In proceedings of the 1998 Australian Workshop on Software Architecture, Melbourne, Nov 1998.

Tool integration, collaboration and user interaction issues in component-based software architectures

John Grundy[†], Rick Mugridge[††], John Hosking[††] and Mark Apperley[†]

Department of Computer Science
University of Waikato
Private Bag 3105
Hamilton, New Zealand

{jgrundy, M.Apperley}@cs.waikato.ac.nz

Phone: +64-7-838-4452
Fax: +64-7-838-4155

Department of Computer Science
University of Auckland
Private Bag 92019
Auckland, New Zealand

{john, rick}@cs.auckland.ac.nz

Phone: +64-9- 373-7599 ext 8914
Fax: +64-7- 373-7453

Abstract

Component-based software architectures are becoming increasingly popular solutions for use in a wide range of software applications. Particular areas in which these architectures may provide improved software development support include tool integration, distribution and collaborative work support, and human interaction and end-user configuration. However, a number of open research issues exist to do with the deployment of component-based solutions in these areas. We review our recent research experiences in deploying component-based solutions in these problem domains, and overview potential research directions.

1. Introduction

In recent years there has been an increasing interest in the use of component-based software architectures (also known as "componentware"). These architectures use the notion of a software component object, which advertises its methods, properties and events for use by other components, and use large-scale component composition to build up software applications. This contrasts with traditional software construction using libraries and frameworks, which results in "monolithic" software applications that are difficult to build and dynamically reconfigure. Component-based systems usually allow end-users to reconfigure the components that make up their applications, and "plug and play" a range of third-party components.

Various software architectures have been developed using components, including JavaBeans [Javasoft, 1997], COM/DCOM [Microsoft, 1998] and OpenDoc [Apple, 1995]. Tools allowing such architectures to be used to specify components and component-based applications include JBuilder [Borland, 1997], Visual Javascript [Netscape, 1998] and Visual Age [IBM, 1997]. Component-based solutions offer great potential for reusing components that support tool integration, collaborative work and object distribution, and end-user interaction and configuration of applications. Component-based solutions contrast to more traditional approaches such as federation [Bounab and Godart, 1995], toolkits and frameworks [Linton-et-al, 1989], and UIMS [Myers et al, 1997] used to implement these aspects of software applications. However, component-based approaches require careful design to make reuse of these aspects straightforward.

We describe our recent research into component-based software architectures. This focuses on support for component-based architectures, component-based tool integration and collaborative work support, and issues of end-user interaction with and reconfiguration of component-based software systems. Examples of systems we have developed using component-based solutions are discussed, with particular emphasis on difficult and unsolved research issues. Our research has led to the formulation of a design and implementation approach we call "aspect-oriented" component-based system development. We briefly discuss this approach and illustrate prototype tool support for it, which endeavours to characterise aspects of components concerned with human-interfaces, end-user configuration, distribution and collaboration.

This paper begins with an example component-based software application, a process modelling and enactment environment. The following sections overview the use of components in supporting tool integration, collaborative work, human interaction and configuration in this environment and its software architecture. The use of aspects to characterise parts of components supporting these facilities is examined, and our research compared and contrasted to related architectures and systems. We conclude with a discussion of our future research directions and summarise our experiences to date with component-based software architectures.

2. Problem Domain

Figure 1 shows a screen dump from Serendipity-II, a component-based workflow management system [Grundy et al, 1998a]. The top two windows show parts of a work process model (in this case, a simple process for modifying a software system). The dialogues show information about process enactment (i.e. work history), and a to-do list for each user of the system. This application allows users to design models of their work processes, enact (run) these process models and use them to guide their work, and to track actual work performed using the processes [Grundy et al, 1998a; Grundy and Hosking, 1998].

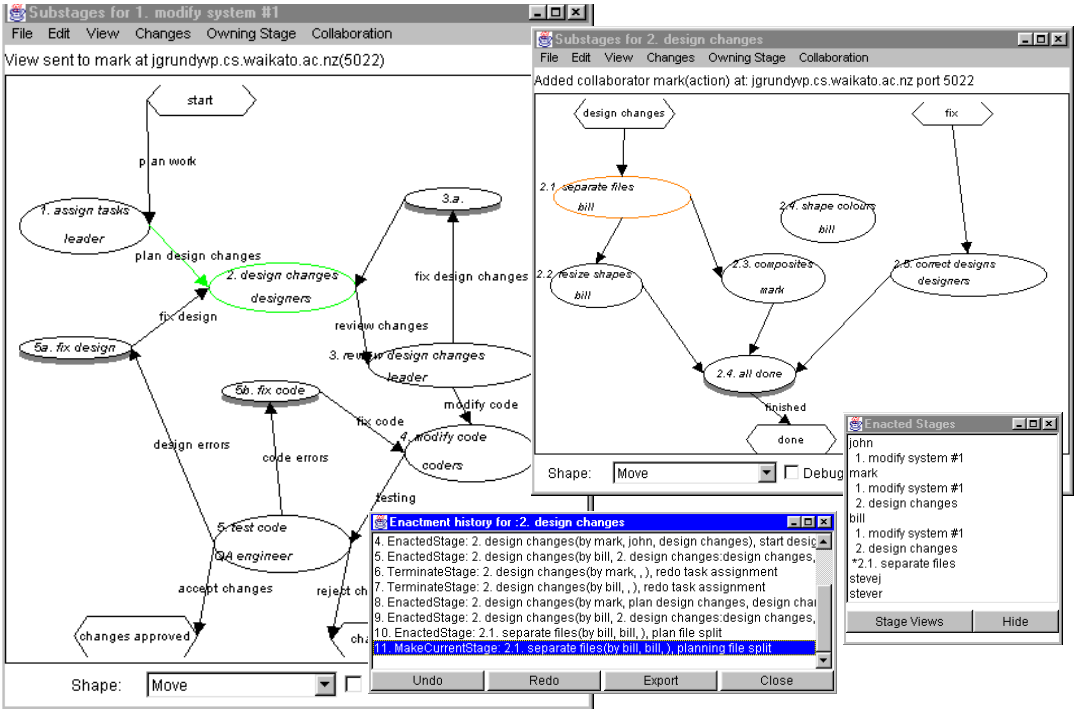


Figure 1. Serendipity-II: example of a component-based software system.

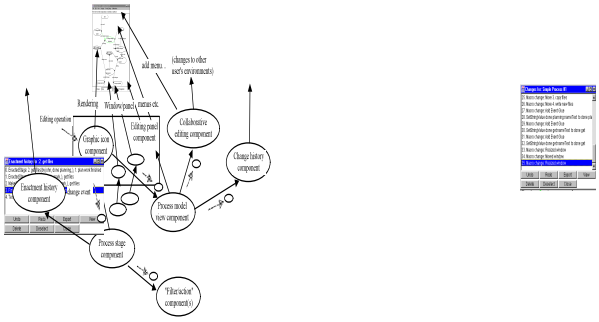


Figure 2. The component-based architecture of Serendipity-II.

Software components are used throughout the implementation of Serendipity-II. Figure 2 illustrates the software architecture of part of Serendipity-II. Icons and editing tools used for the graphical views are all reusable components that we have deployed in other applications. These are linked to application-specific "process stage" and "process model view" components, used to represent process model information. The Collaboration menu is the interface to a "collaborative editing" component, also reused elsewhere. The enactment history dialogue is the interface to a generic "change history" component that is used throughout our component-based applications to store events and make these accessible to users. Users can add components on-the-fly, which we call event filters and actions, to provide useful additional functionality, modify how existing facilities of Serendipity-II work, or provide interfaces to third-party systems [Grundy et al, 1998a]. Event propagation is used extensively to keep process models consistent, record change histories, support collaborative editing, and drive process enactment. The shaded lines in Figure 2 indicate propagation of "change event" objects between components in Serendipity-II.

Serendipity-II illustrates several uses of component-based software architecture solutions:

- *Tool integration*. We have integrated Serendipity-II with several other tools, including communications software, a file sharing server, CASE tools, programming environments, and office automation tools. These third party tools were integrated with Serendipity-II process models by using filters and actions to "wrap up" the functionality of the tools, and thus provide component-based interfaces to them.
- *Distribution and collaborative work*. Serendipity-II environments support multiple users, and thus process models and enactment information needs to be both distributed and collaboratively accessed and edited. Components comprising the architecture must support distribution and appropriate collaborative editing facilities must be provided to users. Our component-based software architecture provides basic component object distribution and event propagation mechanisms, which are leveraged by components in Serendipity-II to provide versioning, configuration management and collaborative editing facilities.
- *End-user interaction and configuration*. Users of Serendipity-II need to be able to interact with the components that make the environment in appropriate and consistent ways, and be able to plug in new components and reconfigure existing components. As Serendipity-II is comprised of a variety of reusable components, these should provide a consistent look-and-feel to users. Users of the environment are provided with both simple configuration "wizards" to add new components and reconfigure components, and with advanced, visual interfaces to carry out complex component configuration. This allows users to extend their environment's functionality and to integrate other tools with Serendipity-II process support facilities.

We have developed a novel component-based software architecture and object-oriented class framework called JViews [Grundy et al 1998c]. JViews was designed to facilitate the development of multiple view, multiple user design environments and software engineering tools. JViews includes abstractions to assist with repository and view structure and semantic information modelling, view consistency management, versioning and collaborative editing, component distribution and persistency. To facilitate the development of component-based software architectures, we have developed the JComposer CASE tool and BuildByWire graphical user interface construction tool [Grundy et al 1998c]. JComposer provides visual language to support modelling JViews-based software architectures, and generates JViews class implementations of these architectures. JViews-based environment components can be visualised at run-time, and reconfigured using a visual language similar to that of JComposer's, using the JVisualise tool [Grundy et al 1998c]. We have built several environments with JComposer, including the Serendipity-II process support environment and have reengineered JComposer using itself. While JComposer is used to statically design and generate JViews component specifications, Serendipity-II and JVisualise can be used to dynamically compose, reconfigure and integrate component-based systems.

In the following section we describe JViews and JComposer. We then discuss and illustrate how these tools support the specification and implementation of tool integration, collaborative work and user interaction facilities in environments like Serendipity-II. We then overview our recent work in identifying and codifying these and other "aspects" of component-based systems, and compare our work with other research in this area.

3. JViews and JComposer

We have been developing component-based software architectures for the construction of design environments for several years [Grundy et al, 1998b; Grundy et al, 1996]. Our latest architecture, implemented using the JavaBeans component-based API, is JViews [Grundy et al, 1998c]. Figure 3 shows the basic characteristics of JViews-based systems: components (rectangular icons), component relationships (oval icons), inter-component

links, and the propagation of events (which we call "change descriptions"), along links and relationships. These abstractions provide structural foundations for representing application data, interconnectivity and inter-component dependency and constraints. Additionally, event filter and action components provide reusable event-handling behaviour. Component interconnection can be both statically and dynamically specified.

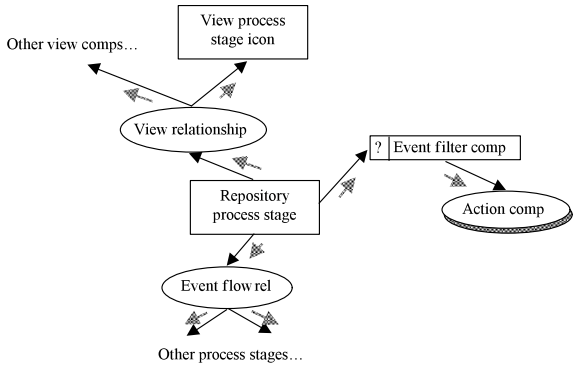


Figure 3. The JViews component-based software architecture.

JViews provides a richer range of event detection and propagation mechanisms than do other component-based approaches like JavaBeans, COM and CORBA. This includes the facility for components to listen for change descriptions generated both before and after component state has been modified, and to handle change descriptions sent to other components both before and after a third component's state has changed. Change descriptions can be stored, used to support undo/redo and versioning, and propagated to other users' environments to supporting collaborative editing [Grundy et al, 1996].

For example, in Figure 3 a repository process stage component is linked to: a view relationship, linked to view components representing some aspect of the process stage (e.g. graphical view icons and textual view descriptions); other process stage components, via an event flow relationship; and an event filter component. When the state of the process stage is modified, e.g. its name property changed, JViews change descriptions are sent to the linked components. Relationship components typically handle the event and update other linked components appropriately, while directly linked components respond to the event themselves e.g. updating their own state, redisplaying their icon/text, enforcing a constraint etc.

JViews is implemented as a set of framework classes in Java, extending the JavaBeans componentware API. Developing applications with the JViews framework classes directly is tedious, time-consuming and error-prone. We developed a CASE tool, using JViews, to assist developers to build component-based environments. Called JComposer [Grundy et al, 1998c], the tool provides visual languages for specifying components, relationship components, , and a range of interdependency links between these components. It also supports the definition of semantics by the use of a novel event handling language comprising event filter and action components. JComposer generates JViews class specialisations, implementing the specified environment as a set of JViews components. A set of reverse engineering tools allows developers to construct JComposer specifications from JViews classes, or to reverse engineer JViews specifications which "wrap" JavaBeans classes. JComposer specifications can be saved to persistent store and can be collaboratively edited using the same facilities as in Serendipity (See Section 5).

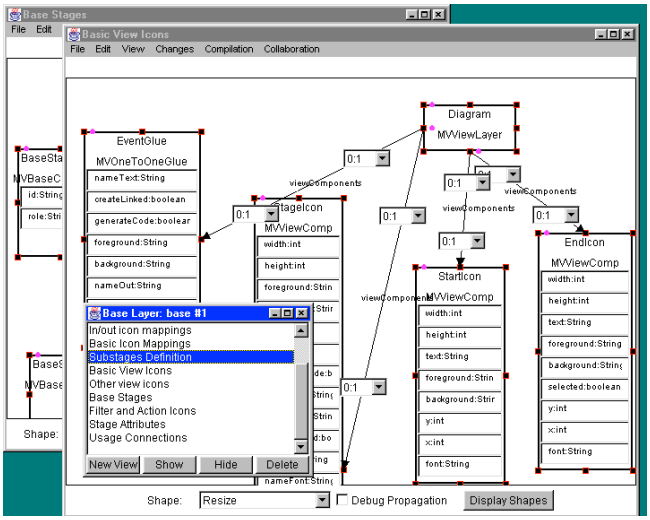


Figure 4. An example of JComposer being used to specify Serendipity-II.

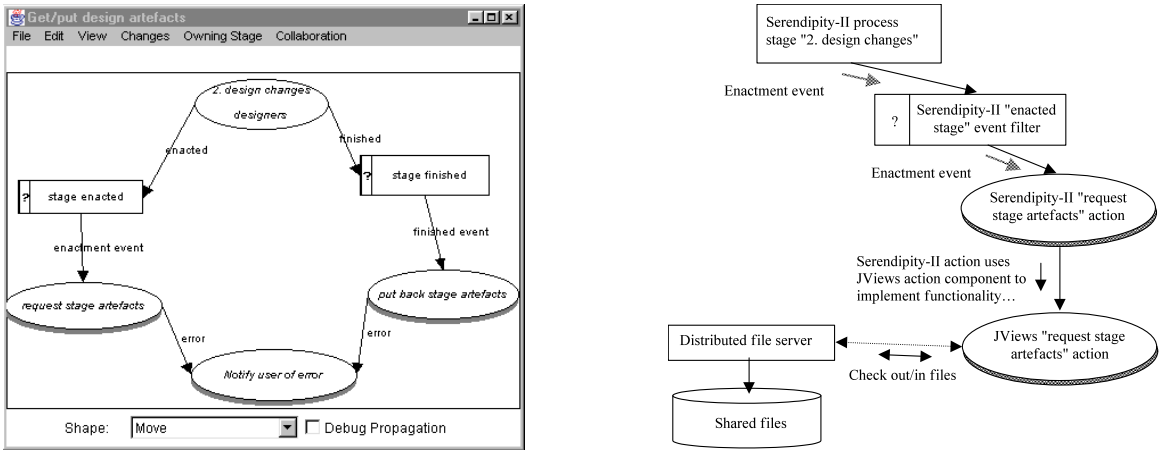
Figure 4 shows JComposer being used to specify Serendipity-II. A number of views of Serendipity are used, each providing specification of different aspects of Serendipity such as graphical specifications of components, filters and actions, or Java code implementing specialised JViews methods for detailed processing.

4. Tool Integration

In order to effectively use component-based environments like Serendipity-II, appropriate integration of third party tools and components must be supported. Serendipity-II 's filter/action event handling language, based on that of JComposer, allows users to plug in components representing third-party tools and link them to Serendipity artefacts and other reusable filters and actions via event propagation connections.

Figure 5 shows the specification of a simple software agent that automatically downloads and uploads files from a shared file server. When the Serendipity process stage "2. Design changes" is enacted (i.e. the user starts work on this stage), an event indicating this is generated and sent to all stages, filters and actions connected to this stage. The Serendipity filter and action model on the left specifies that when an enactment event is detected from this process stage, a "request stage artefacts" action should be run. This reusable component determines the files associated with the "2. Design changes" stage (specified in another Serendipity-II view) and downloads (checks out) these to the user's computer from the file sever. When the stage is completed, the files are automatically uploaded (checked in) back to the file server.

The illustration on the right in Figure 5 shows how components providing the tool integration between Serendipity-II and the shared file server work. When the user adds the "request stage artefacts" action, a JViews component (implemented with JComposer or directly in Java) is created which implements the Serendipity-II action's behaviour. The JViews component establishes a connection to the distributed file server. When the Serendipity-II stage is enacted, the enactment event propagated to the action by the filter instructs it to request all files associated with the process stage be downloaded.



(a) Specifying external tool usage in Serendipity-II.

(b) Intercommunication between tools.

Figure 5. Specifying the use of a third-party file server in Serendipity-II.

Another example of component-based tool integration is shown in Figure 6. In this example, a set of tools for planning travel, all independently developed, is being integrated. This tool set includes a reusable tree editor (used in this example to edit and browse a structured travel itinerary), a map visualisation, a chat tool, and a Web browser. The JComposer filter/action model show in Figure 7 specifies interconnection between these tools, represented as components. This results in the map visualisation being dynamically updated whenever the travel route specified in the itinerary editor is changed, and a chat message is written when the map visualisation (and hence itinerary travel route) is updated.

The chat tool, map and tree browser/editor are third-party Java applets which have been "wrapped" with JViews component interfaces, with the Java events generated by these tools being converted into JViews change descriptions. A JViews action was written which parses the textual representation of the tree editor for city names and updates the map city route. Another action sends a chat message via the chat tool with parameters for user and chat message. JComposer was then used to create instances of each of these tool "components" and actions, and additional filters which detect itinerary item updates and map route property updates respectively. Connecting these components, filters and actions in JComposer results in the integrated environment illustrated in Figure 6.

Using JComposer to facilitate tool integration has been quite successful in the domains we have used it for. This has included Serendipity-II integrated with CASE tools, programming environments, communication tools and office automation tools, and several tools characterised with JViews components and then integrated. We have been able to successfully and very tightly integrate Serendipity-II, JComposer, a UML modelling tool and an ER modelling tool [Grundy et al, 1998c]. Limitations of this approach are encountered when having to "wrap" third-party tools that do not already have well-developed component interfaces. This requires the development of JViews components to communicate with these tools, often in a limited way, and the translation of tool events into JViews change descriptions. This can be a complex process [Meyers, 1991; Dossick and Kaiser 1998; Valetto and Kaiser, 1995], and one which sometimes can only provide limited interface solutions. We hope that as component-based architectures become more wide-spread, third party tools will increasingly provide component-based interfaces we can leverage more effectively than many existing tools.

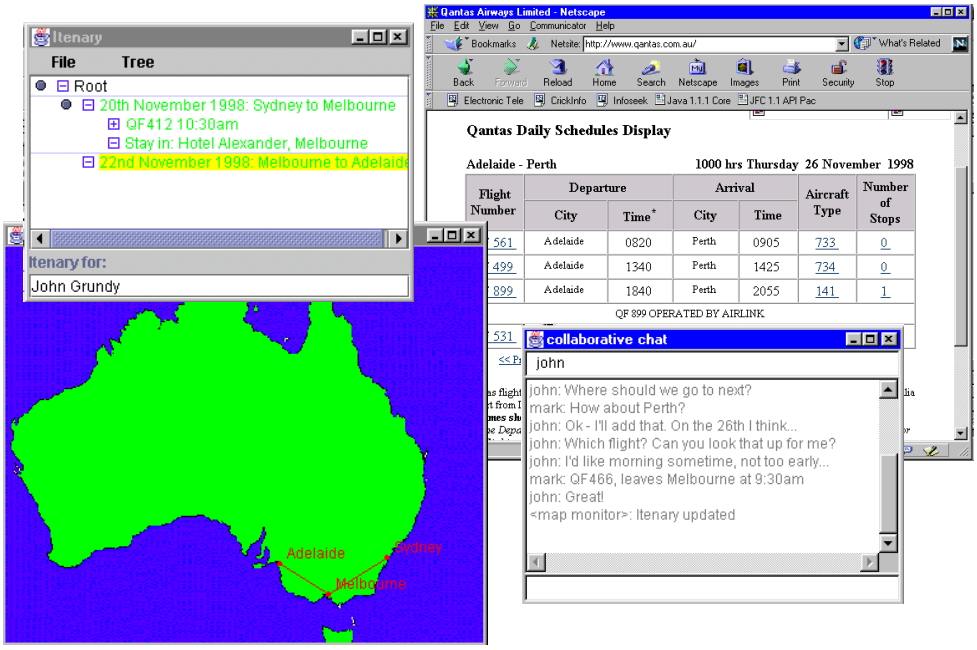


Figure 6. Integrating an itinerary editor, map visualisation and chat tool using Serendipity-II.

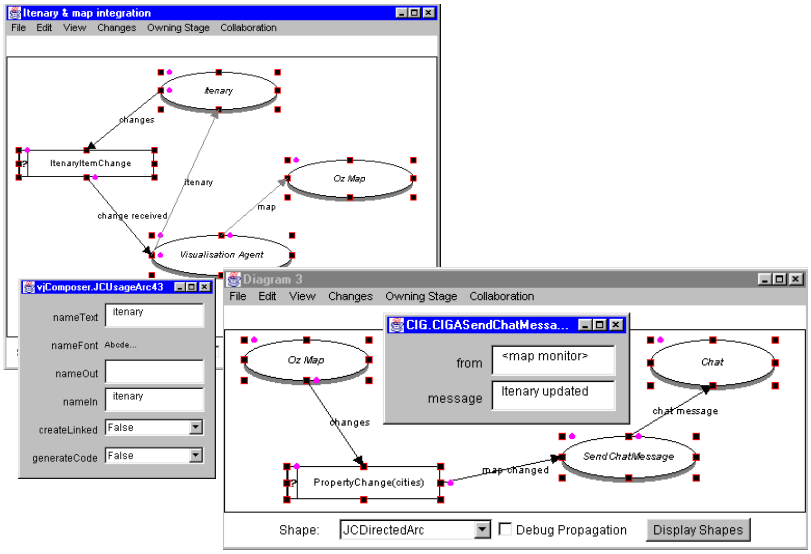


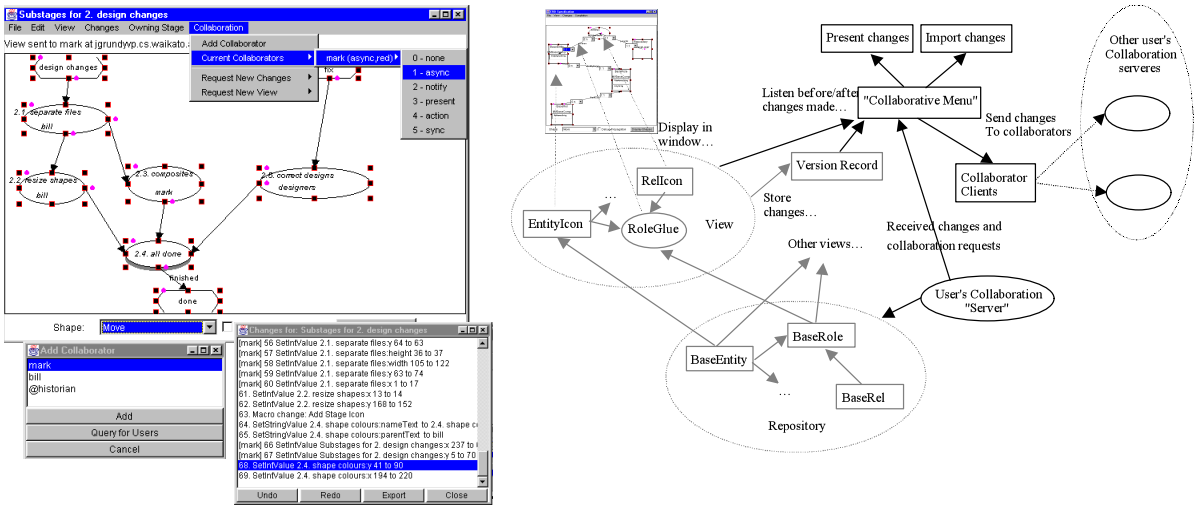
Figure 7. Serendipity-II filter/action models specifying tool integration mechanism via components.

5. Component Distribution and Collaborative Work Support

Supporting collaborative work has been an important aim in many of our component-based environments [Grundy et al, 1996; Grundy et al, 1998a]. Usually much effort is required to adequately architect an environment for collaborative editing, and to re-architect a single-user environment is very difficult and time-consuming. Our JViews-based environments were all originally single-user environments, and we used a component-based approach to seamlessly add flexible collaborative editing support, without necessitating changes to the environments or the components we used to facilitate this editing functionality.

Figure 8 shows a "collaboration" menu in use in Serendipity-II to configure the "level" of collaborative editing with a colleague (e.g. asynchronous, synchronous and "presentation" i.e. show editing changes to others as they occur but don't action them) [Grundy, 1998]. The "change history" dialogue on the bottom, right hand side shows a history of editing events for the user's process model view. Some changes were made by the user ("John") and others by a collaborator ("Mark").

The illustration on the right in Figure 8 shows how these collaborative editing components were added to Serendipity-II (and can in fact be added to any JViews-based environment, with no change to these components or the components that make up the environment). A "collaboration menu" component is created when the user specifies they want to have collaborative editing of a view. This listens to editing changes in the view, and records these. If the user is in presentation or synchronous editing modes with another user, the changes are propagated to that user's environment. There they are stored and presented in a dialogue (presentation) or actioned on the other user's view (synchronous editing). With asynchronous editing, a user requests a list of changes made to another user's view and selects from a dialogue which they wish to have applied to their view.



(a) Human-interface for collaborative editing.

(b) Software components.

Figure 8. Collaborative editing components in Serendipity-II.

JViews has abstractions supporting the replication of components (via object versioning), which is used to maintain copies of collaboratively edited views. Change descriptions generated in one view **are** propagated to another user's environment with component references translated appropriately. The change description propagation and listening support of JViews made it very easy to add collaborative editing components to existing JViews-based environments. It even allows fully synchronous editing, with locking, to be properly supported with no change to the original environment or the collaborative editing components [Grundy, 1998].

Figure 9 shows another example of distributed components in Serendipity-II. In this example, a distributed software agent is being specified using reusable filter and action components. When artefacts (in this example, Java source files) are modified while stage "4. Modify code" is enacted, events describing these chages are sent to all components linked to this stage. The two filters note the modification of Shape.java or EditorWindow.java, and the action forwards the change descriptions it receives to a "remote" stage, identified by remote user and name properties specified for the sender action. The receiver action in the right hand view forwards all changes it receives to a store action, which records the change description in an event history artefact. The left-hand view is run by Bill's Serendipity-II environment, while John's runs the right hand view. The nett result is that changes made by Bill to either of the two classes are sent to John for later examination via the user interface of the history artefact.

Implementing and deploying JViews-implemented collaborative editing components for JViews-based environments has been successful. In general, however, it is difficult to distribute components and provide appropriate collaborative editing functionality for them if they have not been designed with this in mind. Often component-based systems use simple subscribe-notify patterns that broadcast component update events only after the event has been actioned. It is often very difficult to provide fully synchronous editing for such components. Propagation of events and replication of component objects across multiple machines requires component registration and identification mechanisms that are also hard to retrofit to software components designed for single-user use.

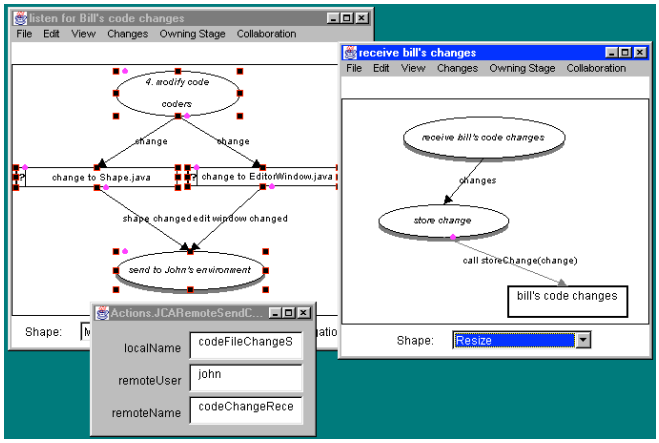
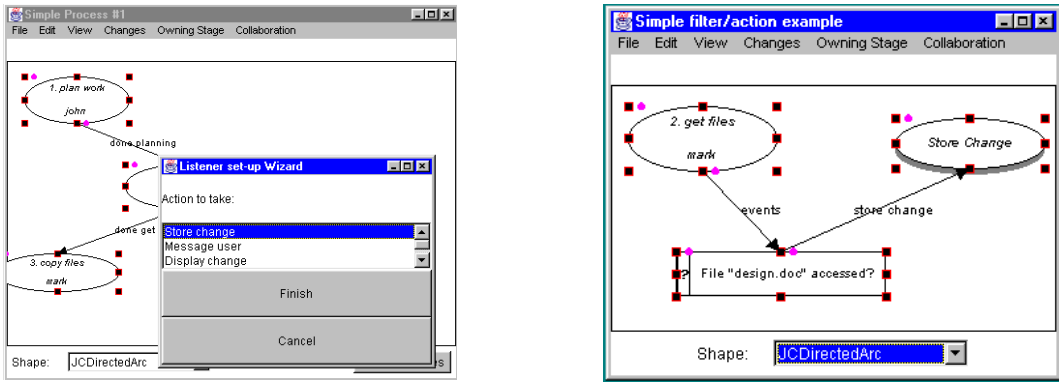


Figure 9. Specifying distributed software agents in Serendipity-II.

6. User Interaction and Configuration

Component-based software systems must provide consistent user interaction mechanisms across all components, and must allow users to modify the configuration of their application components in appropriate ways. A key implication of component-based software is the need to support extensible interaction mechanisms i.e. allow new components added to an environment to extend existing interaction menus, buttons, dialogues and windows.

We have developed a range of extensible interaction mechanisms and reusable components with human interfaces in our environments. For example, the change history dialogue (Figure 1) and enactment history dialogue (Figure 8 (a)) are both human interfaces of the JViews "version record" component, used to record change descriptions. The collaborative editing component illustrated in Figure 8 extends the menu bar provided by the view editing panel component of JViews-based systems. The buttons and menu items of JViews components can be extended by other components e.g. actions can add extra menu items to Serendipity-II process stage icon pop-up menus to allow users access to extra functions they provide.



(a) Part of a simple configuration "wizard" (b) Visual component configuration in Serendipity-II

Figure 10. End-user configuration in Serendipity-II.

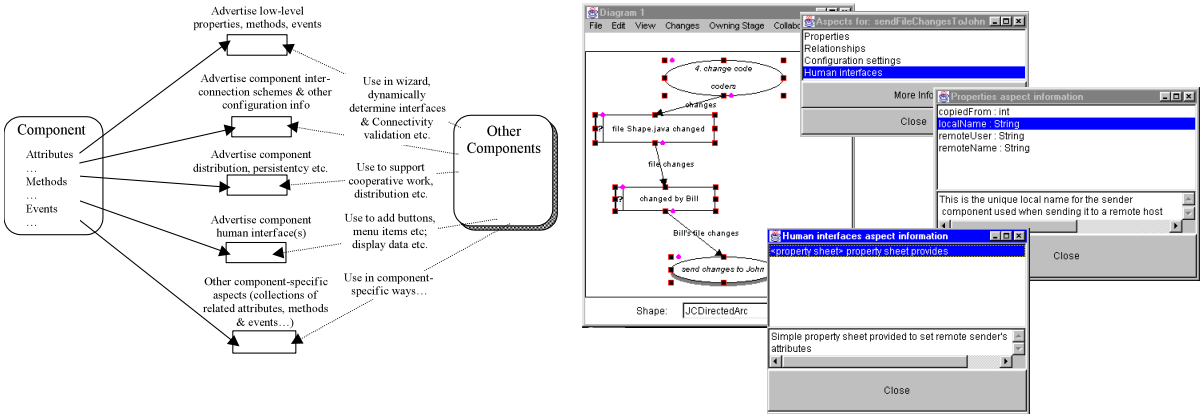
End-user configuration of components is supported in our environments by several mechanisms: "wizards" which guide users through component configuration, visual manipulation of component object representations, the use of plug-in actions, property sheets, and menus and buttons provided by components. Figure 10 illustrates a configuration wizard dialogue allowing a user to configure a simple change monitoring action and the visual configuration of an equivalent change monitor using filters and actions. Both of these techniques have been successful in our environments, with wizards useful for end-users with little knowledge of the component-based architectures behind the environment, and visual filter and action component composition useful for more sophisticated configuration by experienced users.

7. Advertising and Using Component Aspects

End-user interaction and configuration are two common "aspects" of component-based systems and inter-component interaction that need to be carefully designed, implemented and used, to ensure appropriate mechanisms are supported and are accessible by both end-users and other components. For example, the JViews version record dialogue should provide the ability to disable and/or hide its undo/redo buttons, which are sometimes not used when this component is reused, e.g. as a Serendipity-II enactment history. Similarly, JViews editing panel components should provide methods allowing (or disallowing) other components to extend its menu bar, as done by the collaborative editing component. Other aspects of component-based systems we have commonly dealt with include distribution and collaboration support, persistency, and inter-component linkage for tool integration. Each of these requires a component to advertise certain characteristics (such as unique identifier allocation, change locking and/or transactioning ability, and valid component types it can be linked to), in order for other components to effectively interact with this component.

We have been developing a design and implementation methodology, and appropriate support tools in JComposer and Serendipity-II, for representing and using such aspects of components. Figure 11 illustrates the publication of and subscription to such aspects in the JViews architecture, and how end-users can use these when configuring and using component-based systems. A component publicises information about aspects it supports and how these can be used e.g. a set of methods to allow extension/modification of its human interface aspects or for managing its persistency. Other components, which need to interact with this component, query it for these aspects and use them as appropriate. Aspects are complementary and orthogonal, in that they are used to characterise sets of attributes, methods and events of a component concerned with particular aspects of a component-based application. A component method, for example, can be used to support more than one aspect of the component's overall functionality. Another component can utilise one or more of the aspects publicised by a component. Aspects of a component can be statically described or changed dynamically at run-time, and third party components can have aspects described by proxy components.

Our implementation of component aspects in JViews uses a set of design patterns and associated AspectInfo classes and methods to provide an extensible set of characterisations of a component. JComposer and Serendipity-II allow component designers and users to access these aspects and check components are correctly linked and interacting by using aspect information. Automatic reconfiguration of components is also supported by the use of these aspect design patterns.



(a) Implementing aspects for JViews components (b) Using component aspects in Serendipity-II

Figure 11. Describing and using component aspects in JViews and Serendipity-II.

The example on the right in Figure 11 shows an end-user browsing the aspects of a "RemoteSendChange" action, used in this example to propagate change descriptions from one user's environment to another. This action component publicises four aspects (its parameters, links to other components, configuration constraints and human interface elements). The user can obtain extra information about these aspects as desired. Other components can also make use of these aspects, for example wizards, visual configuration tools and other components that want to reuse and/or extend the component. Aspects differ from low-level property, method and event descriptions JavaBeans publicise and type libraries COM components supply in that they typically describe various responsibilities of sets of component methods, properties and events. Our implementation of

AspectInfo classes in JViews also provide a range of methods allowing other components to use different kinds of aspects in "standard" ways, no matter what methods, properties and events are used to implement the aspect.

8. Discussion

Much recent work has been done in relation to component-based systems development. Various component-based software architectures have been developed, including JavaBeans [Javasoft, 1997], COM [Microsoft, 1998] and OpenDoc [Apple, 1995]. In addition, distributed object management architectures like OMG CORBA [OMG, 1998] offer capabilities for component modelling and distribution. Various tools have been developed for engineering systems with such architectures including JBuilder [Borland, 1997], Visual Age [IBM, 1997], Visual Javascript [Netscape, 1998], and those for specialised application domains, for example 3D modelling [Wagner et al, 1996]. These systems all provide fairly low-level support for component design and implementation. End-user configuration support is generally limited to simple component object linking. All component-based architectures provide mechanisms for publicising component properties, methods and events, but generally have no concept of higher level aspects encompassing collections of component features. We have found with our component-based tools and environments that characterising such aspects of components leads to more readily reusable components with better integration, collaboration and human interface solutions.

Tool integration approaches include federation [Bounab and Godart, 1995], enveloping [Valetto and Kaiser, 1995], middleware architectures [Dossick and Kaiser, 1998], database and file-based integration [Meyers, 91], and message passing [Reiss, 90]. While these techniques have proved successful in limited domains, none has managed to provide an ideal solution. Enveloping, middleware and message passing architectures are the most similar to component-based tool integration approaches that we have used, and these techniques can be usefully combined.

Workflow management systems and process-centred environments generally provide some support for task automation, tool integration and collaboration. Examples include ProcessWEAVER [Fernström, 1993], SPADE [Bandinelli et al,1996], Oz [Ben-shaull and Kaiser, 1994] and Regatta [Swenson et al, 1994]. These generally adopt low-level, macro or programming languages to support environment extension, or use a basic range of inflexible event-based triggers configured with Wizards. We have found the use of primarily visual notations to support environment extension to be the most useful, although Wizards and property sheets provided by components are also important. It is difficult to design components for end-user configuration in advance as designers often do not know exactly how and where their components are to be deployed.

Computer-supported cooperative work tools and environments allow groupware tools to be constructed which support distributed workers and artefacts. Examples include GroupKit [Roseman and Greenberg, 1996], SUIT [Dewan and Choudhary, 1991], COAST [Shuckman et al, 1996], and TeamRooms [Roseman and Greenberg, 1997]. TeamRooms and COAST adopt component-based approaches to implementing groupware functionality, but require systems to be designed with this functionality in mind from the outset. We have managed to re-architect JViews-based tools to support various forms of collaborative work without the need to re-implement existing components.

9. Conclusions

Component-based software architectures are becoming increasingly important as solutions for a wide range of software engineering problems. We have focused on the use of software components to support diverse tool integration mechanisms, collaborative work and object distribution, extensible human-interfaces, and end-user configuration of software applications. We have developed both software architectures which support component-based software development and tools for design and implementation with these architectures. Component-based solutions for tool integration, collaboration, human-interfaces and end-user configuration have proved appropriate and useful in the domains we have worked. However, further development of component-based architectures and design tools is necessary to improve their support for various aspects of software applications. The publicising of and subscribing to particular aspects supported or required by components is necessary to enable component-based solutions to be effectively used.

Currently we are developing JViews-based "wrapper" components for a variety of third-party components and tools, including standard JavaBeans, COM components and CORBA objects, and software development, office automation and database tools. The use of semi-automatic component interface and event generation tools in

JComposer will make this easier and repeatable. Mapping components supporting complex inter-component event and method mappings are being developed, allowing both tool integration and collaborative work with components to be better-supported. These mapping components will utilise JComposer and Serendipity-style visual languages, making them accessible to end-users and component developers. We are working on techniques for designing extensible human interface aspects for components, along with automatic wizard generation for end-user configuration of components. The publicising of and subscription to component aspects, and a development methodology for components incorportating standardised aspect patterns, is being done by extending JViews, JComposer and Serendipity-II, enabling more flexible combination of all aspects of software components.

References

Apple (1995): *OpenDoc™ Users Manual*, Apple Computer Inc.

Bandinelli et al (1996): Bandinelli, S. and DiNitto, E. and Fuggetta, A., Supporting cooperation in the SPADE-1 environment, *IEEE Transactions on Software Engineering 22 (12),* December 1996, 841-865.

Ben-Shaull and Kaiser (1994): Ben-Shaul, I.Z. and Kaiser, G.E., A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment, *Sixteenth International Conference on Software Engineering*, IEEE CS Press, 179-188.

Borland (1997): *Borland JBuilder™* , Borland Inc, http://www.borland.com/jbuilder/.

Bounab and Godart (1995): Bounab, M. and Godart, C., A federated approach to tool integration*, Proceedings of CaiSE*95*, Finland, June 1995, LNCS 932, Springer-Verlag, pp. 269-282.

Dewan and Choudhary (1991); Dewan, P. and Choudhary, R. Flexible user interface coupling in collaborative systems, in *Proceedings of ACM CHI'91*, ACM Press, April 1991, pp. 41-49.

Dossick and Kaiser (1998): Dossick, S. and Kaiser, G.E., Middleware architectures for workgroups, *Proceedings of IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Stanford University, June 17-19, 1998, IEEE CS Press.

Fernstrom (1993): Ferstrom, C. ProcessWEAVER: Adding process support to UNIX*, 2nd International Conference on the Software Process*, Berlin, February 1993, IEEE CS Press, 12-26.

Grundy (1998): Grundy, J.C. Engineering Component-based, User-configurable Collaborative Editing Systems*, Proceedings of 7th Conference on Engineering for Human-Computer Interaction*, Crete, September 14-18, 1998, Kluwer Academic Publishers.

Grundy and Hosking (1998): Grundy, J.C. and Hosking, J.G., Serendipity: an integrated environment for process modelling, enactment and work coordination, *Automated Software Engineering 5 (1),* Kluwer Academic Publishers, January 1998, 27-60.

Grundy et al (1996): Grundy, J.C., Hosking, J.G. and Mugridge, W.B., Supporting flexible consistency management with discrete change description propagation, *Software - Practice & Experience 20 (9),* September 1996, 1053-1083.

Grundy et al (1998a): Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D., A decentralised architecture for process modelling, to appear in *IEEE Internet Computing*, September-October, 1998, IEEE CS Press.

Grundy et al (1998b): Grundy, J.C., Hosking, J.G., and Mugridge, W.B., Inconsistency management in multiple-view software engineering environments, to appear *in IEEE Transactions on Software Engineering*, IEEE CS Press.

Grundy et al (1998c): Grundy, J.C., Mugridge, W.B., and Hosking, J.G., Visual specification of multi-view visual environments, *Proceedings of 1998 IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, September 2-4 1998, IEEE CS Press.

IBM (1997): *VisualAge™ for Java*, IBM, http://www.software.ibm.com/ad/vajava/.

Javasoft (1997): *The Java Beans™ 1.0 API Specification*, Sun Microsystems Inc., http://www.javasoft.com/beans/.

Linton et al (1989): Linton, M. and Vlissides, J.M. and Calder, P.R., Composing User Interfaces with InterViews, COMPUTER 22 (2), February 1989, 8-22.

Microsoft (1998): Component Object Model™ , Microsoft Corporation, http://www.microsoft.com/com/.

Meyers (1991): Meyers, S. Difficulties in Integrating Multiview Editing Environments, *IEEE Software 8 (1),* January 1991, 49-57.

Myers et al (1997): Myers, B.A. et al, The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE Transactions on Software Engineering 23 (6)*, June 1997, 347-365.

Netscape (1998): *Visual Javascript™ ,* Netscape Communications Inc, http://www.netscape.com/compprod/products/tools/visual_js.html.

OMG (1998): *OMG CORBA*, Object Management Group, http://www.omg.org/.

Reiss (1990): Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment, *IEEE Software 7 (7),* July 1990, 57-65.

Roseman and Greenberg (1996): Roseman, M. and Greenberg, S., Building Real Time Groupware with GroupKit, A Groupware Toolkit, ACM *Transactions on Computer-Human Interaction 3 (1),* March 1997, 1-37.

Roseman and Greenberg (1997): Roseman, M. and Greenberg, S., Simplifying Component Development in an Integrated Groupware Environment, *Proceedings of the ACM UIST'97 Conference*, ACM Press, 1997.

Shuckman et al (1996): Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. (1996) 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.

Swenson et al (1994): Swenson, K.D. and Maxwell, R.J. and Matsumoto, T. and Saghari, B. and Irwin, K., A Business Process Environment Supporting Collaborative Planning, *Journal of Collaborative Computing 1 (1),* 1994.

Valetto and Kaiser (1996): Valetto, G. and Kaiser, G.E., Enveloping Sophisticated Tools into Process-centred Environments, *Automated Software Engineering 3,* 309-345.

Wagner et al (1996): Wagner, B., Sluijmers, I., Eichelberg, D., and Ackerman, P., Black-box Reuse within Frameworks Based on Visual Programming, in *Proeedings of the. 1st Component Users Conference,* Munich, July 14-18 1996, SIGS Books.