 1 shows some component-based software engineering tools and distributed information systems we have developed[17, 18]. View (a) shows a workflow tool process diagram, used to describe workflow for software developers (but can be used to describe office automation workflow etc). View (b) shows an E-commerce application, a collaborative travel itinerary planner, we have developed[18]. This allows travel agents and customers to collaboratively plan a travel itinerary using a variety of tools, including an itinerary tree editor, itinerary item dialogues, itinerary visualisations and web browser. View (c) shows a CASE tool diagram under construction, used by software developers to aid the design of complex software systems. All of these tools were built by composing collections of "software components" i.e. reusable building blocks. Many components are shared between these quite different application domains (e.g. editing tools, data management and distributed system support, icons, event histories, and so on)[19].

When using such applications, users require a wide range of "groupware" (collaborative work) facilities, to help them work effectively together[6, 13, 11, 17]. While the travel itinerary planner is a very

different application to the software development tools, users require very similar kinds of collaborative work support. Such groupware functionality might include[20]:

- *Collaborative work support.* This includes collaborative view editing, both synchronous and asynchronous (supporting multiple users editing CASE diagrams, workflow diagrams and travel itineraries together on-line or off-line). Versioning of edited information is necessary to support off-line work. "Group awareness" needs to be supported i.e. so other users can tell what a user is doing[21, 13, 14, 1].
- *Communication support.* This allows users to communicate about their work, for example text messaging, email, text and audio chat, video conferencing and note annotations on work artefacts[22, 17].
- *Co-ordination support*. Users need appropriate locking of view components currently being edited, highlighting of currently edited or recently edited components to avoid conflicts in their updates. Histories of work done need to be supported. Automatic notification of updates to views via communication mechanisms should be provided so users are told when others do "interesting things". A shared work schedule ("to-do list") is often desirable, along with work co-ordination via workflow tools[21, 1].



(a) Workflow tool (b) Travel planning tool (c) CASE tool

Figure 1. Examples of applications requiring groupware facilities.

When developing the systems illustrated in Figure 1, we have reused many software components, as reported elsewhere[19]. However, much of the collaborative work support these environments provide we originally built into the various framework abstractions and components we reused[17, 19]. Thus while we used reusable components to built applications, our groupware support followed the "monolithic" application development approach, where developers had little control on what groupware support was included in specialised components, and this resulted in sometimes inappropriately reused and over-complex component functionality[19, 18]. In addition, users were not given any control over what groupware functionality their environments supported, even if this was inappropriate to their specific needs.

The usefulness of this component-based approach to building tools such as those illustrated in Figure 1 leads to the question of whether it might be possible to develop components that embody the various collaboration facilities each of these applications (and others) require. Different kinds of collaborative work support could then be incorporated into discrete software components and be

able to be selectively reused, and even sometimes dynamically added to or removed from running applications as users require.

Such "groupware components" need to satisfy a number of key requirements to be effectively used in diverse systems. Groupware components need to:

- *Be deployable statically or dynamically*. Developers should be able to incorporate groupware components in applications at design and implementation time, and end users plug them into their running systems when in use.
- *Be seamlessly integrated with other software components*. Groupware components must utilise the existing event and method invocation interfaces of application components to provide the range of collaborative work facilities outlined above. Their user interfaces must also seamlessly integrate with the user interfaces of existing application components, even when dynamically deployed. This implies a need for components to provide appropriate methods to carry out such adaptations in a consistent, de-coupled manner.
- *Have discrete functionality*. Each groupware component should offer a distinct, preferably non-overlapping set of tailorable groupware facilities, allowing developers and end users to choose and configure a set of groupware components to provide the range of collaborative work facilities they require. Thus all groupware components should inter-operate in a seamless manner, and must also adapt each other's user interfaces in a consistent manner.
- *Reuse suitable user interface, middleware and persistency components*. Where possible, groupware components should use similar or preferably the same packaged components that application components use to realise their user interface, communication (middleware) and data persistency needs. This reduces incompatibilities and redundancies in resulting groupware-enabled applications.

Many groupware applications have been built using ad-hoc approaches i.e. no specialised tool-kit or components. Examples include IRC and ICQ, Email, Grove[13], Netscape's Cooltalk, BSCW[23], Oz[16] and SPADE[15]. The main advantage with using standard distributed systems programming APIs for groupware applications is flexibility. The main disadvantages include a lack of high-level, reusable abstractions and components for building such tools (making their construction very time-consuming and difficult), and an inability for users (and often developers) to extend or reconfigure the groupware functionality of these applications[15, 17].

Because building groupware applications with standard APIs and frameworks is difficult, many groupware toolkits have been developed. Examples include GroupKit[1], Suite[6], Rendezvous[4], Meta-MOOSE[8], and PCTE[24]. These give developers built-in abstractions for constructing common groupware application facilities. Many successful groupware applications have been developed using these toolkits. However, most groupware toolkits suffer from "hard wired" facilities, a lack of extensible groupware facilities, and lack of ability of users to configure these facilities at run-time. In addition, applications must be built from scratch to use these toolkit facilities, and it is extremely difficult to integrate most groupware toolkit-built applications with existing applications, or extend existing applications with these toolkits. These problems lead to groupware applications with "fixed" functionality and often an inability of developers to provide some kinds of groupware facilities, as the toolkit they are using doesn't support it. Conversely, many applications built using such toolkits often exhibit "groupware bloat", with many of the groupware facilities provided by the toolkit unused but included in the applications anyway.

Component-based systems development technologies have become a popular approach to solving some common software development problems, such as lack of reuse and run-time configuration, and difficult-to-maintain, bloated, monolithic applications[25, 26]. Examples of component development methods and technologies include Catalasys™[27], Select Perspective™[28], COM[29], Enterprise JavaBeans[38] and OpenDoc[30]. Building groupware applications solely with standard

components shares the lack of abstractions and reusable components problem with using standard APIs and frameworks. However, some groupware applications have been developed using components alone, including CocoDoc[10], TeamWave[9], Netmeeting[12], and Orbit[31].

The use of specialised component frameworks for building groupware applications has been shown to offer benefits. Examples of such frameworks include Clock[3], our original JViews[19], COAST[7], Xanth[32], and using CORBA and OODBMSs[33]. Most of these approaches provide a component-based architecture with groupware functionality built into components in the architecture. Building groupware applications using such components is effective[19, 7, 3], but this still often results in problems such as fixed groupware functionality, component-bloat and lack of dynamically configurable and deployable groupware facilities. This is usually due to these groupware framework components themselves being hard-coded with specific kinds of groupware functionality. Most of these systems are oriented to quite limited problem domains e.g. synchronous groupware in COAST, tool integration in Xanth, and MVC-based synchronous editing environments in Clock. Most do not support user interface adaptations for components, making component integration difficult and resulting in poor quality interfaces. Some component architectures do support adaptive plug and play of components[34, 35], but to our knowledge have not been applied to component-based groupware development. Some aspect-oriented systems provide for the kinds of adaptations to running systems[36], but again to our knowledge have not been used for component-based groupware development.

OVERVIEW OF OUR COMPONENT-BASED GROUPWARE

In this section we illustrate various groupware component facilities using some simple scenarios based on extending the applications from Figure 1 with groupware capabilities. This was achieved by plugging groupware components into these three applications at run-time (and can also be removed at run-time).

Collaborative Editing

Consider John and Mark using a workflow system like that of Figure 1 (a) to describe their work processes (for some work domain – we use a simple software development process in the example below). They need to collaborate to view and edit these workflow diagrams. Initially they need to decide on the collaboration "level" – will they edit a workflow diagram synchronously (as one changes things the other immediately sees the changes made to their copy of the diagram) or asynchronously (they each edit independently then merge changes)? Figure 2 (1) shows John specifying, using a configuration menu of a "collaborating editing component", "action"-level editing (this is semi-synchronous – changes are sent to Mark as John makes them and vice-versa but the other users(s) can make changes at the same time i.e. no locking/waiting for changes to be made to view). A message is sent to both John and Mark using a text message component (2), indicating the change in editing state for the shared diagram.

If John and Mark had previously been editing the view asynchronously, John may want to see changes Mark made off-line and merge them in with his copy of the view. This is done using an editing event history component (3). John can select edits to have applied automatically to his view, or may message Mark to discuss edits he disagrees with. Editing histories can be checked in/out (as can whole diagram copies) from a version control component (4). This allows workers to manage multiple versions of the same diagram/document using deltas (edit event groups) or copies of the entire work artefact (e.g. diagram). John may create a new version of the workflow diagram before doing further edits/applying Mark's asynchronous edits to ensure the old information is not lost and can be later reviewed.

While editing the view, John and Mark need some cues as to what each is doing (so their edits don't clash). A multiple cursor component (5) shows where another user's cursor is and as Mark moves his mouse, this is refreshed on John's screen (typically each second or so). When Mark begins to change something, a highlighting component (6) shows John in-place annotations on diagram elements to keep him aware of Mark's in-progress changes.

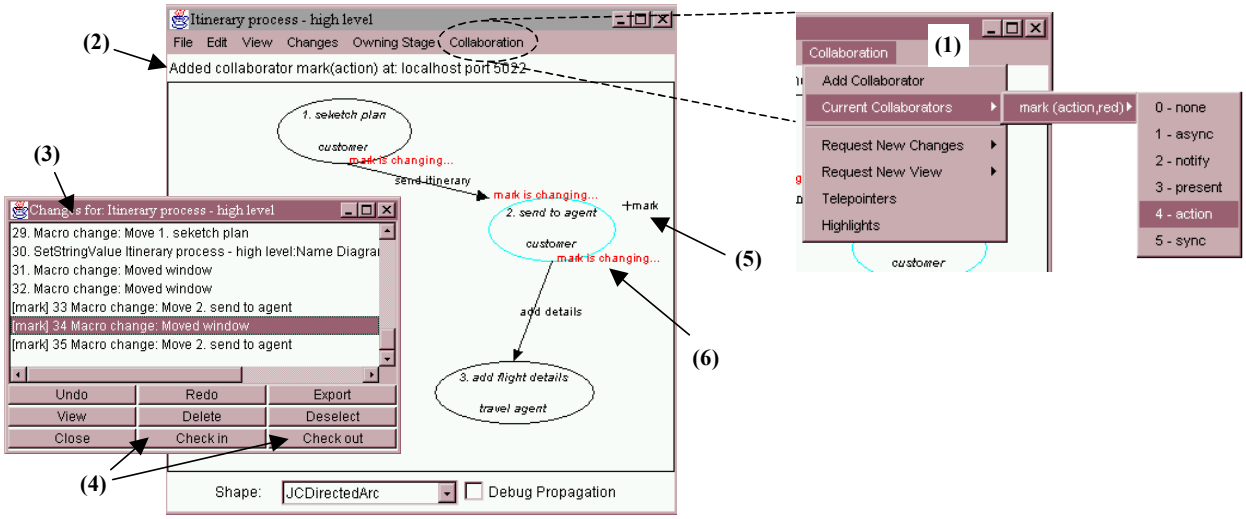


Figure 2. Examples of collaboration support groupware components in workflow tool.

Communications

Consider John and Mark collaborating to plan a trip. They share a travel itinerary editor (tree layout of trip details) and various visualisation views. One such high-level view is a map visualisation showing the legs of the travel (Figure 3). John and Mark may synchronously or asynchronously make changes to their shared travel plan. After changing the plan, John adds a note annotation on the map visualisation. This can later be clicked on and note read by Mark, John or other users (2).

If John and Mark are working asynchronously (one is off-line) they can communicate via email-style messages using an email component. If they are both on-line, they can use a text or audio chat (3) component to communicate and discuss their shared travel planning work. A text message component can be used to provide scrolling messages inside diagram and document windows, giving in-place messaging (4) as well as notification. Changes to the travel plan can be viewed (as in previous workflow example) with an event list component. The same component can be used by Mark to review a conversation with John (5), or by another user to view a conversation they missed.

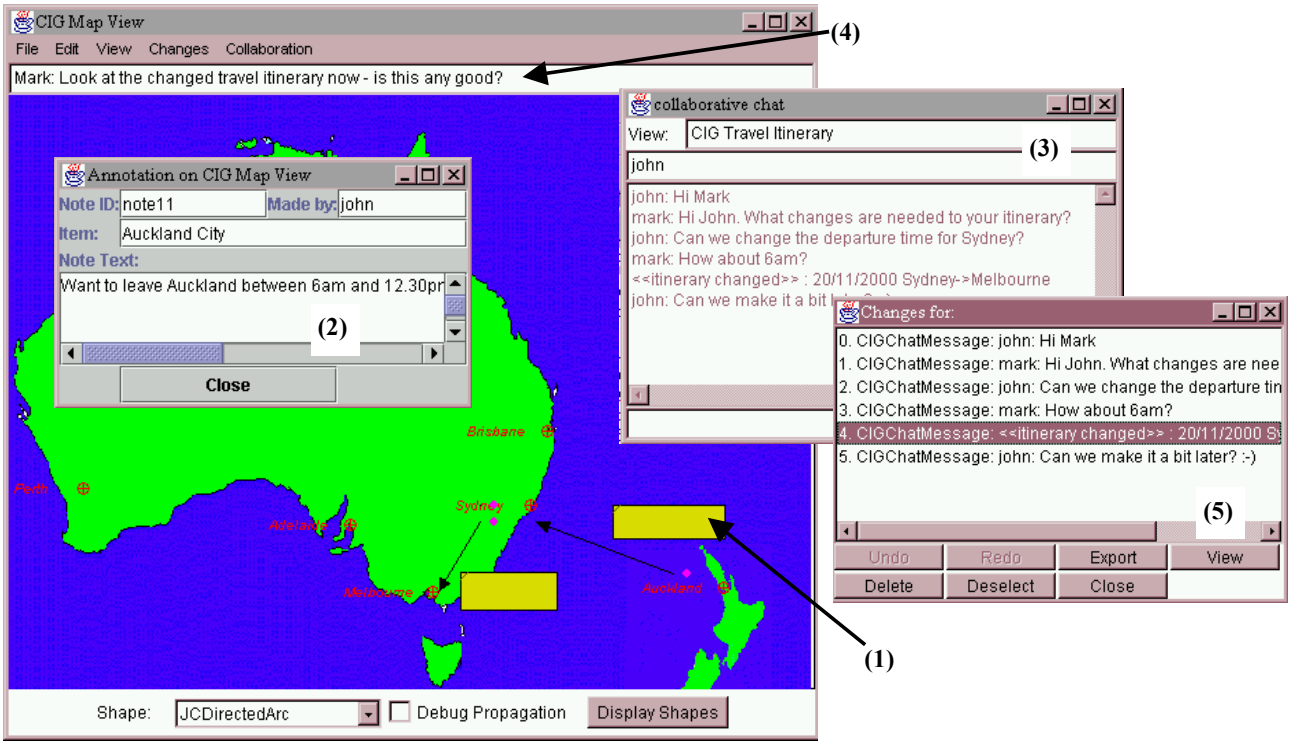


Figure 3. Examples of communications support groupware components in travel planner.

Co-ordination

Consider John and Mark editing a CASE diagram synchronously. At various times they want to be notified of changes that each other are performing, but may miss the transient awareness support described above (cursors and temporary annotations) e.g. because they briefly go to another window to do some work. A notifier component can be configured by Mark to inform him, in various ways, of changes John is making. For example, John may request a chat message be sent to him by the notifier when Mark makes a change (1) or a text message be sent(2). The notifier makes use of the chat and text messaging components to do this. In addition, when collaboratively editing the view, a locking component can explicitly deny access temporarily to components while they are being edited (3).

Because John may go off-line for some time, he may want annotations made to the document to inform him of Mark's changes. John may request e.g. a note annotation be automatically created (4) against changed items, or changed items highlighted when he goes back to this view (5). The notifier makes use of the note component and highlighter component to do this.

John and Mark wish to remain aware, not only of low-level work artefact modifications, but also higher-level tasks each is doing. A shared to-do list component is used to record tasks (6). The tasks can be manually entered by users, or obtained automatically from a workflow system like the one illustrated above. Task information can be displayed in work diagrams, for example using text messages or annotating the diagram. A to-do list component has annotated the CASE diagram window title (7) to indicate what task John is performing while modifying the diagram, allowing Mark to see this at a glance.

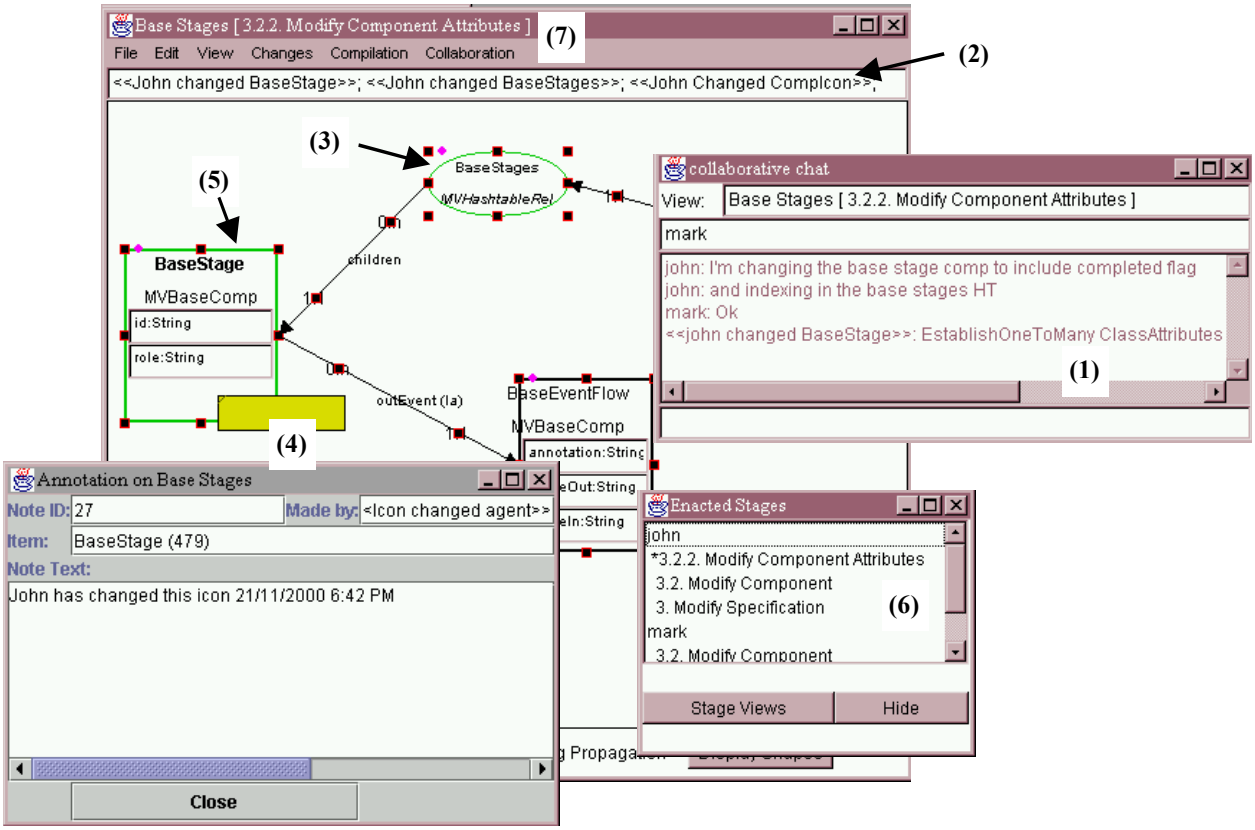


Figure 4. Examples of co-ordination support groupware components in CASE tool.

Plugging in Groupware Components

John and Mark need to plug groupware components into their applications in order to make use of the facilities described above. Figure 5 shows examples of the three ways they can do this. In (1), John opens a file containing groupware client code – when this is opened the component code in the file is loaded and the groupware component initialised. In (2), John uses a "Wizard" to deploy a selected groupware component from a set of available plug-in components (in this example, a notification component is being configured). These two approaches are appropriate for novice users as they hide details of the plug-in component architecture and links. In (3), John adds new components and connects them to existing components using a visual component deployment tool. This allows John to add in new groupware support from pre-existing components. This approach is suitable for expert users who wish to have flexible control over component creation and linkage.

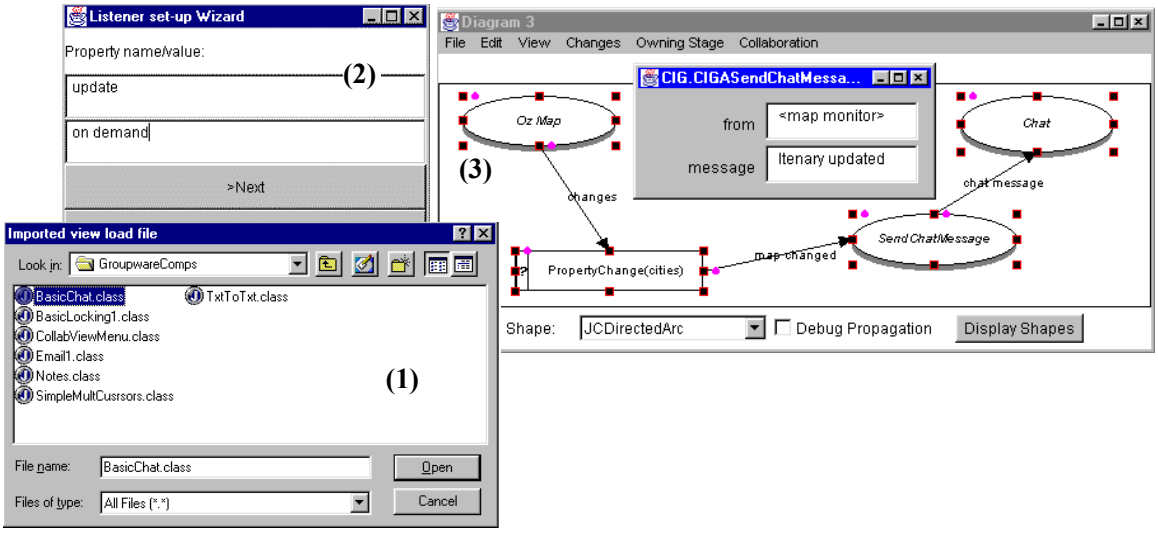


Figure 5. Examples of adding groupware components to an application.

When a groupware component is added to one of our applications, it may need to adapt parts of the interface the user of the application interacts with. For example, in Figure 2 additional menu items have been added to the workflow diagram menu by the collaborative editing support component when it was initialised. A versioning component added check in and check out buttons to the editing event history window. In Figure 3 the text messaging component added a message display field below the travel visualisation diagram's menu bar, and the email, note and chat components added menu items to the Collaboration menu to allow users to access their facilities. Such adaptations allow users to seamlessly interact with added groupware support. There are complex issues that arise when making such user interface adaptations that are beyond the scope of this paper to discuss in detail (e.g. What if two components want to add the same-named menu item? What if adding buttons changes the layout of a button panel? What if one component wants to disable a control and another assumes the control is accessible?). We discuss these issues in detail elsewhere[18].

ABSTRACT GROUPWARE COMPONENTS

In this section we identify a number of abstractions we have identified when developing groupware components as illustrated in the previous section.

Taxonomy

Table 1 shows taxonomy of groupware components that we have found useful when developing such systems. Some components provide client-side user interfaces to support collaboration (editing, versioning), communication (notes, chat, messages) or co-ordination (locking, to-do lists). Some provide server-side centralised data and event management (message exchange, event exchange and message, event and version histories). Some provide infrastructure services (building data and event sending/receiving services, extensible user interfaces and persistency management services).

Component Category	Examples	Description
Groupware clients		
Collaboration	Multiple cursors	Shows other users' cursor positions
	Collaborative editing client	Provides configuration and collaborative editing facilities for application elements
	Versioning client	Provides version control facilities
Communication	Chat client	Provides text-based chat between 2 or more users
	Email client	Email messages/documents
	Text message client	Scrolling text area in application's window frame
	Notes client	Note annotations
Co-ordination	Locking client	Highlights items other users are modifying
	To-do list client	
	Notification client	Provides configuration interface for notifier
General-purpose	Event history client	Provides list of edit, message, note, to-do item etc events
Groupware Servers (or "receivers" if Point-to-Point)	Message server	Used by chat, email, text message clients. Broadcasts messages to other users.
	Data/event history server	Used by message server, note, to-do list and version clients. Stores list of retrievable messages/events.
	Event server	Used by collaborative editing, notifier and multiple cursor clients. Broadcasts events to clients.
Groupware Infrastructure		
Data/event exchange	Data/event sender	Encodes data and events for sending client->: server or server->client
	Data/event receiver	Decodes data and events sent server->client or client->server
Persistency Infrastructure	Database access	For storing large numbers of small, discrete, well-structured data items
	File Access	For storing smaller numbers of unstructured data items
	XML Access	For storing moderate numbers of semi-structured data items
User Interface Components	Extensible menus, button panels, text areas, frames etc	Range of components allowing user interface building and run-time adaptation
Domain-specific components	Itinerary Editor/Server CASE diagram editor Workflow diagram editor Workflow engine …	Application components plugging groupware into

Table 1. A basic groupware component taxonomy.

Architectures

Groupware can be built using two fundamental architectural approaches: client-server and peer-to-peer[1, 19], as illustrated in Figure 6. Client-server has the advantage of simplicity and centralised data management, but the disadvantage of single point-of-failure and bottleneck in the server. Peer-to-peer has the advantage of robustness (no single server to fail) and flexibility, but is generally more complex to build with synchronising copied data often difficult. Our groupware components are designed and implemented to operate in either mode, and can also utilise a "hybrid" approach of utilising both client-server and peer-to-peer styles. If operating in peer-to-peer mode, groupware client components communicate directly with other groupware clients. If operating in client-server mode, clients communicate with a server that broadcasts events and data between clients as necessary. All our groupware components by default hold copies of shared data allowing peer-to-peer operation, but can use a server to synchronise data or provide a single, central data/event distribution point.

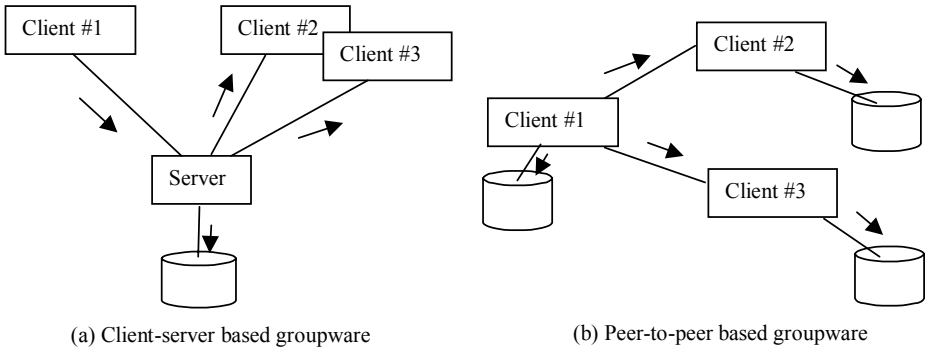


Figure 6. Client-server vs. peer-to-peer vs. hybrid groupware architectures.

Figure 7 illustrates the basic structure of our component-based groupware systems. Groupware client components share a common set of user interface and infrastructure components. They may also share event and local persistency (for peer-to-peer groupware) management components. Client components interact with domain-specific system components, like the travel itinerary editor and map, workflow system diagram editor and CASE tool diagram editor, detecting events and applying updates to the domain-specific interface. Clients communicate either with one or more groupware servers (if client-server operation) or one or more other groupware clients (if peer-to-peer operation). If operating in client-server mode, clients usually use the servers to store and retrieve groupware data. If operating in peer-to-peer mode, clients also store their own data locally.

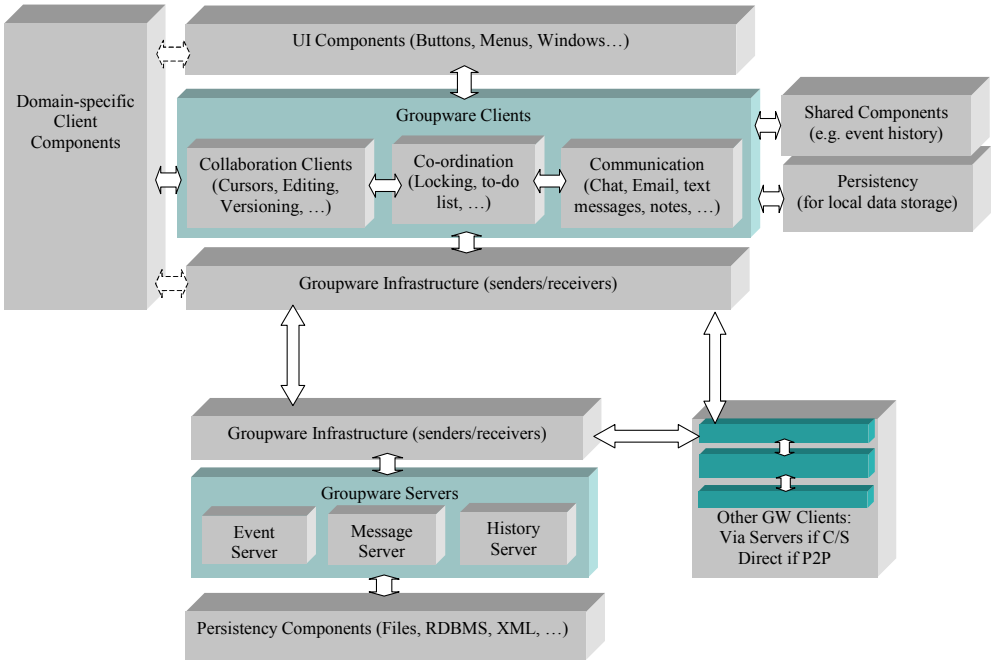


Figure 7. Basic groupware component infrastructure.

Collaborative Editing

To illustrate at a high-level how the groupware examples from the previous section are built using these groupware component abstractions we will consider three scenarios: collaborative editing and group awareness (via multiple cursors); editing event notification using text messaging and note annotation; and adding a new groupware component that must integrate itself with both domain-specific components and other groupware components.

Consider John and Mark collaboratively editing a workflow diagram, as illustrated in Figure 2. Figure 8 shows how the various components interact when a) John moves his cursor (keeping Mark

synchronously informed of John's workspace interactions), and b) John modifies a workflow icon (showing Mark the changes made). These examples both use a client-server architectural approach to connect the distributed groupware components. In the top example, John moves his cursor. The Multiple Cursor Awareness groupware component is informed of this event after the cursor has been moved, and sends a "mouse moved" event to the Event Server (using an Event Sender component). The Event Server broadcasts this to all interested users' multiple cursor components that have previously subscribed to other users' mouse move events. Mark's Multiple Cursor component receives notification of John's mouse move and displays a "cursor" for John in Mark's corresponding workflow diagram.

In the bottom example, John changes the name of a workflow stage. This results in the edit event(s) being sent to the event server. If fully synchronous editing is being done, the event server first checks if the data John is editing is "locked" i.e. being changed by another user. If not, it is locked for John's use and the edit event is stored by the history server and sent to others editing the same diagram. Mark's collaborative editing component receives the event(s) and applies updates to Mark's diagram. This can be done automatically or the events first presented to Mark for approval. John's Collaborative Editing component records the editing event(s) locally so they can be undone. Stored editing events can be exchanged as a block with another user to support asynchronous work i.e. the other users merges the stored changes with their own version of the diagram.

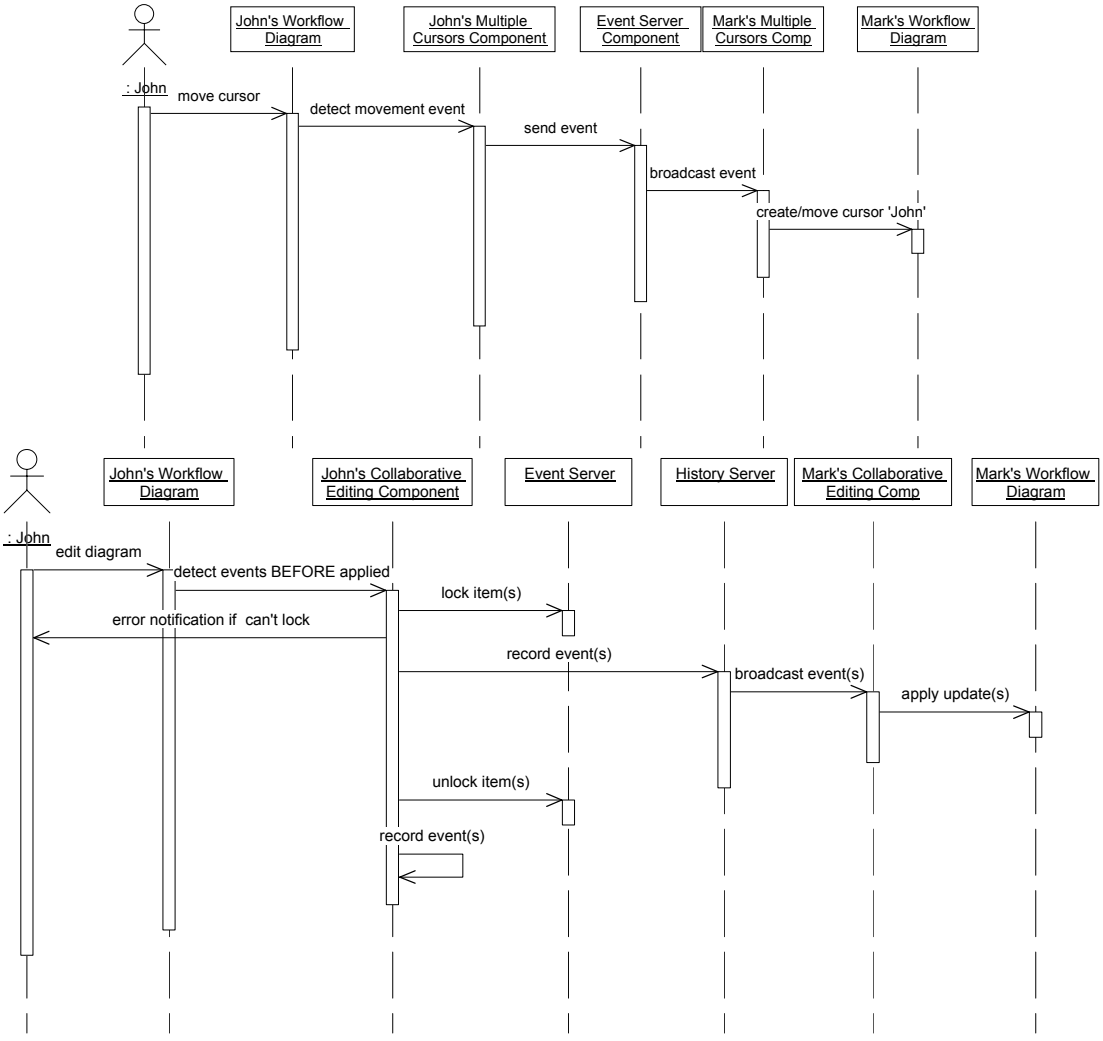


Figure 8. Example groupware component interactions: a) group awareness and b) collaborative editing.

Co-ordination

Consider John and Mark collaboratively planning a trip, as illustrated in Figure 3. When John changes a travel item Mark needs to be informed. This can be done synchronously via e.g. a text or chat message, or asynchronously via e.g. an email message or note annotation on the itinerary view. In the example in Figure 9 notification is done by a notification component using two communication components: a text message appears on Mark's screen and a note annotation is added to the changed item. These examples use a peer-to-peer architecture i.e. John's notification client communicates directly with Mark's message and annotation clients. As with the collaborative editing example above, a history component could be used to remember the editing or text message events.

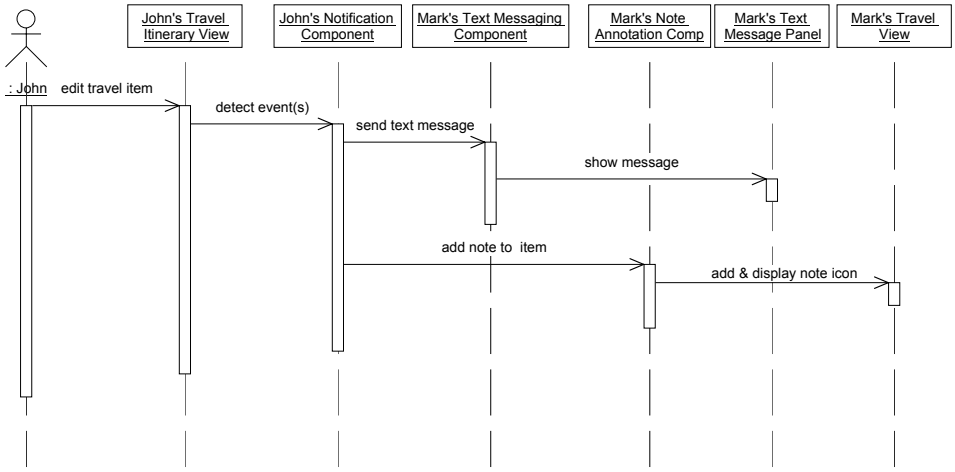


Figure 9. Example interactions: edit event notification via messaging and note annotation.

Plugging in Components

Consider John wanting to collaboratively edit a CASE tool diagram with Mark. John needs to add collaborative editing functionality at run-time i.e. the CASE tool doesn't currently have such a feature. Figure 10 illustrates the basic component interactions that take place. John chooses a collaborative editing groupware component from a list of available plug-in components for his CASE diagram. The collaborative editing component discovers the user interface elements the diagram provides and adds a set of menu items to the CASE diagram view, allowing John to configure collaborative editing facilities. The collaborative editing component then determines the events the diagram generates and the event sender available to the diagram for it to use. Finally, the collaborative editing component registers itself with an event server (or other users' environments directly, if a peer-to-peer architecture). John can then set the "level" of collaboration with Mark (e.g. asynchronous, synchronous or semi-synchronous) via the new menu items.

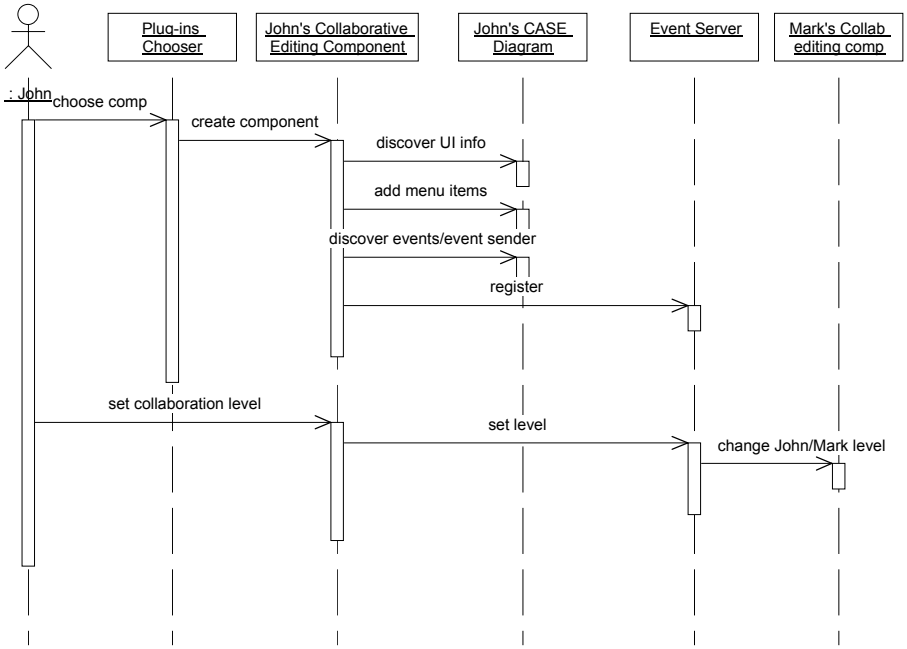


Figure 10. Example of adding a collaborative editing groupware component.

JVIEWS FRAMEWORK

Framework Abstractions

In order to build flexible groupware components like those outlined in the previous section a developer needs a range of abstractions with which to construct such components. One approach is to make use of a component-based class framework that encapsulates the fundamental building blocks for both the groupware and domain-specific components. Key features of such a framework should include:

- Components and inter-component relationships. These provide pluggable encapsulations of data and functionality, and manage inter-component relationship management.
- Component update events. These are generated when component or inter-component relationships are modified, or key events relating to the component's state occur.
- Before- and after- subscription to events. Components need to be able to be informed of changes to other components both after these have been made, but also BEFORE they are changed (e.g. to support locking and checking of shared components). JViews also allows components to receive notification of events being sent between two other components, enabling over-riding of their default event notification behaviour without code changes.
- Description of component functional and non-functional characteristics. The user interface, distribution and persistency management approaches used by components must be inspectable by other components so run-time adaptation and reuse of components can be adequately supported.

We have developed a framework called JViews, originally designed for building multiple-view, multiple-user design environments e.g. CASE tools, CAD tools, programming tools, workflow tools etc[19, 17]. JViews incorporates component-based building blocks for constructing such systems from reusable parts. When developing JViews, we originally added groupware capabilities to the framework classes i.e. hard-coded them in, in a similar fashion as done with GroupKit, COAST and Meta-MOOSE[9, 7, 8]. Thus while we used reusable component abstractions to built applications, our groupware support followed the "monolithic" application development approach, where developers had little control on what groupware support was included in specialised components, and this

resulted in sometimes inappropriately reused and over-complex component functionality[19, 18]. In addition, users were not given any control over what groupware functionality their environments supported, even if this was inappropriate to their specific needs.

Figure 11 illustrates the key abstractions in our JViews-based component framework. Components have attributes (state) and specialisations have operations for modifying state. Inter-component relationships may be simple zero-to-many links (component-to-component) or component-relationship component-component. Relationship components include hashtables (indexed) and vectors (sequential, unindexed). Events describe component state changes and a number of specialisations exist, including attribute and relationship changes. Aspects describe characteristics of the component e.g. user interface, distribution and persistency mechanisms provided or required.

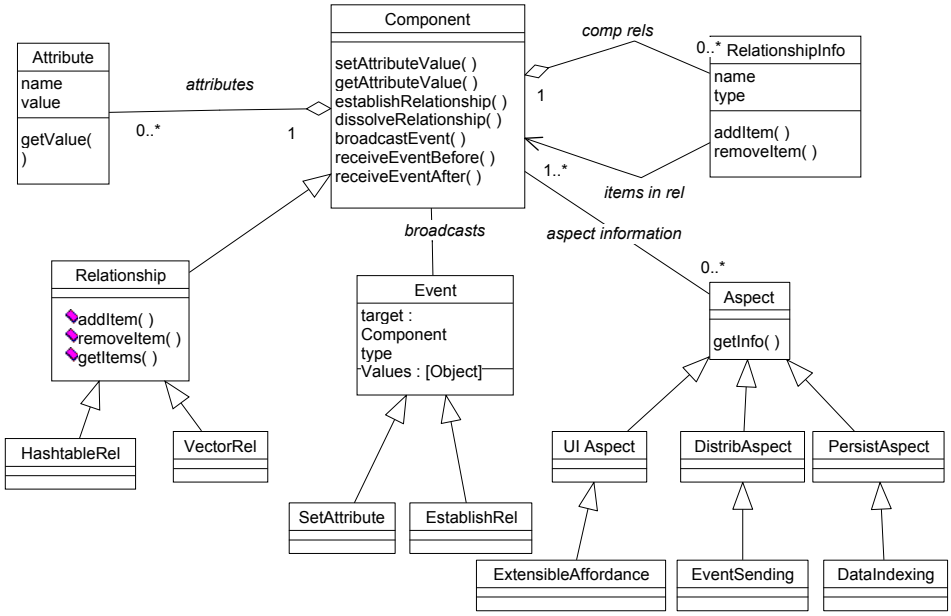


Figure 11. The JViews component-based framework.

The following table outlines some of the specialisations of these component, event and aspect framework classes.

JViews Abstraction	Examples	Description
Components	Base Components	Repository-level informational component. Used to model state shared between views in design environments.
	View Components	View-of-base (copy) – icon, connector, group, text etc. Used to model different views of the base components.
	Presentation Components	UI component e.g. window, panel, button, icon, connector, layout constraint etc.
	Relationship Components	Manages bi-direction connectivity between components. 1:1, 1:n, m:n, ordered (e.g. vector) and unordered, indexed (e.g. hashtable, B-tree) etc.
	Infrastructure components	Event/data sending/receiving; persistency management; 3rd party component integration
Events	Component Created	Component added
	Component Deleted	Component deleted – this is REVERSABLE in JViews – component not thrown away, just relationships to others dissolved (and can be undone)
	Attribute Modified	Set value (records before and after values of attribute so can be reversed easily)
	Established relationship	Components connected
	Dissolved relationship	Components disconnected
	UI Events	Window opened, button clicked, mouse moved etc
	Macro Events	Group of events – can be undone/redone/transported as a group
Aspects	UI Aspects	Frame, extensible panel, extensible menu, can be disabled/hidden, UI component information (size, colour etc), how display properties etc
	Distribution Aspects	Sends/receives data, transports data, encrypts/decrypts data, location information, performance information
	Persistency Aspects	Saves/loads data, produces data, storage mechanism used, query support
	Configuration Aspects	Software interfaces for configuration, UI for configuration, configuration properties

Table 2. Examples of JViews component, event and aspect abstraction types.

Collaborative Editing Components

Consider again the example of John and Mark collaboratively editing a workflow diagram. Figure 12 shows how this collaborative editing scenario, here using a client-server architecture, is realised using Jviews component abstractions. John's collaborative editing component subscribes to workflow diagram update events (1). When John changes a workflow diagram by direct manipulation (2), one or more events are created to describe the view state change about to be applied (3). These events are sent to the collaborative editing component (4) which then sends them across the network via the event sender (5) to the event server (6), which processes these events (7). The event server checks if the item(s) about to be changed by John are already locked e.g. Mark is editing them (8) and a response returned to John's collaborative editing component (9). If locked, this returns an error event to John's collaborative editing component which then aborts the attempted editing change(s) (10) and informs John of the concurrent editing clash. If unlocked, the event server locks the item(s) and sends the edit events to the history server (11). The history server component notifies all other users synchronously editing this view (12), and in this example Mark's collaborative editing component is informed (13), which then updates Mark's workflow diagram (14). Finally, John's collaborative editing component records the edit event(s) (15).

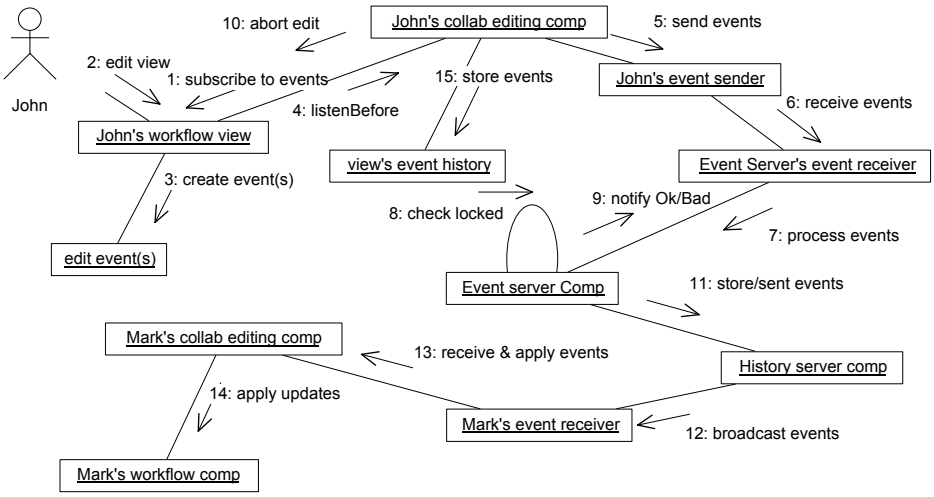


Figure 12. Example: JViews event handling for collaborative editing.

JViews uses the event sender and receiver components to achieve event distribution across a network (we have implemented socket, Java RMI and CORBA data transportation protocols for such components). In some situations the sending JViews application broadcasts the events whenever they occur, while in others filters are applied to the events to determine whether or not to transport them across the network. For example, to support multiple cursors the sending JViews environment records cursor movement events but only sends a single, absolute position of the user's cursor every second (by default – this can be changed by user preferences). This significantly reduces network overhead. Similarly, a notification component is told of all changes made to an application component, but we apply filters at the source and only sends events matching specified criteria to receiving components across the network. While such constraints can be specified in either source or target JViews application, sending many irrelevant events to the target system is costly.

Co-ordination Components

The implementation of a notification scenario is illustrated in Figure 13. In this example certain changes made by John to travel itinerary items e.g. item creation, arrive/depart date change etc, are indicated to Mark via synchronous text messaging and asynchronous note annotation. John's notification component subscribes to the events generated by the itinerary view (1). When John changes an itinerary item (2), events are created to describe this (3) and sent to the notification component after the change has been made (4). John's notification component then determines if the event meets the criteria describing the kind of change the user has specified. Send message/add note events are then created (5) and sent to Mark's note annotation and text messaging components (6, 7, 8, 9), instructing them to display a text message and annotate Mark's travel item view. When these events are sent to Mark's components the text messaging component formulates a new text message (8, 10) and displays this (12). Mark's note annotation component creates a new note (9, 11) and displays this in Mark's travel view (13) and records the note details for future perusal (14).

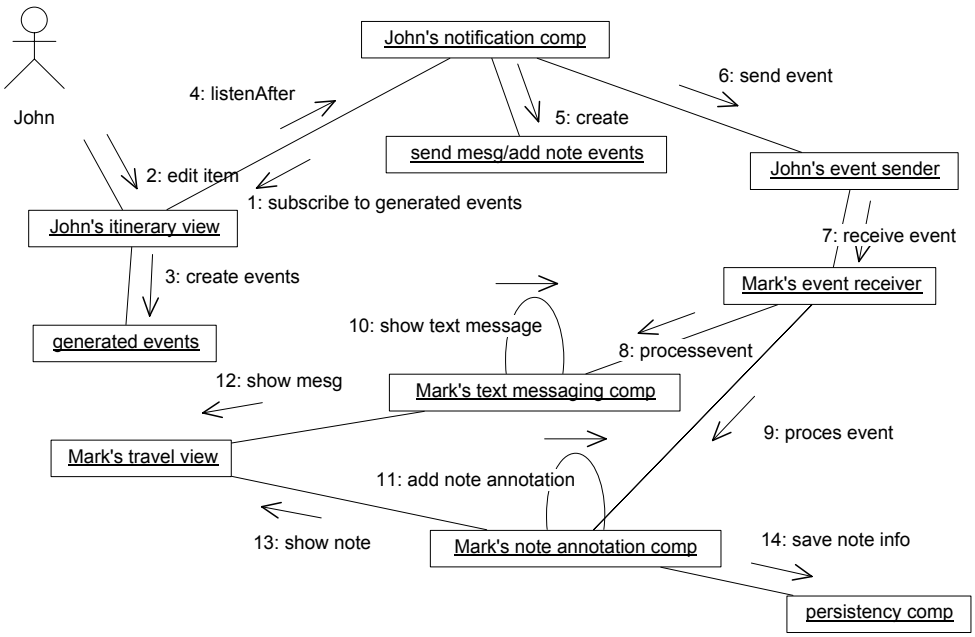


Figure 13. Example: JViews-implemented groupware notification and awareness components.

Plugging in Components

Our groupware components built using Jviews utilise "aspect" information encoding associated with other Jviews components to determine, at run-time, how to extend related component user interfaces, determine events to subscribe to, and to reuse related component distribution and persistency services[37]. To illustrate how this works, Figure 14 shows some interactions between a collaborative editing component and workflow diagram component when the collaborative editing component is initialised.

When the collaborative editing component is associated with e.g. a workflow diagram, it determines the available UI aspects of the diagram's component (1), so that it can extend the diagram's user interface (2) to add the collaborative editing configuration interface items it requires (3). In this example the collaborative editing component adds "affordances" (e.g. Add User, Set Collaboration Level, Send View, etc) by calling functions published by the workflow diagram's extensible menu aspect information (2). The extensible UI aspect object provided by the workflow diagram creates menu items to provide the required affordances (3) and adds these to the workflow diagram main menu bar (4). The collaborative editing component has no direct knowledge of where the items are added nor even that they are menu items (they could be implemented as pop-up menu items or buttons). Similarly, the collaborative editing component subscribes to events the workflow diagram generates (5-7) by introspecting the event generation aspect object the workflow diagram provides. It also determines and reuses the local event sender and persistency components used by the workflow diagram (8, 9) via its distributor and storage aspects. If the collaborative editing component is associated with a different diagram e.g. the travel planner map visualisation, then it will obtain the same kinds of aspect information but using these may provide quite different results e.g. pop-up menu items are added, a different set of events are subscribed to, and different distribution and persistency components are obtained.

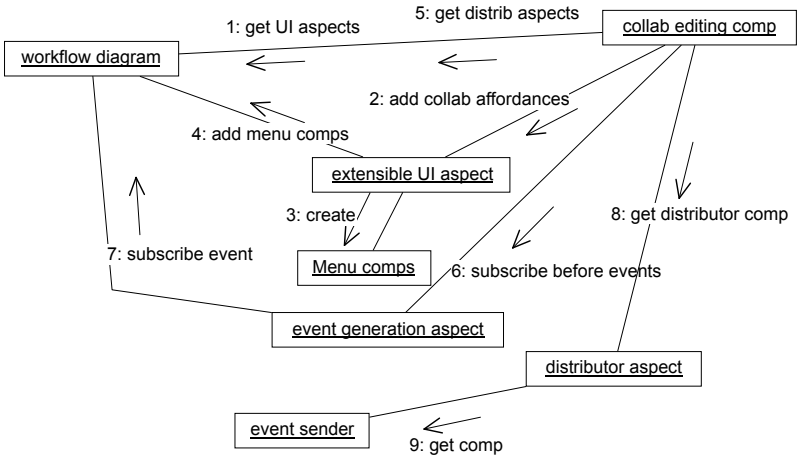


Figure 14. Example: adding groupware components and using JViews component aspects.

DESIGN AND IMPLEMENTATION

JViews Event Model

In order to implement the groupware components described in the previous sections, we need to have a component framework that provides sufficiently flexible run-time linking of components and handling of generated events. Jviews allows components to be linked by name at run-time without any compile-time knowledge of one another. We use a "relationship information" (RelInfo) class to do this. This allows any JViews component to be linked to one or more other Jviews components by a named relationship, and for the relationship to be queried by name. This inter-relating mechanism is also used to support flexible event subscription between Jviews components. Each named relationship link indicates when the related components should be told about events affecting a component. Relationships in Jviews are bi-directional so either end can listen for events affecting the other end of the relationship. Relationships can be named and used statically i.e. at compile-time, but for our groupware components most are constructed at run-time i.e. are dynamic. This means groupware components can establish relationships with domain-specific components, each other or infrastructural components, and none require any compile-time knowledge of each other.

Each Jviews component maintains a list of related components. This is a bi-directional relationship where each related component also maintains a reverse link to the component. When a component is deleted, all related components are informed of this so no "dangling links" are possible. Deletion of components can also be undone as the deleted component's relationship links are retained - only related component's links are updated to remove reference to the "deleted" component. The relationship link structure is illustrated in Figure 15 (a).

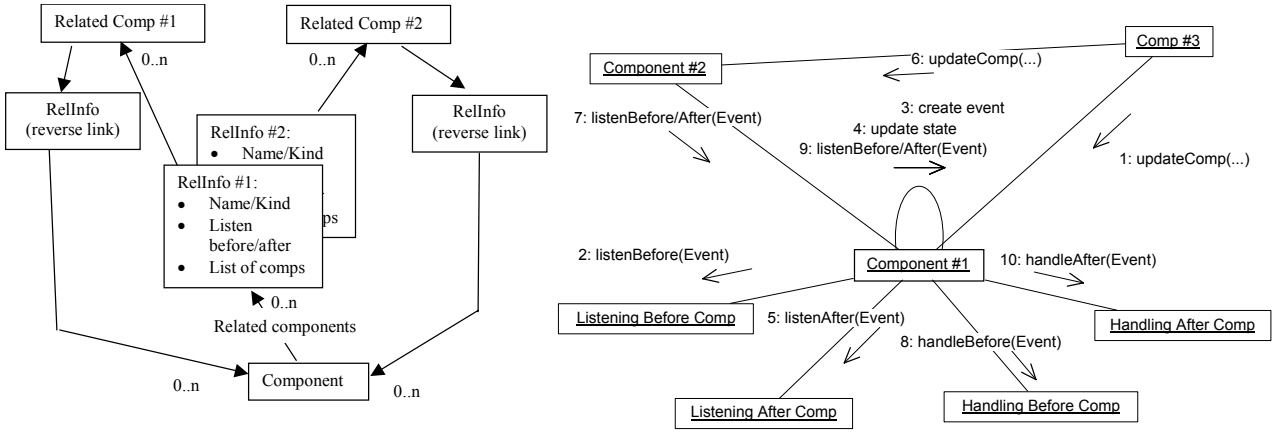


Figure 15. (a) Jviews relationship management and (b) flexible subscribe/notify.

Each Jviews RelInfo relationship-implementing object records when to tell all of the components it relates about state changes to the JViews component they belong to i.e. when to send events to them. To subscribe to events generated by another JViews component, a component must establish a relationship to that component (if none already exists) and then specify for that relationship when it wishes to receive the generated events. Events can be sent to related components:

1. before a component's state is actually changed i.e. "listen before changed";
2. after a component's state has been changed i.e. "listen after changed";
3. when a component has received an event from a 3rd component but before the component has handled the event itself i.e. "handle change before"; and
4. when a component has received an event from a 3rd component and the component has handled the event itself i.e. "handle change after".

Event Propagation

This subscription/notification mechanism allows components to monitor other components state changes before or after the changes have been made (1 and 2), as well as intercept events sent from one component to another before or after the receiver has handled the changes (3 and 4). Note that when a component specifies it wants to receive events from another JViews component, it receives ALL generated events i.e. there is no subscribe-to-specific-events in the basic JViews subscribe/notify model Figure 15 (b) illustrates this subscribe/notify mechanism in Jviews.

Other event models, such as those provided by JavaBean event listeners[39], Model-View-Controller architectures[40], and systems like Rendezvous' ALV[5] generally provide only "listen after changed" event subscription. This makes it very difficult to implement many groupware facilities like locking, some kinds of notification and versioning without application code changes to support these[41]. Some event models support listen-before notification, such as the ItemList[42] and ODGs[43]. The Jviews model is more flexible in that monitoring of components receiving events is also supported, allowing useful over-riding of event handling behaviour to be seamlessly added without the knowledge of the sender or receiver relationship structures. While Jviews doesn't provide subscription to specific kinds of events nor prioritisation of event notification directly, we have implemented reusable components that provide event filtering and priority-ordered distribution of notification.

Figure 16 illustrates the algorithm used to propagate events between related Jviews components. When updated, a Jviews component creates an event object to represent the change and then broadcasts this event to all components linked to it whose RelInfo objects indicate the related components want to be sent the event before the state change is done i.e. "listen before" subscription. We use this mechanism to implement locking and highlighting for our groupware components. If these components have "handle before" listening components i.e. ones that want to process the event before the listen before components, these components are sent the event to respond to first. This is used to implement some forms of collaborative editing and multiple cursors where these groupware components listen indirectly to event stores and window frame components, sent changes by view components. Responding components may throw an exception, causing the event to be aborted and not actioned, or return a null event object indicating event propagation should go no further (i.e. "consume" the event). Normally, the state change described by the event is then applied by the broadcasting component to itself. After the state change has been done, the component sends the event to all "listen after" components for them to respond to it. This is used to implement chat and email messaging, to-do list item updating and version exchange in our groupware components. Each listen after component may send the event to "handle after"

components once they have processed it. This is used to support recording of editing, message and co-ordination events in our groupware components.

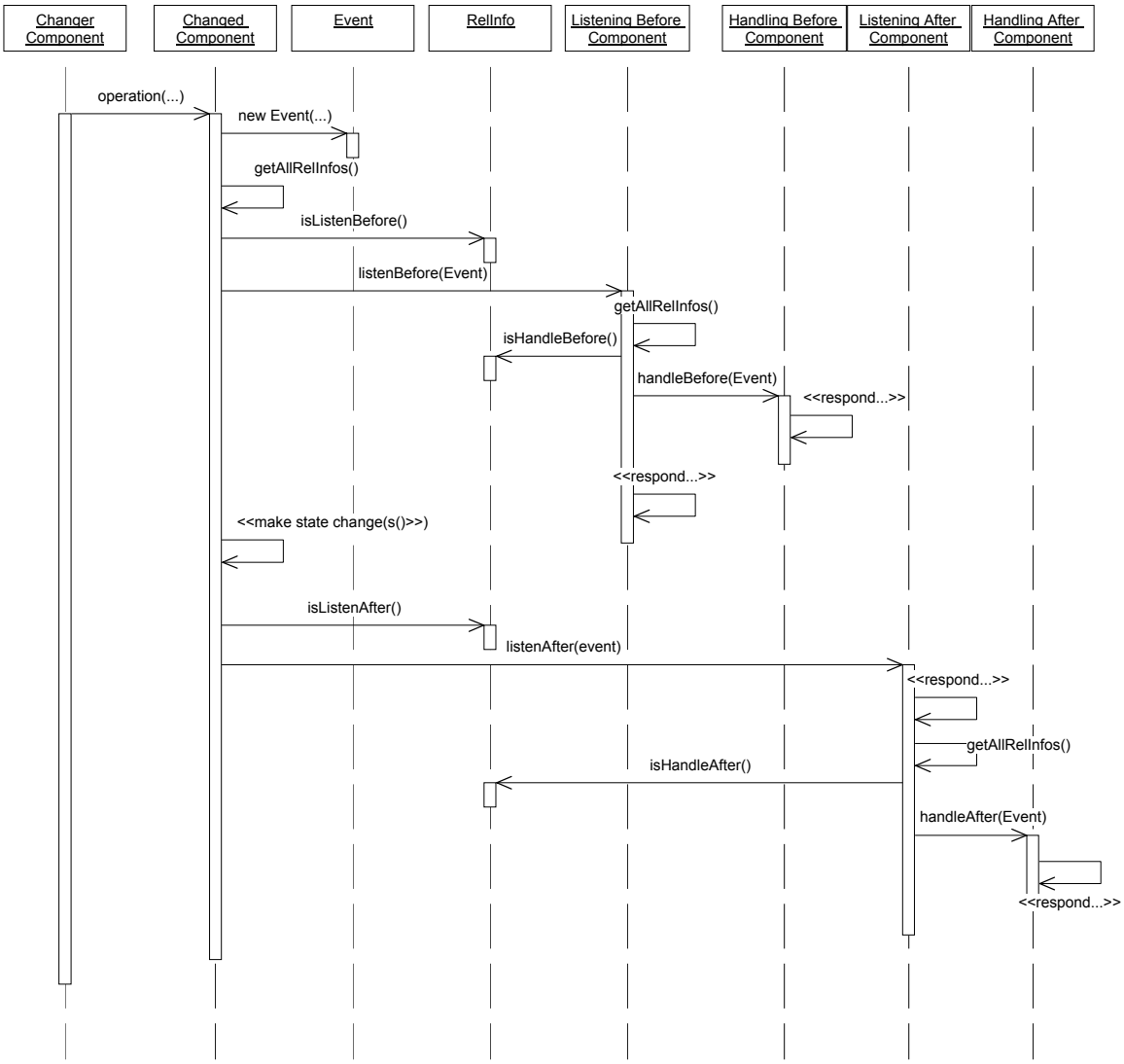


Figure 16. Event propagation in JViews.

We have developed several reusable components supporting the distribution of JViews events across a network and more refined event subscription mechanisms. Event distribution is done by sender/receiver components. The sender subscribes to Jviews component events and broadcasts these to one or more remote receivers (JViews component state transfer is supported by the same mechanism). A cross-application unique ID generation mechanism and component versioning are provided to allow multiple JViews applications to simultaneously maintain different versions of the same JViews components. This is needed for peer-to-peer architectures[17]. As noted in Section 5, care is needed when setting up such remote event broadcasting infrastructure to ensure wherever possible event filtering is done at source to minimise unnecessary event transport across a network.

Implementation Technologies

We have implemented our groupware components using several technologies. Our JViews component implementation framework is implemented in Java, and JViews components extend the JavaBeans component model. We added our extended event subscription/notification model to JavaBeans, together with our aspects codification technique and a number of abstractions to support multi-view tool development. We also developed a range of JViews components to provide reusable view editing tools, event histories, event and component distribution and persistency management.

These provide developers basic component building blocks to realise the applications and groupware components described previously.

Our distributed data and event management components supply developers with various abstractions for constructing distributed applications, specifically our groupware components. Originally we developed event and data broadcasting components using TCP/IP sockets, as provided by Java's APIs. We implemented a custom component and event data serialisation mechanism to marshal and demarshal JViews components and events between distributed applications. This used a textual data encoding and used parsing to decode received data. We also developed a custom remote component identification (naming) mechanism and custom component versioning support. We used point-to-point and client-server based application interaction, with an emphasis on point-to-point communications to improve application robustness and performance. These custom naming, serialisation and versioning techniques were necessary in order to support functions such as the replication-based flexible collaborative editing of views, identification of remote task automation and workflow agents, and maintenance of multiple versions of event and message histories. Our TCP/IP socket-based model works well in many application domains and for many of our groupware components. However, the text-based data encoding is quite slow and inefficient for high volume event inter-change. It also takes much of programmer effort to use the (very low level) socket-based distribution components, and this is error-prone and difficult to scale and extend. The custom naming and serialisation mechanisms make our components very difficult to combine with third-part tools, particularly for task automation.

We have retained our socket-based distribution management components for remote component versioning and multiple component data exchange. To support event exchange (e.g. multiple cursors, locking, messaging and remote workflow and notification event broadcasting) we have developed new distribution components using Java RMI and CORBA. The RMI-based event broadcasting components provide high-performance event subscription and notification facilities. They also support broadcasting a wide range of Java types far more easily than our custom socket-based protocol, allow improved remote component identification and location, and are easier for component developers to use. We use a modified form of our event serialisation and component identification mechanisms to encode component references, but let RMI encode all basic Java types and API objects. Our RMI event distribution components perform much better than our socket-based components for components with high-volume event exchange or for events with complex Java data types. They do not, however, offer any improvements in fault-tolerance and third-party component integration over the socket-based protocol.

We developed an additional component and event distribution facility using CORBA-based remote object interfaces to address these issues. Some components can be named, looked up and have their methods invoked remotely using these CORBA-based facilities. Occasionally we wanted to directly interact with third-party (i.e. non-JViews implemented components) components, particularly to support task automation and notification, and this technique is more general than using a custom socket protocol or RMI, and less work than building JViews component "wrappers" around third party remote components. There are also times when our groupware components want a more robust distributed method invocation infrastructure e.g. important notification agents and components supporting distributed component locking and version control. Visibroker™, the CORBA implementation we used, offers much better support for multiple remote object instantiation and transparent fault tolerance than our socket-based and RMI-based mechanisms.

An advantage of our textual component and event encoding is that data serialised using this facility can be used to make components persistent in indexed files. We originally provided persistency management components that stored component and events locally in serial or indexed files, or remotely via a shared file server. Like our socket-based component and data distribution

mechanism, this approach unfortunately proved difficult to program and not scalable for large applications. We have recently used an object database (PSE Pro™, a simplified form of the ObjectStore™ OO database) to improve persistency management. This provides a high performance local or remote component and event storage facility. We also extended JViews components to enable basic component versioning to be supported using the object database. One disadvantage is that while programmers could previously simply attach a persistency component dynamically at run-time to any JViews component and have it transparently stored and retrieved from a file, PSE Pro™ requires post-compilation and annotation of Java binary files. This means any JViews components to be made persistent with our PSE Pro™-based persistency components need this post-compilation annotation run on them before any instances have been created. This limits dynamic choice of a persistency mechanism by our groupware components.

EXPERIENCE

We have built a range of collaboration, communication and co-ordination groupware using our component-based approach. Collaboration groupware includes collaborative editing, including both synchronous and asynchronous editing support, multiple cursors, group awareness messages, and version control. Communication groupware includes text chat, email messages, scrolling text messages, and note annotations. Co-ordination groupware includes item locking, shared to-do list and workflow-based co-ordination and task automation. We have deployed these groupware components in a workflow system, a CASE tool, a travel planner, a software architecture design environment, a software component development tool, and a distributed system application generator. These groupware can be used independently or in groups, and all can be plugged into or removed from a tool at run-time. We have developed both peer-to-peer and client-server based versions of most of these groupware components.

Performance of the groupware implementations we have developed to date varies depending on the technology used, the architectural style of the groupware organisation and the number and locality of users. Considering response time, all of our groupware perform well when a small number of users on a local area network are being supported. If more than a dozen users are concurrently using groupware facilities, response time degrades due to bottlenecking of the shared servers. This can be partially overcome using peer-to-peer networking, but memory and local disk storage go up considerably, and each client utilises a socket connection for every active collaborator, which can exhaust available connections on some host machine configurations. Some groupware functions well over wide-area networks, such as low-bandwidth chat, email, note annotation, to-do lists and workflow events. Synchronous collaborative editing, text messaging and multiple cursors work across a modem connection but perform quite slowly i.e. a considerable time-lag can be experienced by the users. This is due to before/after event subscription for synchronous editing, and large numbers of small event sends for multiple cursors and automatically generated text messages.

While our JViews event mechanism and groupware components generate potentially many events, there may be ways to mitigate large numbers of network event broadcasting. These include using “multiplexing” of event sends i.e. having sender components group all (unrelated) events generated in a specific timeframe and forward as one group to the receiver(s) on other machines. Similarly, some of our event generating components, like the mouse movement events, are aggregated into a single event (e.g. one per second) and only this one event is broadcast. We plan to investigate giving users some control over such approaches, allowing them to tune aspects of event broadcasting to enhance application performance.

We prefer using a peer-to-peer architecture for our groupware as this is more reliable than the client-server variant: with all of our groupware facilities, a client can fail at any point and other collaborating users can continue to work unaffected (well – without being able to communicate and

collaborate with the failed client until it reconnects to the network). Synchronising shared data in peer-to-peer groupware requires support for multiple versions of data, version merging and conflict resolution. This complicates the implementation of some groupware facilities, particularly collaborative editing, to-do lists and note annotations. Our JViews component model provides these facilities but at quite a low level of abstraction (a detailed discussion of JViews component versioning can be found elsewhere[19]). We have found the client-server versions easier to implement and maintain, though at a cost of less reliable facilities and sometimes-poorer response times for users.

None of our groupware currently supports security protocols, apart from simple authentication to identify each user and simple access control lists for groupware administration. If deployed in situations where secure data access and transmission is required, encryption support needs to be added along with secure control lists to ensure proper authorised access to each groupware component's facilities and shared data. We have prototyped some "security" groupware components that encrypt/decrypt transmitted data using CORBA object wrappers, and some simple access control lists for some groupware clients (for to-do list item access/update control and version access/update/deletion control). These all require further enhancement for larger scale application deployment. Interaction of domain-specific application security approaches and reusable groupware needs to be investigated, and handled in a similar way to our persistency and distribution component sharing via aspects.

Our groupware components have been evaluated as parts of two usability studies, one focusing on using our workflow system in a software development setting[17] and one on our collaborative travel planner[18]. Both of these studies had small groups of users carrying out collaborative tasks with the workflow and travel planner systems and groupware facilities. The main usability problems reported from these studies include partial use of AWT-based Java user interface elements and modal-based diagram editing within our design environments. While these are not groupware component-specific issues, they adversely impact on the usability of these components within these applications. In general, users found the range of groupware facilities suitable and the basic functionality intuitive. The collaborative editing facilities have proven to have very good flexibility, though they take some time to become accustomed to[44]. Adding groupware components to environments is difficult for non-expert users, requiring the provision of a more effective, easy-to-use end user component deployment tool.

We are extending our work with component-based groupware in a number of ways. We have been developing new techniques for adaptive user interface support we hope to use with our groupware components to improve the ability of developers to build adaptable interfaces. We have extended our aspect-oriented development approaches to provide richer descriptions of components using XML-encoded information, which we plan to use to describe our groupware and provide more automated run-time initialisation of components. We have recently been building thin-client groupware component prototypes. These use HTML- and WML-based user interfaces with all groupware functionality in web server components (realised by Java Server Page, CORBA and Enterprise Java Bean implementations). The integration of the thick-client groupware described in this paper and these thin-client groupware applications is an area of keen interest. We are looking at re-designing our server-side components to utilise the Enterprise Java Bean architecture and technology to increase their reusability and run-time deployment capabilities. We are developing a new infrastructure for event-based systems using SOAP-style web services and asynchronous messaging protocols, and plan to experiment with re-implementing some of our groupware communications using these architectures and technologies. This will hopefully provide more generalised interfaces to our groupware components and allow for greater reuse of our components with 3rd party groupware and applications using other implementation technologies.

SUMMARY

Many applications require various kinds of groupware functionality. We have developed a range of plug-in groupware components that provide various group awareness capabilities. The key contributions of our work include:

- Our approach to extending component-based systems by dynamically plugging in appropriate groupware-supporting components. These extend applications to provide the desired groupware capabilities in a seamless fashion. Groupware component facilities can also be statically specified by application developers, and be turned off (unplugged) by end users if not required.
- Our approach is elegant in that no code modifications to neither groupware nor application components are necessary - the architectural facilities of our JViews component framework are leveraged to obtain the necessary application component (and groupware component) reconfiguration to provide the groupware facilities.
- We have developed an architectural model, framework design and implementations that support a wide range of plug-in groupware capabilities, each expressed in discrete, separately reusable software components. These groupware components are mutually compatible and where possible share similar user interface, persistency and distribution components.
- We have developed a flexible event subscription and notification mechanism, and aspect-based introspection and decoupled interaction mechanism, both used to realise out groupware components. Such architectural and framework design features, and our implementation techniques for them, can be used to support other kinds of dynamic component-based systems extension and reconfiguration (such as for persistency and distributed processing, partially explored during our development of groupware components).

Our groupware components are generic and have been successfully reused in a diverse range of applications, providing a wide range of collaborative work-supporting facilities required by end users. Based on our experiences, we suggest the following to others developing component-based groupware applications:

- Carefully identify component responsibilities within the architecture aiming to maximise reuse of abstractions e.g. user interface components, event broadcasting components, data persistency components, locking co-ordination, inter-person messaging, and so on. A good set of component abstractions makes groupware development much easier and results in more maintainable solutions.
- Peer-to-peer architectures work well and provide robust groupware facilities, but require more complex data mirroring, data synchronisation and have a degree of unproven scalability, particularly in the presence of a large number of event exchanges.
- Components need to be engineered to support event-based interaction, ideally before-change and after-change event subscription for synchronous groupware facilities, and a well-defined set of common events makes integrating components easier.
- Components need to publicise information about their events and services in order to allow run-time plug and play and suitable automatic adaptations to component configurations to be built. This is not easy, and requires careful thought, design and implementation. Our aspect characterisations are one of many possible techniques that can be used, though emerging standards for new component technologies like .NET and J2EE may provide more generic support for component introspection.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the many helpful comments of the anonymous reviewers on earlier versions of this paper.

REFERENCES

1. M. Roseman and S. Greenberg, Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer-Human Interaction 3,* 1 (March 1996), 1-37.
2. T.C.N. Graham, C.A. Morton, C.A. and T. Urnes, ClockWorks: Visual Programming of Component-Based Software Architecture. *Journal of Visual Languages and Computing*, (July 1996), 175-19
3. T. Urnes and T.C.N. Graham, Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations. In *Proceedings of Design, Specification and Verification of Interactive Systems*, 1999.
4. R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson and W. Wilner, The Rendezvous Architecture and Language for Constructing Multi-User Applications. *ACM Transactions on Computer-Human Interaction 1*, 2 (June 1994), 81-125.
5. R.D. Hill, The Abstraction-Link-View Paradigm: Using Constraints To Connect User Interfaces to Applications. In *Proceedings of CHI '92: Human Factors in Computing,* ACM Press, 1992, pp. 335-342.
6. P. Dewan and R. Choudhary. (1991) Flexible user interface coupling in collaborative systems. In *Proceedings of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.
7. C. Shuckman, L. Kirchner, J. Schummer and J.M. Haake, Designing object-oriented synchronous groupware with COAST. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, ACM Press, November 1996, pp. 21-29.
8. R.I. Furguson, N.F. Parrington, P. Dunne, J.M. Archibald and J.B. Thompson, MetaMOOSE – an Object-oriented Framework for the Construction of CASE Tools. In *Information and Software Technology 42*, 2 (January 2000).
9. M. Roseman and S. Greenberg, Simplifying Component Development in an Integrated Groupware Environment. In *Proceedings of the ACM UIST'97 Conference,* ACM Press, 1997.
10. G.H. ter Hofte and H.J. van der Lugt, CoCoDoc : A framework for collaborative compound document editing based on OpenDoc and CORBA. In *Proceedings of the IFIP/IEEE international conference on open distributed processing and distributed platforms*, Toronto, Canada, May 26-30, 1997. Chapman & Hall, London, 1997, p. 15-33.
11. S.M. Kaplan, W.J. Tolone, A.M. Carroll, D.P. Bogia and C. Bignoli, Supporting Collaborative Software Development with ConversationBuilder. In *Proceedings of the 1992 ACM Symposium on Software Development Environments,* ACM Press, pp. 11-20.
12. Microsoft Corp., *Microsoft NetMeeting 2.1*. See: http://www.microsoft.com/netmeeting/.
13. C.A. Ellis, S.J. Gibbs and G.L.Rein, Groupware: Some Issues and Experiences. *Communications of the ACM 34*, 1 (January 1991), 38-58.
14. B. Magnusson, U. Asklund and S. Minör, Fine-grained Revision Control for Collaborative Software Development. In *Proceedings of the1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, Los Angeles CA, December 1993, ACM Press, pp. 7-10.
15. S. Bandinelli, E. DiNitto, and A. Fuggetta. Supporting cooperation in the SPADE-1 environment. *IEEE Transactions on Software Engineering 22,* 3 (December 1996), 841-865.
16. I.Z. Ben-Shaul, G.T. Heineman, S.S. Popovich, P.D. Skopp, A.Z. Tong, and G. Valetto, Integrating Groupware and Process Technologies in the Oz Environment. In *Proceedings of the 9th International Software Process Workshop:The Role of Humans in the Process,* IEEE CS Press, Airlie, VA, October 1994, pp. 114-116.
17. J.C. Grundy, J.G. Hosking, W.B. Mugridge and M.D. Apperley, A decentralised architecture for software process modelling and enactment. *IEEE Internet Computing 2*, 5 (Sept/Oct 1998), IEEE CS Press, pp. 53-62.
18. J.C. Grundy and J.G. Hosking, Developing Adaptable User Interfaces for Component-based Systems. *Interacting with Computers 14,* 2 (March 2002), Elsevier Science Publishers, pp 175-194.
19. J.C. Grundy, W.B. Mugridge and J.G. Hosking, Constructing component-based software engineering environments: issues and experiences. *Journal of Information and Software Technology 42*, 2, (January 2000), pp. 117-128.
20. T.C.N. Graham and J.C. Grundy, External Requirements of Groupware Development Tools. In *Engineering for Human-Computer Interaction*, Chatty, S. and Dewan, P. Eds, Kluwer Academic Publishers, August 1999, 363-376.
21. P. Dourish and V. Bellotti, Awareness and coordination in shared workspaces, In *Proceedings of the 1992Conference on Computer-Supported Cooperative Work*, Toronto, Canada, 1992, ACM Press, pp. 107-113.
22. A.J. Dix, Computer-supported cooperative work – a framework. In *Design Issues in CSCW*, Rosenburg, D. and Hutchison, C., Springer Verlag, 1994, pp. 23-37.
23. R. Bentley, T. Horstmann, K. Sikkel and J. Trevor, Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. In *Proceedings. of the 4th International WWW Conference,* Boston, MA, December 1995.
24. L. Wakeman and J. Jowett, *PCTE: the standard for open repositories*. Prentice-Hall, 1993.
25. A.W. Brown and K.C. Wallnau, Current state of Component Based Software Development. *IEEE Software* (Sept/Oct 1998), 37-46.
26. C.A. Szyperski, *Component Software: Beyond Object-oriented Programming*. Addison-Wesley, 1997.
27. D.F. D'Souza and A. Wills, *Objects*, *Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.

28. P. Allen and S. Frost. *Component-Based Development for Enterprise Systems: Apply the Select Perspective™,* SIGS Books/Cambridge University Press, 1998.
29. R. Sessions, *COM and DCOM: Microsoft's vision for distributed objects*. Wiley, 1998.
30. Apple Computer Inc., *OpenDoc Users Manual*, 1995.
31. S.M. Kaplan, G. Fitzpatrick, T. Mansfield and W.J. Tolone, Shooting into Orbit. In *Proceedings of Oz-CSCW'96,* University of Queensland, Brisbane, Australia, August 1996.
32. G.E. Kaiser and S. Dossick, Workgroup middleware for distributed projects, In *Proceedings of IEEE WETICE'98*, Stanford, June 17-19 1998, IEEE CS Press, pp. 63-68.
33. W. Emmerich, CORBA and ODBMSs in Viewpoint Development Environment Architectures. In *Proceedings of the 4th International Conference on Object-Oriented Information Systems,* Springer Verlag, 1997, pp. 347-360.
34. M. Mezini and K. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of OOPSLA'98*, Vancouver, WA (October 1998), ACM Press, pp. 97-116.
35. M. Mezini, L. Seiter and K. Lieberherr, Component Integration with Pluggable Composite Adapters. *Software Architectures and Component Technology*, M. Aksit, Ed, Kluwer, 2000.
36. J.L. Pryor and N,A. Bastan, Java Meta-level Architecture for the Dynamic Handling of Aspects. In *Proceedings of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vagas (June 26-29 2000), CSREA Press.
37. J.C. Grundy, Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering 10*, 6 (December 2000).
38. R. Monson-Haefel, *Enterprise JavaBeans*. O'Reilly, 1999.
39. J. O'Neil and H. Schildt, *Java Beans Programming from the Ground Up*. Osborne McGraw-Hill, 1998.
40. G.E. Krasner and S.T. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (1988), 8-22.
41. J.C. Grundy, J.G. Hosking and W.B. Mugridge, Supporting flexible consistency management via discrete change description propagation. *Software - Practice and Experience 26*, 9 (September 1996), Wiley, 1053-1083.
42. R.B. Dannenberg, A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors. *Software-Practice and Experience 20*, 2 (February 1990), 109-132.
43. M.R. Wilk, Change Propagation in Object Dependency Graphs. In *Proceedings of TOOLS US '91,* Prentice-Hall, August 1991, pp. 233-247.
44. J.C. Grundy, Engineering Component-based, User-configurable Collaborative Editing Systems. In *Proceedings of 1998 International Confernce on Engineering for Human-Computer Interaction*, Crete, Greece, Sept 14-18 1998, Kluwer Academic Publishers.