Preprint of paper published in Interacting with Computers, vol. 14, no. 3, © Elsevier Science Publishers.

Developing Adaptable User Interfaces for Component-based Systems

John Grundy[1, 2] and John Hosking[1]

Department of Computer Science[1] and Department of Electrical and Electronic Engineering[1, 2],
University of Auckland, Private Bag 92019, Auckland, New Zealand
{john-g, john}@cs.auckland.ac.nz

Abstract

Software components are becoming increasingly popular design and implementation technologies that can be plugged and played to provide user-enhanceable software. However, developing software components with user interfaces that can be adapted to diverse reuse situations is challenging. Examples of such adaptations include extending, composing and reconfiguring multiple component user interfaces, and adapting component user interfaces to particular user preferences, roles and subtasks. We describe our recent work in facilitating such adaptation via the concept of user interface aspects*, which support effective component user interface design and realisation using an extended, component-based software architecture.*

Keywords: component-based user interfaces, user interface adaptation, software architecture

1. Introduction

Component-based software applications are composed from diverse software components (software "building blocks") to form an application [1, 19, 24, 25]. Developers and sometimes end-users compose ("assemble") applications from often stand-alone components in flexible ways to achieve a desired set of functionality. Two key aims of component technologies are to increase reuse of software in diverse situations without code modifications, and to enable end users to extend and reconfigure their applications via plug-and-play of components. Typically many of the components used to build and/or extend an application have been developed separately, with no knowledge of the user interfaces of other components they may be composed with. This can result in component-based applications with inappropriate, inconsistent interfaces.

For example, two components with user interfaces that need to be accessed simultaneously may be hard-coded to each open a separate window. Composed components may also provide inconsistent interface metaphors, e.g. menu items vs buttons. They may show unsuitable interfaces or parts of interfaces to a user, due to the user's level of expertise, the task and/or role being performed, and users' personal preferences. As end users reconfigure their applications, they may add new components with user interfaces that introduce further complications or inconsistencies to the overall application interface.

There is thus a need for software components to provide more adaptable user interfaces than most do at present. Unfortunately the design and implementation of many existing software components, and the architectures they are built upon, do not adequately support the description of component user interfaces and adaptation of them. Mechanisms are needed to allow components to: inspect and understand other component user interface elements; programmatically adapt related component interfaces to suit a particular reuse situation; and be able to extend and combine the interfaces of other components with their own.

We describe our approach to addressing these issues. This uses *component user interface aspects* to describe user interface elements and adaptability. These aspects are characterised by component developers and are encoded in component implementations. Other components can use them to determine the user interface elements of a component, and standardised programming interfaces are used to extend, compose and reconfigure component interfaces in various ways. We use the state of a workflow engine to support adaptation of interfaces to particular user roles and subtasks.

Section 2 illustrates the need for component user interface adaptation using a component-based, Collaborative Information System, and Section 3 reviews related research. Section 4 briefly describes our concept of user interface aspects and the expression of such aspects in a software architecture used to support component implementation. Sections 5 to 7 illustrate particular kinds of user interface adaptation our approach supports, and briefly discusses realisation of these techniques using our architecture. Section 8 summarises the contributions of this research and outlines some future work directions.

2. Need for User Interface Adaptation

Our need to develop improved approaches to component user interface adaptation arose from experiences developing several component-based environments [8, 9, 10].

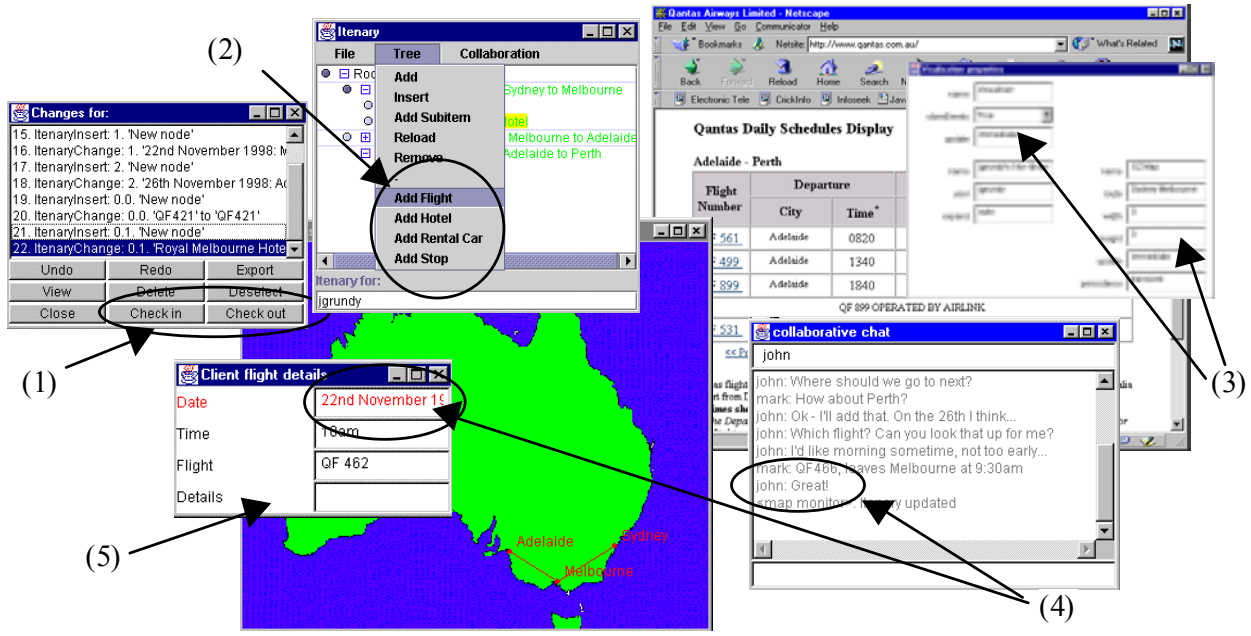


Figure 1. Some Itinerary Planner component user interfaces.

A screen dump from one such system, a collaborative travel itinerary planner, is shown in Figure 1. This system provides multiple views to a travel agent and client allowing them to co-operatively plan a trip. Views include structured itinerary, map visualisation, itinerary item details, web-based information, and a visual plug-and-play component configuration interface.

A variety of software components have been composed to produce this system, many reused from other applications. However, in order to provide end users with appropriate user interfaces, a number of individual component user interfaces had to be adapted in various ways. Through our work with a variety of component-based systems we have identified four main kinds of component interface adaptation:

- *Extension*. This is where one component allows one or more of its user interface elements to be extended in a controlled, consistent fashion by other components, to support a single, consistent interface for all. For example, in Figure 1 a component storing editing events has its button panel extended by a version control tool, allowing sets of itinerary changes to be tracked over time. This seamlessly provides users access to the version control tool's facilities (1). In a similar way, the itinerary editor's menu is extended by each kind of itinerary item component (2), providing users a quick access mechanism to item creation. Extension avoids the problem of multiple, inconsistent access points to common functionality.
- *Composition*. Combining elements of one component's interface with those of one (or more) others may be more suitable for users than to have each presented separately. For example, property sheets from multiple components, such as the map and map visualisation agent, can be combined (3), as the properties for these components are almost always changed at the same time by users. A composed interface avoids using multiple windows and disjoint access points to the two components' properties.
- *Reconfiguration*. Other software components may need to reconfigure a component's interface. For example, a collaborative work software agent has adapted a component's user interface to suit its group awareness needs by highlighting parts of the interface other users are interacting with (4). This reconfiguration gives the user concrete feedback about the multiple user nature of the system through existing component interface elements, rather than adding new elements for this purpose.
- *Adaptation to user, role and subtask.* Users may specify preferences about which elements or alternative interfaces they want to use, default values and constraints, and what adaptation approaches are preferred. Some component user interfaces and/or elements suit some users but not others, based on the particular user's role or subtask being performed. For example, itinerary item dialogues need some items hidden e.g. the fare class, if a customer is the user of the interface, rather than a travel agent (5).

Other forms of adaptation we are exploring include seamlessly adapting the itinerary component interfaces to web-based and PDA devices, including the differing user interaction styles used as well as differing colour, resolution and base user interface element facilities available. To support the development of software components that are amenable to all of these kinds of user interface adaptation, new approaches to specification, design and implementation of adaptable interfaces are needed.

3. Related Research

Common adaptive user interface techniques used by software developers include extensible menus and panels and programmatical reconfiguration of interfaces [16, 19, 7]. However, no commonly agreed design guidelines exist for building systems with adaptable user interfaces. Just as significantly, no commonly agreed software architectures and implementation techniques exist to allow developers to build adaptable components.

User interface frameworks, such as Interviews [13], AWT and JFC [6] and Amulet [17], permit composition of interfaces from discrete objects representing user interface elements, and most allow interfaces to be dynamically built and changed at run-time. However, there is typically little guidance or control over how other objects go about discovering, understanding and adapting interfaces built with these frameworks [4]. Thus systems built using these frameworks use ad-hoc approaches to adaptation which may well be incompatible with other's approaches, greatly reducing the reusability of software components with adaptable interfaces.

Component-based software architectures for building user interfaces, such as JavaBeans [19], Active-X [3] and OpenDoc [1], provide more powerful component introspection mechanisms that allow other components to discover properties, methods and events dynamically. However, neither these introspection mechanisms nor the design methods and coding standards for such systems address the need for user interface adaptation in any general, high-level way. Basic design guidelines that do exist [25, 19, 3] suggest components should support adaptation of the user interfaces, and architectures should allow this, but no consistent approaches are used nor appropriate implementation support exists. There has been work at attempting to define communications architectures to support system inter-operability and adaptability, such as Jini [18]. Our user interface adaptation work can be seen as a similar thrust but focusing on interface component adaptation rather than service look-up and adaptation.

Work on adaptable user interface systems [12, 5, 27], intelligent user interfaces [20, 21], and agent-based systems [15, 23] has contributed to the development of techniques supporting various kinds of interface adaptation. Some adaptable and agent-based systems support techniques for designing and implementing user interface adaptation facilities. However, such approaches use custom architectures and implementations that assume all other components are designed and built in the same way. A more major limitation is the number of assumptions made about the kinds of user interface techniques to be supported. These are typically limited to extensible menus, message areas and command lines. While agent-based systems use knowledge encoding techniques extensively in order to exchange information, they don't use these to exchange information about their user interfaces to support adaptation.

Many end user computing systems [14, 16] support user interface-level configuration by end users. This often necessitates adapting "component" interfaces to incorporate user preferences as well as integrate added component interfaces. Again, most end user computing systems adopt either ad-hoc solutions, incompatible with each other, or restrict adaptations to simple menu or tool bar extensions [16].

Extensible workflow systems [9], process-centred environments [2], and collaborative work tools [22, 26] have long recognised the need for integrating and modifying user interface elements. Most characterise the adaptable parts of interfaces at very low levels of abstraction however, and do not agree on a consistent approach to implementing such adaptability. Many make unreasonable assumptions about the adaptation and software interfaces provided by related tools and components, greatly reducing their flexibility.

4. A Supporting Architecture

4.1. User Interface Knowledge Representation

Due to the limitations of current approaches, we have been developing a technique for characterising component user interfaces at a high-level of abstraction. Support for describing and inspecting these characteristics forms the basis for implementing adaptation facilities in a component-based software architecture. This work has been part of the development of a new component engineering methodology called aspect-oriented component engineering [11]. This approach uses systemic *aspects* to describe the way in which components provide services or require services from other components. In addition to using this approach as a methodlogy, we have also added architecture support for it to a component development framework. This allows component developers to identify, describe, reason about and implement generic persistency, distribution, collaborative work and end user configuration support for component-based systems.

User interface information for components may also be characterised using aspects These describe the user interface-related services a component provides to and requires from other components. Examples of user interface aspects include dialogues and windows a component provides (or requires from another component for extension), panels (composite user interface elements) provided or required, and menus, buttons and other basic interface elements provided or required. Information recorded about these aspects may include the nature of the interface element provided or required, related elements and/or interfaces for the component, how an element may be adapted and/or preferred adaptation approaches, information about the component's software interfaces which enable adaptation of elements, and information about particular users, roles and subtasks for which elements are relevant.

Consider a very common example of menu bar extension: in this case itinerary item components extending the itinerary editor component's menu bar, as

illustrated in Figure 1. This is achieved by having the itinerary editor designer characterise the menu bar as being an extensible user interface affordance the editor provides for other components. The itinerary item designer characterises the user interface needs of these components as requiring a component that provides an affordance (of some kind) they can extend. Constraints may be specified about both the extension provisions and requirements of each of these components: the editor may limit extension of its menu bar to adding menus to the end or only adding menu items to existing menus in the bar. Similarly, the itinerary item component may require specifically an extensible menu bar, or may generalise this to some extensible menu (pull down or pop-up), or even any extensible affordance (which may be a menu, button panel, list or combo box or whatever).

Figure 2 illustrates the publication of user interface aspect information by an itinerary item factory component, responsible for creating itinerary items of a particular kind, and the itinerary editor component, responsible for viewing and editing itinerary items grouped in a tree hierarchy. Aspect details with a "+" in front are provided by the component, "-" are required. For example, the itinerary item factory component requires an extensible affordance it can add a user interface element to. This is restricted to being a menu (pull-down or pop-up) by its KIND property. The itinerary editor provides an extensible pull-down menu bar which can be extended by the addition of menus or menu items (specified by its EXTENDS_BY property).

Another, more complex example is where a composed user interface is being used. A component providing UI composition (typically inside a window or dialogue frame) obtains information about the UIs of other components to compose by querying each component's aspects. It determines the mechanism to obtain each components UI elements (typically grouped as a panel) then builds and displays a composite UI for the group as a whole.

Note that conflicts can arise during user interface adaptation. For example, two components may need to extend a third component's menu, but both want to use the save menu item label. One component may want to disable a control while another highlight it. One component may want to hide a component's panel from the user due to their role or current task, and another compose it and display it with other panels. This is similar to the conflicts that can arise in programming languages using multiple inheritance (e.g. C++) where same-named functions are inherited and must be renamed.

We currently resolve such conflicts using basic mechanisms. The default situation is that the last component to make an adaptation to a UI typically "wins" in that its adaptation over-rides others. However, developers can build components to e.g. annotate same-named menu items and buttons or reposition/separate them; check relative "priorities" of adaptations and apply only the most important ones; or allow end users to modify display and adaptation preferences to solve conflicts in the way that most suits the user's needs.

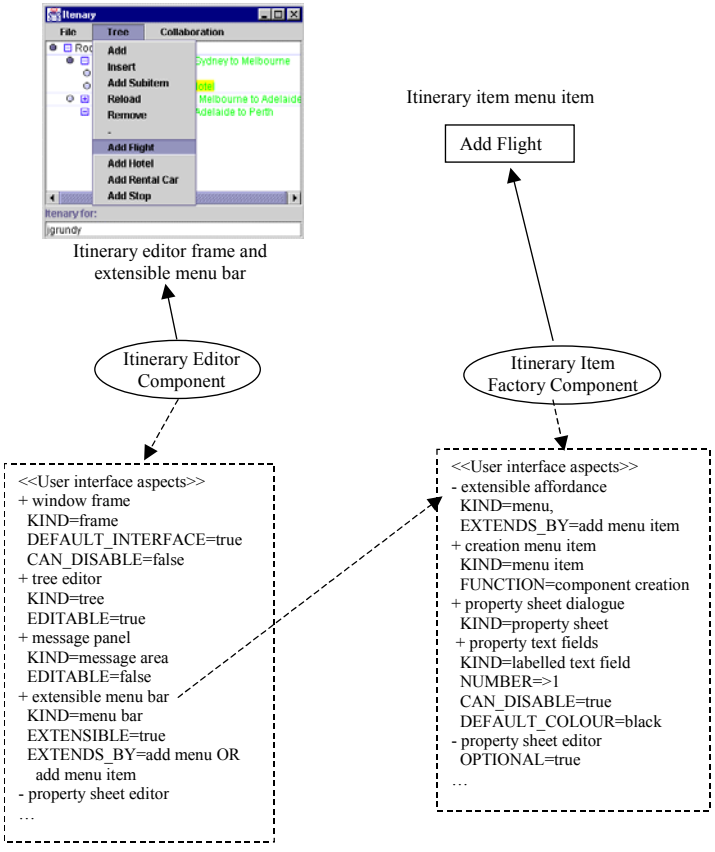


Figure 2. Aspect publication by software components.

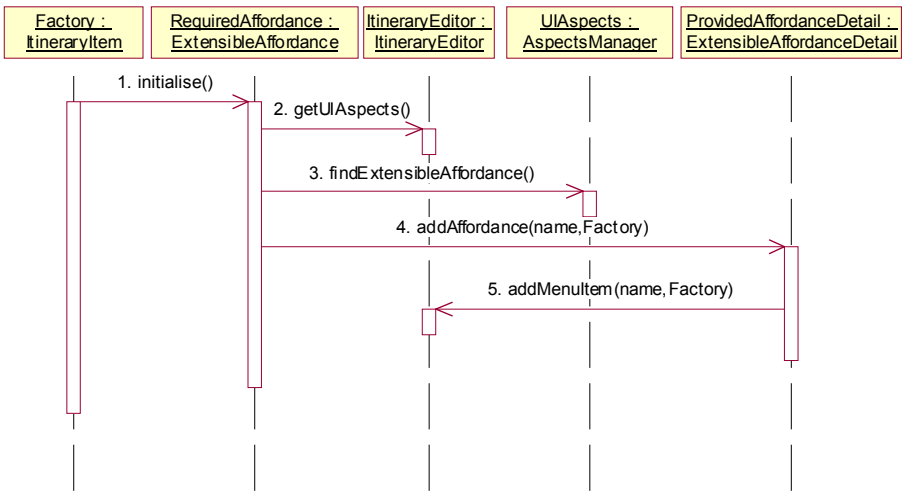


Figure 3. Extensible affordance user interface aspects.

4.2. Architecture and Component Framework Support

We have extended our Java-based JViews [8] software architecture and implementation framework to allow components (implemented as components called "JavaBeans") to advertise such user interface (and other) aspects. Component developers specify a component's user interface aspects during design, using our JComposer CASE tool [11]. This information is encoded in component implementations. JViews includes several classes that encode this information, and which provide standardised APIs for adapting other component's user interfaces. Aspect encodings can be modified dynamically, for example to change default and preferred values and to specify additional information, such as role and subtask. Together, these mechanisms allow independently-developed components to exchange knowledge about their user intrerfaces in a commonly-agreed manner, and support programmatic interface adaptation using commonly-agreed mechanisms.

Figure 3 illustrates the way JViews components achieve user interface adaptation in a de-coupled manner using aspects. After the factory and editor components have been linked, the factory tells its required extensible affordance aspect detail to initialise (1). This aspect detail object obtains a provided extensible affordance aspect detail object from the itinerary editor (2, 3) and asks this to extend the editor's menu (4). The editor's extensible affordance detail knows how to appropriately extend the editor's user interface, and adds a new menu item (5), which, when selected, informs the factory of this user interaction. Note the factory and editor have no knowledge of each other's interfaces - all extension is via their (standardised) aspect detail objects. Various constraints can be added in aspect detail objects to e.g. check requested adaptation is sensible, appropriately control user interface layout and interaction, modify permitted adaptations at run-time and so on.

4.3. System-wide Aspects

Figure 4 shows a larger example consisting of JViews components from our collaborative travel itinerary planner together with some of the user interface aspects of these components. Note that user interface aspect details may overlap, such as a panel aspect detail which describes an aggregate of several textfield and button aspect details. Not all user interface elements relating to a component need have an aspect detail characterisation, for example if the component designer wants them always treated as a composite element or to not be adaptable.

We use the Serendipity-II workflow system [10] to provide information about users, their roles in a task specification and the particular subtask they are performing. User preferences about interface adaptation are associated with role information and a task adaptor component monitors the workflow engine state. Several of the components in the travel itinerary planner, such as the tree-based itinerary editor, editing history, version control tool, map visualisation and collaborative awareness components have been reused from other applications [9].

The following three sections illustrate some examples of adaptation of components using extension and composition of related component interfaces, reconfiguration of a component's interface by other components, and adaptation based on user preferences and subtask.

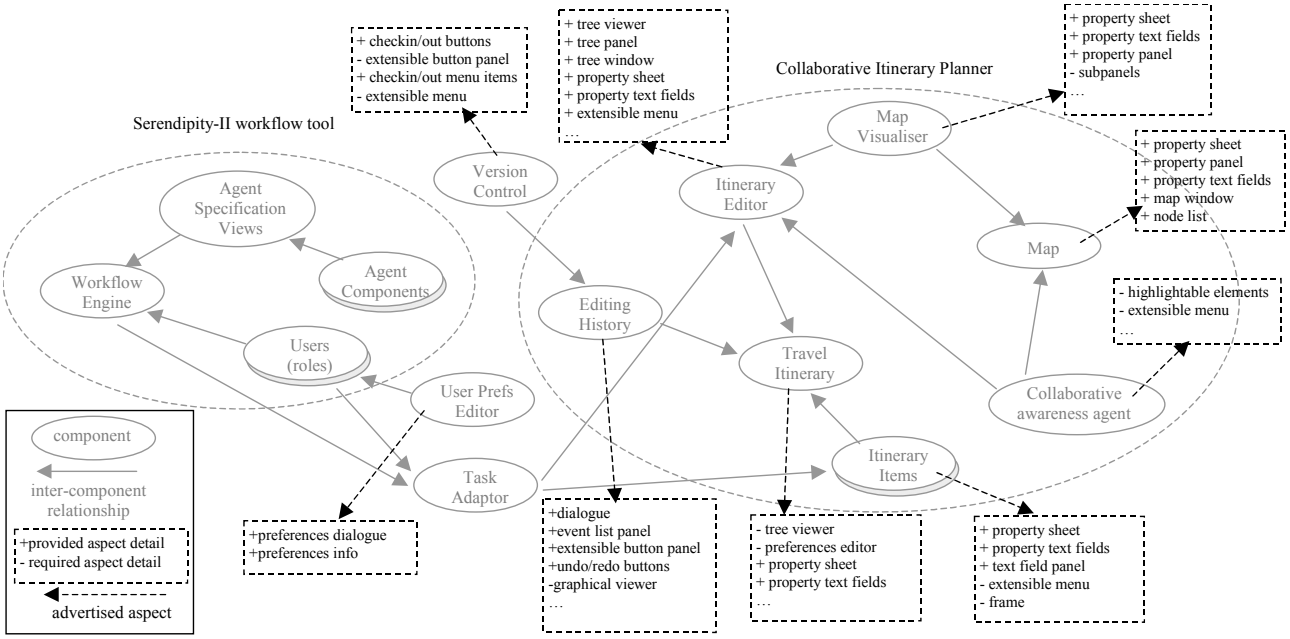
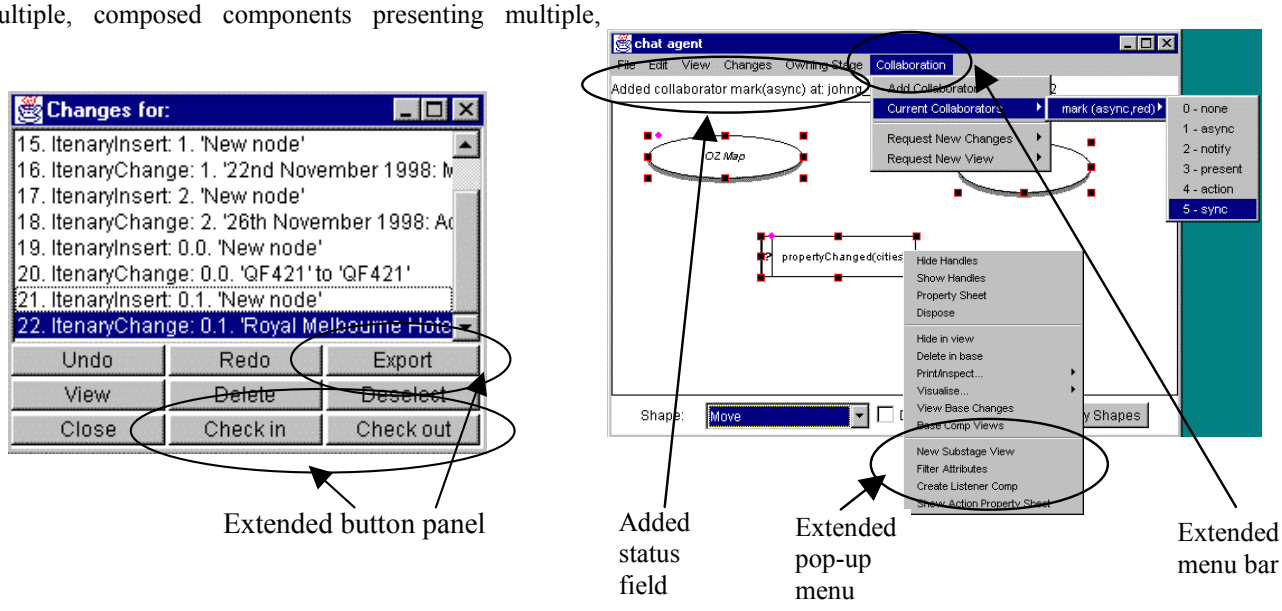


Figure 4. Collaborative travel itinerary planner architecture and some component user interface aspects.

5. Interface Extension and Composition

The most common example of user interface adaptation we have encountered is the need for components to seamlessly share user interfaces. Often this is by one component providing an affordance (e.g. button panel, pop-up or pull-down menu, combo box or text field panel) that other components can extend. The extending components thus present access to their own data and functionality via another component's interface in a seamless fashion.

User interface extension avoids the situation of multiple, composed components presenting multiple, inconsistent interfaces to end users. This often happens in naïve composition of component-based systems where software components are composed with no knowledge of each other's user interfaces and inappropriately open windows, display their user interface elements in the "wrong" place, or use inconsistent appearance and behaviour, thus confusing users. The itinerary editor provides one example of how to avoid this by having all component factories extend its menu bar to provide a consistent, controlled interface to their functionality, no matter where a particular factory is sourced from.



(a) Extensible panel.

(b) Extensible menus and panel.

Figure 5. Examples of user interface extension.

For another example, consider the event history component's dialogue, shown in Figure 5 (a). A file persistency component and version control tool component are to be used with this event history to manage export/import of event object data and versioning of event object lists respectively. These components, instead of providing or using their own user interfaces, have extended the event history's button panel to give the user access to their functionality. Clicking on these buttons will then open file save and version check in/out dialogues as appropriate. Figure 5 (b) shows two more examples of menu extension for a software agent specification view component. A collaborative editing component has extended the view component's menu bar to provide the user with a hierarchical menu. A newly created software agent, represented by the rectangle icon, has extended the icon's pop-up menu to provide the user access to its functionality. A component's interface can be extended by adding discrete elements e.g. buttons, text fields, combo boxes, radio and check boxes and text areas. For example, the collaborative editing component has added a status message field underneath the view's menu bar.

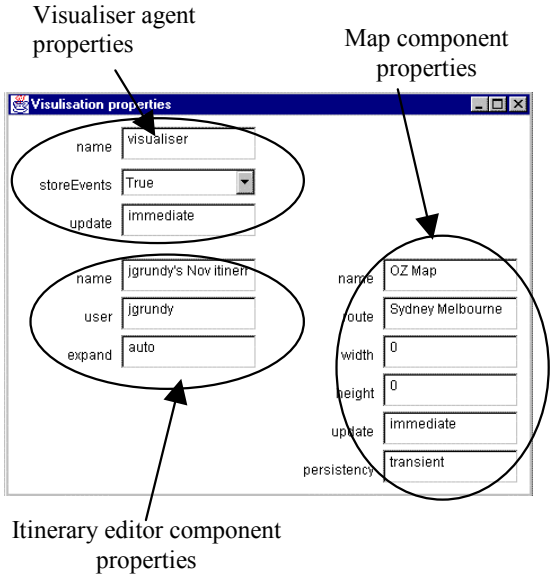


Figure 6. Example of User Interface composition.

A related technique for supporting user interface adaptation is composition of multiple component interfaces. Figure 6 illustrates composition of itinerary editor, visualisation agent and map component property sheet panels. This allows end users to access and/or modify these three component's properties at the same time, rather than have three dialogues. Such composition can also be done at the individual user interface element level, with components inter-mixed in the composite dialogue rather than remaining in separate panels.

Care needs to be taken when designing user interfaces that may be extended and composed, and when designing components that extend or compose other components' interfaces. Designers need to be aware that extending part of an interface will possibly change the appearance, size and layout of the interface. If inappropriate extension or no re-layout of elements is done, undesirable layouts can result. Ordering of a component's user interface elements might be important and should be preserved. For example, extending the menu bar of an application should constrain new menus to be to the right of previously added menus, so the File and Edit menus are always kept at the left. Similarly, it may make sense for a component that is extending another component's user interface to add its affordances in places which relate to the affordances already there, e.g. adding the Check in and Check out buttons BEFORE the Close button (unlike the event history dialogue above!). Label and icon conflicts between composed interfaces can be resolved by annotation or layout changes. Developers need to specify ways in which user interface elements can sensibly be embedded with user interface elements from other components. We have found using panels containing multiple elements gives reasonable control on how these groups can be composed. In addition, care must be taken with constraints, tab ordering and field inter-dependencies, so that behavioural constraints are sensibly preserved when parts of a component user interface are composed.

While designing and implementing user interfaces that support extension and composition takes more care and effort, the reuse costs for these components drop dramatically as new interfaces do not need to be developed nearly as frequently as for components with interfaces that are not adaptable. In addition, we have found that having components that share interfaces dynamically greatly enhances usability of applications.

We achieve most user interface extension and composition for components in the way outlined in the previous section for the editor's menu bar. Composition is more complex than extension, in general, with composite aspect detail components having to obtain a list of interface elements to compose from related components and apply a composition algorithm. Figure 7 shows an example. When the map component (or a related component's) property sheet is requested (1), the map component's composite interface aspect detail object is informed (2). This obtains composite aspect detail objects indicating required composition from each of the components related to the map (3-5). The provided composite interface aspect detail object then combines the interfaces of those requiring composition into one property sheet dialogue and displays this to the user (6). The composition process constrains composed elements to use the same look-and-feel in limited ways.

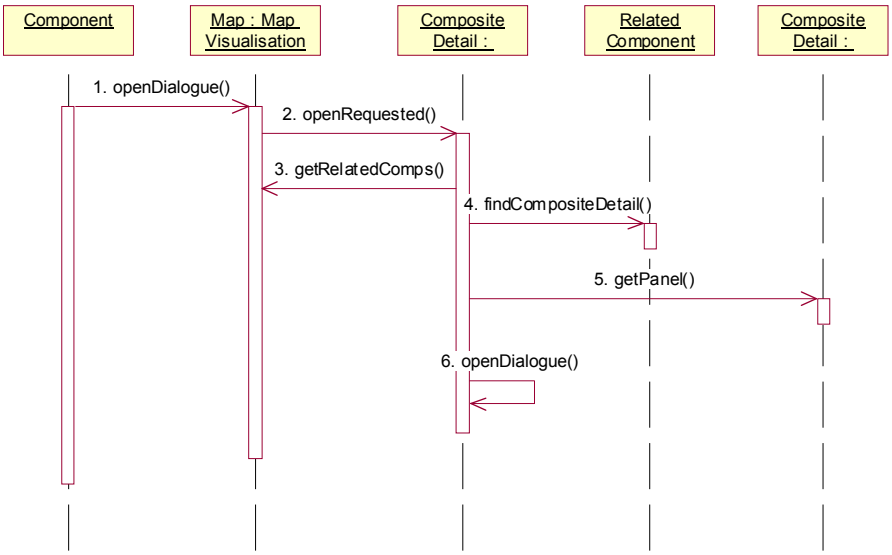


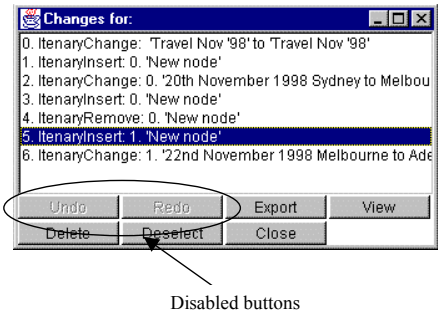
Figure 7. Example of interface composition process.

6. User Interface Reconfiguration

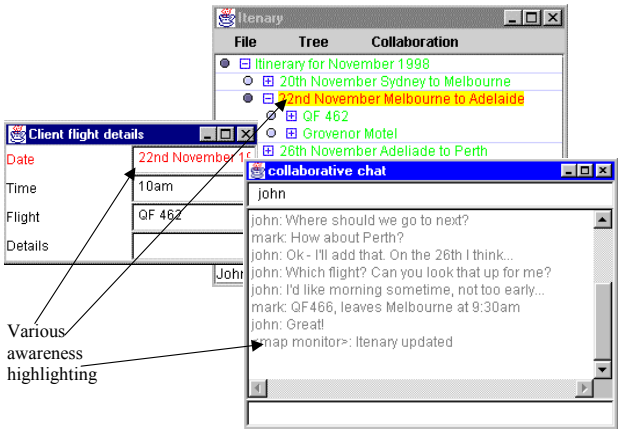
Components often need to reconfigure the existing user interface elements of other components, including hiding, showing, disabling or enabling user interface elements, or changing display and/or behavioural characteristics, such as colour, default values, and layout and editing constraints. In this way a component may make use of the user interface elements provided by a component in ways not anticipated by the original developer, in order to provide appropriate interface characteristics for a component in a new situation it is reused.

Figure 8 illustrates examples of such reconfiguration from our collaborative itinerary planner. In Figure 8 (a) the undo/redo buttons of the event history component have been disabled by another component. This might occur when undoing or redoing the stored events doesn't make sense, for example where the itinerary represents a past trip and can't be changed. Figure 8 (b) illustrates adaptation of itinerary editor and itinerary item component user interfaces by a collaborative work awareness agent component. This agent highlights parts of the itinerary another user is modifying in various ways by adjusting the display and editing characteristics of parts of their user interfaces. For example, the item being edited in the itinerary tree editor is highlighted and is not able to be changed (i.e. is "locked"). The field being edited in an item property sheet dialogue is highlighted in a different way. The agent also appropriates the collaborative chat messaging tool for notification support, sending "map monitor" messages.

Reconfiguring user interface elements may adversely affect usability. When designing user interfaces, component developers may wish to allow only parts of the interface to be adapted by other components, or limit the ways they may be adapted. Care needs to be taken when designing both the reconfigurable component interfaces and components wishing to reconfigure them.



(a) Disabling of user interface elements.



(b) Adaptation of other component's user interface elements.

Figure 8. Examples of user interface reconfiguration

Inappropriate reconfiguration may make an interface difficult or confusing to use, or even prevent users effectively using their application. For example, a collaborative monitoring agent that "forgets" to unlock a dialogue field for a component prevents that field value

being changed. Disabling, hiding or changing layout and interaction behaviour of user interface elements may adversely affect interface look and feel, impacting on overall application usability. One issue that developers and end users need to be aware of is that several components independently developed and reused may want to change a single user interface element of another component in different, incompatible ways. This introduces complex reconfiguration co-ordination problems. We have found conservative use of reconfiguration is necessary to avoid these problems, or the use of "reconfiguration agents" to manage them. Another technique is to prioritise adaptations and apply only "high priority" changes to user interface elements. A challenge is to adapt priorities as component composition occurs or users change their preferences.

Components that may wish to adapt the interfaces of related components need to be provided with general mechanisms to identify and programmatically extend these interfaces. We achieve interface reconfiguration for JViews components in the same way as extension and composition are supported: components advertise parts of their interface which may be reconfigured. User interface aspect information classes provide methods to enable, disable, hide, show and modify the display and editing characteristics of these interface elements. Some constraints on what are permissible interface reconfigurations can be specified. Components wanting to reconfigure other components' interfaces use this aspect information and standardised methods to perform appropriate reconfiguration. We have also developed some extended Java AWT class specialisations and interfaces to support more general adaptation of user interfaces by composition, extension and reconfiguration.

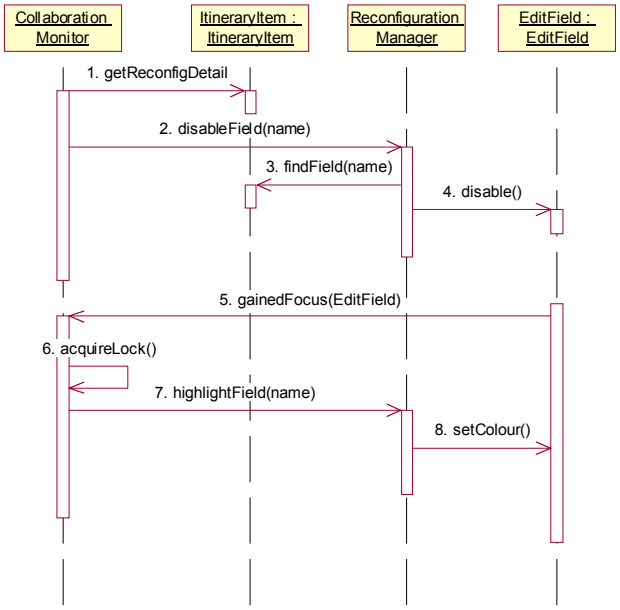


Figure 9. Simple user interface reconfiguration.

Figure 9 shows an example of reconfiguration. The collaboration monitor locates a reconfiguration manager for a component (1), and requests it to disable an edit field (2-4) to ensure it is not updated. If an edit field is about to be modified by a user (it is not disabled), it notifies the collaboration monitor (5), which tries to acquire a lock on the field among the users of the collaborative application (6). If this succeeds, it highlights the field (via the reconfiguration manager). Enabling and highlighting can be done directly to the edit field (using its aspect detail information), but the reconfiguration manager allows multiple components to reconfigure a single user interface element in a co-ordinated way.

7. Adaptation to User, Role and Subtask

Adaptation of user interfaces may be made, as in the previous examples, to extend, compose and/or reconfigure a component's user interface so related components can express their interface needs in a consistent, seamless way. Adaptation may also be required due to particular user preferences, such as a particular interface to display or interface characteristics to use. A component user interface may also need to be adapted to suit a user's role in a task model and/or a particular subtask a user is currently working on, to ensure a component presents an appropriate interface for the user.

A particular user of a component-based application may wish to specify a variety of preferences about the user interfaces the components present. This may include their preferred user interface if multiple alternatives exist for a component, default user interface element appearance characteristics, and preferred extension and composition approaches, if multiple exist for a component. For example, consider the dialogue shown in Figure 10. This is a standard configuration interface provided by our JViews user interface aspect manager allowing basic preferences about a component's user interface to be set. In this example the user may specify which alternative interfaces they want shown for itinerary item components, whether to show or hide "expert" information like performance configuration parameters in itinerary item property sheets, and any user interface configuration-related properties, such as default colours and font to use for user interface elements.

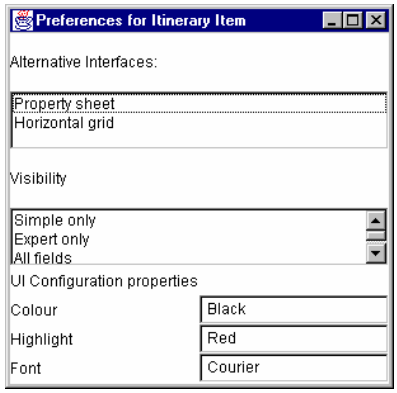


Figure 10. Example of user preferences.

We achieve user preference-based interface configuration for our components by having aspect information record these preferences as annotations and provide programmatic interfaces to access and modify them. Additionally, a user preferences component can be used which provides dialogues allowing users to specify user interface-related preference information for multiple component interfaces. Some preferences may be system-wide defaults, such as colour and font choices. Others are specific to components the preferences component is linked to, and are obtained from user interface aspect information advertised by these components. Some reconfiguration and extension properties of aspect information objects can be changed dynamically e.g. to allow a user to "turn off" certain reconfiguration approaches for some components.

Multiple users of an application typically perform a specified role, with different roles potentially wanting to use only parts of a component's user interface. Similarly, as users perform different subtasks of an overall work task, certain component user interface elements may be appropriate and useful and others may not.

In general, adaptation to user task and role is challenging because 1) a component-based system must determine the user profile, subtask and role, and 2) the appropriate adaptations to role/subtask must be specified, and there may be a large number of these in any non-trivial work domain. In our travel planner the Serendipity-II workflow system provides the role/subtask information for our task adaptors. However, for many systems it may be difficult to characterise a "work process", or to take task models of user interaction and "enact" these while the system runs to obtain role/subtask information. In our travel planner the work process model is quite simple, as illustrated in the workflow diagram on the left hand side of Figure 9. This means it is feasible to specify for some component user interfaces adaptations according to different workflow subtasks/roles. For a system with a large, complex workflow model, this may be very challenging.

As with user interface extension and reconfiguration, conflicts can occur with user preferences, role and task adaptation. For example, a panel may be hidden as its content is inappropriate for a user but another component extends the panel or reconfigures its content, adaptations which should be shown to the user. Such conflicts can be resolved by developers engineering components with priorities or with "adaptation" preferences the user can set, or simple adaptation heuristics e.g. if parts of an interface have been adapted, don't hide it.

Figure 11 also illustrates two component user interface adaptations to role and task that we have found useful in the collaborative itinerary planner. The itinerary item component's user interface has two forms: one for customers which hides some details, and one for agents. For the customer, additional reconfigurations are done depending on whether they are sketching a travel plan (subtask 1 - most fields are hidden and time defaulted) or modifying a detailed travel plan (subtask 5 - all fields visible and editable). Similar adaptation can be employed for different subtasks for the agent. In subtask 2, the agent does not require the Details or stops fields, and can have the fare code defaulted from customer preferences. In subtask 4, however, all fields need to be shown.

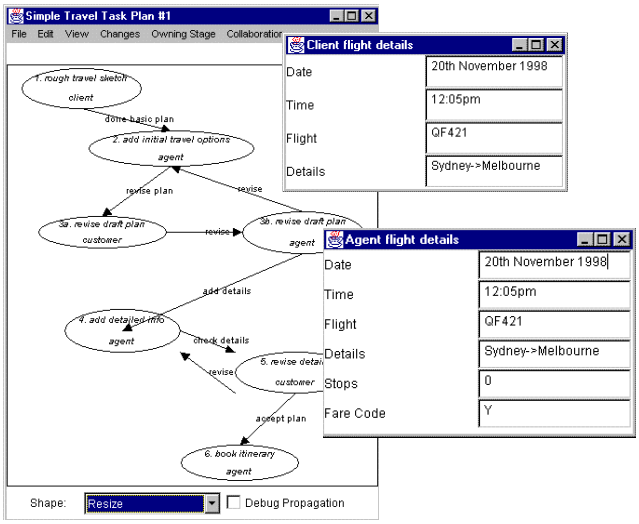


Figure 11. Examples of adaptation to task.

We achieve such role and subtask-based adaptation for our JViews component user interfaces by the use of a task adaptation component. This component is informed of Serendipity-II workflow engine enactment events and role assignments. It also provides a dialogue allowing preferences about the user interface elements of components linked to it to be set, in a similar manner to the user preferences adaptation component. User interface element aspect information is queried and annotated by information such as for a given subtask and/or role, whether or not the element should be enabled, disabled, hidden, shown etc. When the user interface elements of these components are to be displayed, JViews user interface events are detected by the task adaptation component which modifies the user interface elements based on the current role and subtask information it has.

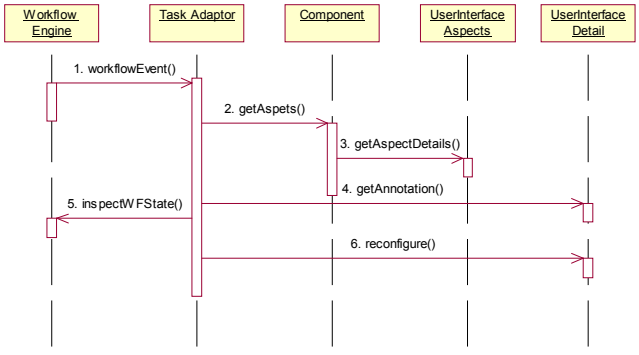


Figure 12. Task and Role Adapatation.

Figure 12 illustrates this basic process. When a workflow engine event is received (1), the task adaptor obtains user interface information from a component's aspects (2, 3). Each UI element's annotations are obtained (4), which record whether the element is relevant to particular workflow subtasks and user roles. The

workflow state is inspected (5) and appropriate reconfigurations applied to the UI element by the task adaptor (6).

8. Conclusions and Future Research

We have described an approach to engineering software components with adaptable user interfaces. High-level characterisations of component user interface elements, including provided and required elements, extensible and composable elements and element groups, reconfiguration properties, and user preference, role and subtask information are specified. Encodings of these characterisations in component implementations enables other components to access this information, and programmatically adapt a component's user interface in standardised ways. Our approach has provided us components with interfaces that can be more suitably adapted in diverse reuse situations.

Some guidelines we have identified for component developers, to guide adaptive user interface construction, are summarised below:

1. Allow for user interface extension and composition. If a component provides a user interface element like a panel, menu or button list, allow this to be extended programmatically by other components in a controlled way.
2. Use extension and composition of other components where possible. If a component needs to present an affordance or feedback element to the user, allow this to be done by extending another component interface if appropriate rather than having to open its own dialogue/window.
3. Allow component user interface elements to be programmatically identified and reconfigured in controlled ways by other components. This is usually not difficult in most UI development toolkits.
4. Capture user, role and task information where possible and allow user interface adaptations to drawn upon these. If "user" preferences can be set programmatically by other components, this may provide another adaptation mechanism e.g. to modify layout, appearance etc to suit adaptations.

Several directions for future research exist. Better-integrating adaptation needs into the component engineering lifecycle is chief among these. Software component developers need to take user interface adaptation into account during each stage of component development. Our current characterisation of user interface elements can be improved by adding more comprehensive layout, appearance and semantic constraint specification in the user interface aspects i.e. enriching the knowledge representation we currently use.

Tool support for specifying user interface aspects is currently rudimentary, with component developers specifying UI aspects in the same manner as other component aspects. Generating aspect characterisations from the user interface specification tool of JComposer would greatly improve this. Third party components can have aspect information specified in JComposer and used by JViews components. Unfortunately these third party components are not implemented with knowledge of aspects and thus can not themselves programmatically adapt JViews component interfaces. We would like to develop our user interface adaptation techniques with common component-based architectural services, such as those of Enterprise Java Beans, Jini or CORBA in future, making them more generally accessible.

A general mechanism is needed to capture user, role and subtask information in order to support appropriate adaptation of user interfaces as these change. Our work has drawn on the process enactment state of the Serendipity-II workflow tool. In general, most application end users do not have their work activities co-ordinated with such tools, making the unobtrusive acquiring of such information for interface adaptation difficult.

We plan to investigate the application of our approach to 3D, Virtual Reality interfaces and ubiquitous user interfaces, such as PDAs, which may provide a greater range of possible adaptation approaches. This may require better characterisation and use of user interface containment i.e. the properties of user interface element containers, leading to the use of "aggregate aspects". There is currently a clear separation in JViews between a component's logical model and its user interface, and a component's properties and methods are not used directly when adapting its user interface. We are investigating the specification of mappings between logical model and user interface realisation, which will include the ability to more easily adapt the appearance and behaviour of an interface based on the way logical model structures need to be composed and related to users, roles and subtasks.

Acknowledgements

Support for this research from the New Zealand Public Good Science Fund is gratefully acknowledged.

References

1. Apple Computer Inc., *OpenDoc Users Manual*, 1995.
2. Bandinelli, S., DiNitto, E., and Fuggetta, A., "Supporting cooperation in the SPADE-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, 841-865, December 1996.
3. Chappell, D. Understanding Active-X and Ole, Microsoft Press, January 1996.
4. Dachselt R. The challenge to build flexible user interface components for non-immersive 3D environments. Proceedings of 8th International Conference on Human-Computer Interaction. Lawrence Erlbaum Associates. Part vol.2, 1999, vol.2. Mahwah, NJ, USA, pp.1055-1059.
5. Eisenberg, M. and Fischer, G. (1994): Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assitance, *Proceedings of ACM CHI'94*, ACM Press, pp. 431-437.
6. Geary, D.M., Graphic Java 2, Mastering the JFC, 3rd Edition, Prentice Hall, 1998.

7. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Environment*. Reading, MA: Addison-Wesley, 1984.
8. Grundy, J.C., Mugridge, W.B., Hosking, J.G. Static and dynamic visualisation of component-based software architectures, In *Proceedings of 10th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, June 18-20, 1998, KSI Press.
9. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
10. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, Vol. 2, No. 5, September/October 1998, IEEE CS Press.
11. Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In *1999 IEEE Symposium on Requirements Engineering*, Limerick, Ireland, 7-11 June, 1999, IEEE CS Press.
12. Grunst, G., Oppermann, R., Thomas, C. G. Adaptive and adaptable systems, In Hoschka, P. (ed.): *Computers As Assistants - A New Generation of Support Systems*. Hillsdale: Lawrence Erlbaum Associates, 1996. 29-46.
13. Linton M.A., Vlissides J.M., Calder, P.R. 1989: Composing user interfaces with Interviews, COMPUTER, Vol. 22, No. 2, February 1989, 8-22.
14. Mehandjiev, N. and Bottaci, L. (1998): The place of user enhanceability in user-oriented software development, *Journal of End User Computing*, Vol. 10, No. 2, 4-14.
15. Moran DR, Cheyer AJ, Julia LE, Martin DL, Sangkyu Park. Multimodal user interfaces in the Open Agent Architecture. 1997 International Conference on Intelligent User Interfaces. ACM. 1997, New York, NY, USA.
16. Morch, A. Tailoring tools for system development*, Journal of End User Computing* 10 (2), 1998, pp. 22-29.
17. 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.
18. Oaks, S. and Wong, H. Jini in a Nutshell : A Desktop Quick Reference, O'Reilly and Associates, March 2000.
19. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
20. Paelke V. Visual presentation agents for 3D environments. 1999 International Conference on Intelligent User Interfaces. ACM. 1999, New York, NY, USA.
21. Rossel M. Adaptive support: the Intelligent Tour Guide. 1999 International Conference on Intelligent User Interfaces. ACM. 1999, New York, NY, USA.
22. 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.
23. Sanchez JA, Azevedo FS, Leggett JJ. PARAgente: exploring the issues in agent-based user interfaces. Proceedings. First International Conference on Multi-Agent Systems. AAAI Press. 1995, Menlo Park, CA, USA.
24. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
25. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
26. ter Hofte, G.H. and van der Lugt, H.J., 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.
27. Wing, H. and Colomb RM. Behaviour sharing in adaptable user interfaces. Proceedings Sixth Australian Conference on Computer-Human Interaction, IEEE CS Press, 1996, Los Alamitos, CA, USA, pp.197-204.