Developing Adaptable User Interfaces for Component-based Systems

John Grundy and John Hosking
Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john-g, john}@cs.auckland.ac.nz

Abstract

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 facilitate effective component user interface design and realisation using an extended, component-based software architecture.*

1. Introduction

Component-based software applications are composed from diverse software components to form an application [1, 14, 16, 17]. Typically many of these components 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. Components which ideally should provide a consistent interface metaphor may adopt different approaches e.g. menu items vs buttons. Components 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 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 allowing components to inspect and understand other component user interface elements, to programmatically adapt related component interfaces to suit a particular reuse situation, and for components to be able to extend and combine the interfaces of other components with their own are needed.

We describe our approach to addressing these issues. This uses a concept of component user interface aspects to describe component 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 methods and interfaces can be used to programmatically extend, compose and reconfigure component interfaces in various ways. The use of a workflow engine's state allows 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 and component implementation framework. 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 [5, 6, 7]. A screen dump from one such system, a collaborative travel itinerary planner, is shown in Figure 1. 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.

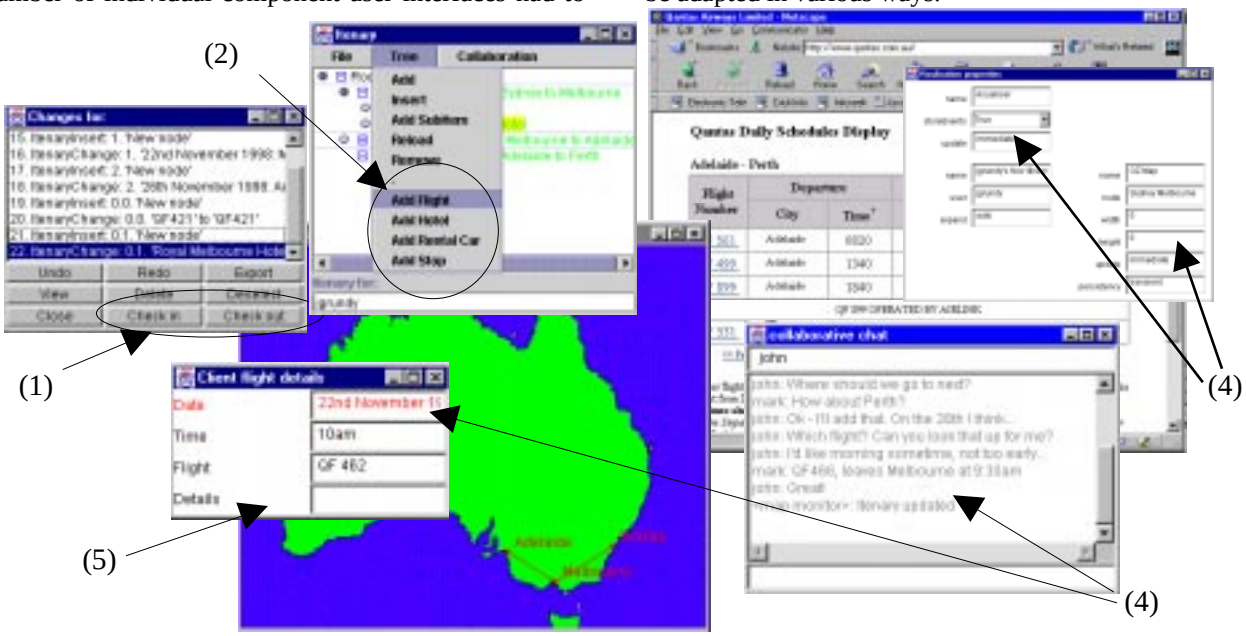


Figure 1. Some Itinerary Planner component user interfaces.

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.This seamlessly provides users access to the version control tool's facilities (1). Similarly, the itinerary editor's menu is extended by each kind of itinerary item component (2).
- *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).
- *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).
- *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).

In order to support the development of software components that are amenable to these kinds of user interface adaptation, approaches to specification, design and implementation of adaptable interfaces and needed.

3. Related Research

Common adaptive user interface techniques used by software developers include extensible menus and panels and programmatical reconfiguration of interfaces [12, 14, 4]. 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 [10] and Amulet [13], 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. Thus systems built using these frameworks must 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 [14] and OpenDoc [1], provide more powerful component introspection mechanisms that allow other components to discover properties, methods and events dynamically. Unfortunately 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. Some basic design guidelines suggest components should support adaptation of the user interfaces, and the architectures allow this, but no consistent approaches are used nor appropriate implementation support exists.

Work on adaptable user interface systems [9, 3, Refs], end user computing systems [11, 12], intelligent user interfaces [Refs], and agent-based systems [Refs] has contributed to the development of techniques supporting various kinds of interface adaption. Some adaptable and agent-based systems support a variety of techniques for designing and implementing user interface adaptation facilities [Refs]. Unfortunately most 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.

Extensible workflow systems [6], process-centred environments [2], and collaborative work tools [15, 18] 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

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 [8]. This approach uses a high-level concept of systemic *aspects* of a software application for which components provide services or require services from other components. We have used this development method, and added architecture support for it in JViews, our component development framework, 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.

We have extended our JViews software architecture and implementation framework to allow components to advertise these user interface (and other) aspects. Component developers characterise different user interface aspect details a component supports during design. This information is then encoded in component implementations. JViews includes several UserInterfaceAspectInfo classes that encode this information, and provide a variety of methods for adapting a related component's user interfaces programmatically. JViews is implemented in Java, and uses JavaBeans components to realise its user interfaces.

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 extensible affordance (of some kind) they can extend. Constraints may be specified about both the extension provision 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).

The implementers of each component encode such user interface information using appropriate JViews UserInterfaceAspectInfo classes. These classes provide standardised methods allowing this aspect information to be obtained and basic adaptation to be programmed. Alternatively, the implementers may use Java interfaces provided by JViews which support various programmatic adaptation of user interfaces.

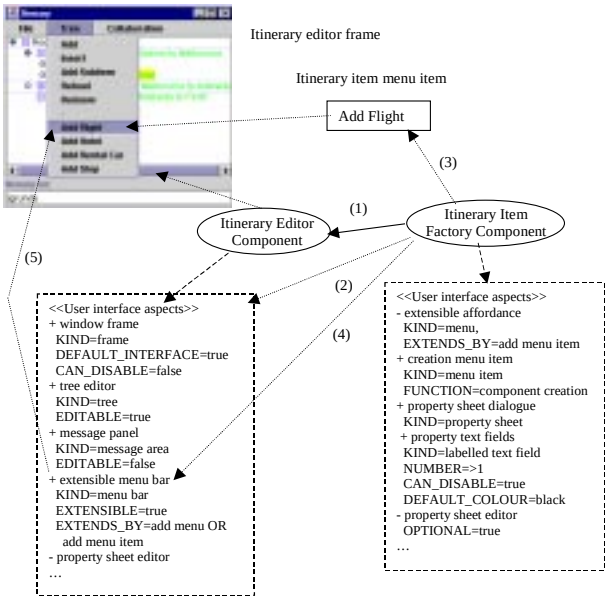


Figure 2. Extensible affordance user interface aspects.

Figure 2 illustrates the way JViews aspect classes are used to provide a decoupled mechanism for achieving this kind of user interface characterisation and adaptation, resulting in highly reusable components. Component objects are in solid line, AspectInfo objects dashed. Aspect details are each represented by their own object. Details with "+" in front are provided by the component, "-" are required. Some aspect detail properties are shown, describing various characteristics of each user interface element. For example, the itinerary item factory component requires an extensible affordance (restricted to a pull-down or pop-up menu by its KIND property) it can add a user interface element to. 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).

Each kind of itinerary item component has a factory component to create items of this kind, which is linked to the editor component. The factory obtains aspect information from the editor (1) and inspects this information to deduce the extensible affordance support of the editor (2). It then creates a menu item to allow users to access its functionality (3) and uses a standardised method provided by the extensible affordance aspect object (4) to programmatically extend the user interface (5).

Figure 3 shows a larger example of JViews components from our collaborative travel itinerary planner with some user interface aspects of these components. Note that user interface aspect details may overlap e.g. a panel aspect detail describing 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 [7] 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 itnerary editor, editing history, version control tool, map visualisation and collaborative awareness components have been reused from other applications [6].

Our JComposer CASE tool allows component developers to specify aspect information as part of their component modelling and design [8]. When JViews components are implemented, information about their user interfaces is encoded by JComposer in a standardised form via UserInterfaceAspectInfo classes. Aspect encodings can be modified dynamically, for example to change default and preferred values and to specify additional information, such as role and subtask.

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.

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. This avoids the situation of multiple, composed components presenting multiple, inconsistent interfaces to end users. The itinerary editor menu bar provides one example of this.

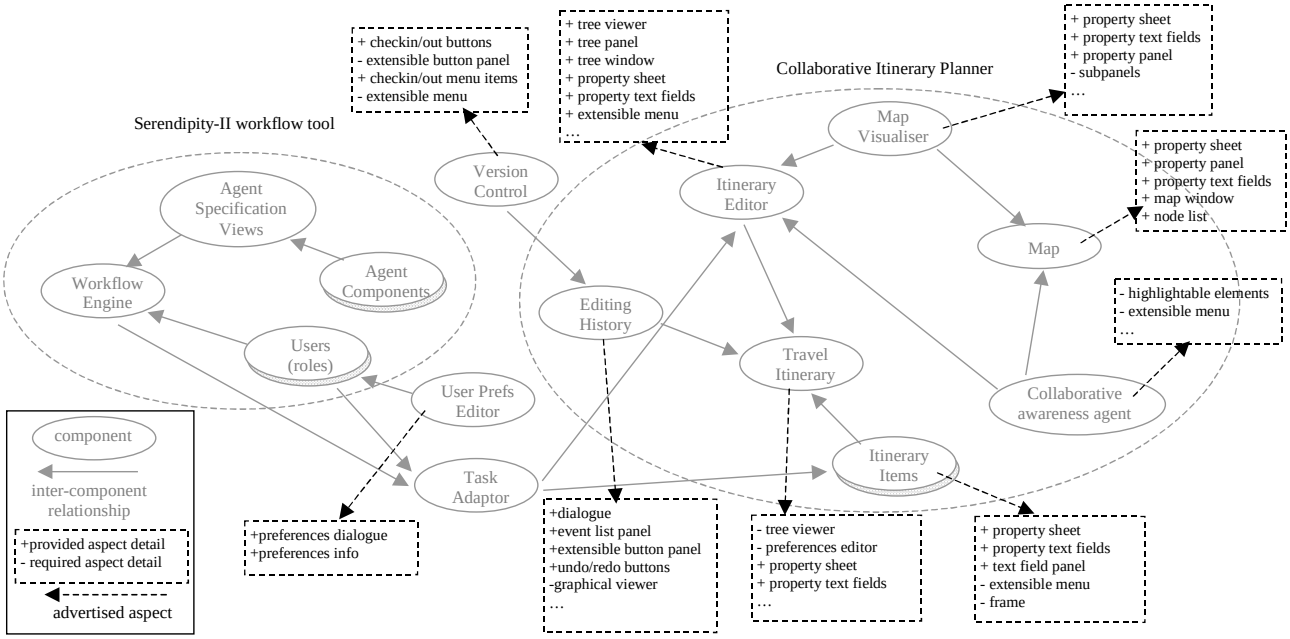
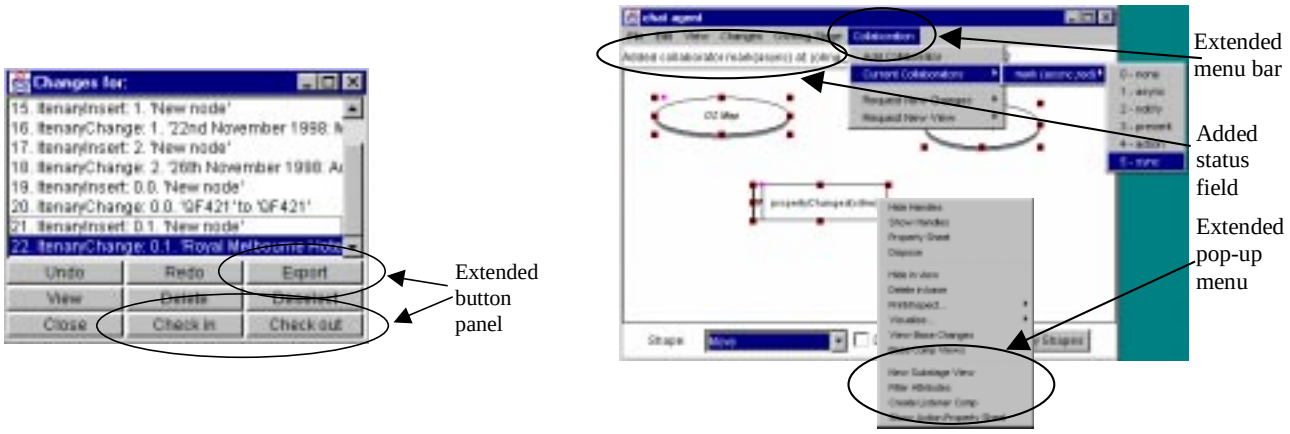


Figure 3. Collaborative travel itinerary planner architecture and component user interface aspects.



(a) Extensible panel.

(b) Extensible menus and panel.

Figure 4. Examples of user interface extension.

For another example, consider the event history component's dialogue, shown in Figure 4 (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 4 (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.

A related technique for supporting user interface adaptation is composition of multiple component interfaces. Figure 5 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.

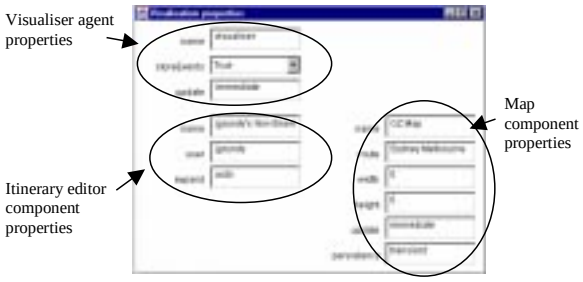


Figure 5. User Interface composition.

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 at the end of the bar, so the File and Edit menus are always kept at the start. 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 for the event history dialogue!). 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.

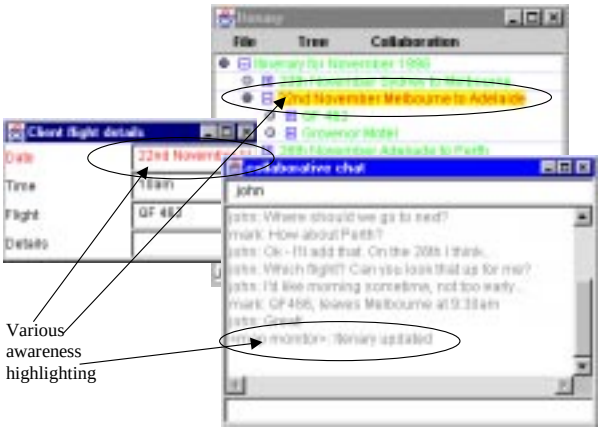
While designing and implementing user interfaces that support extension and composition takes more care and effort, the reuse costs for these components drop dramatically as new interfaces do not need to be developed nearly as frequently than for components whose interfaces are not adaptable. In addition,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

We achieve user interface extension and composition for JViews components in the way outlined in the previous section. Composition is more complex than extension, in general, with UserInterfaceAspectInfo classes providing methods that either return user interface objects to compose, or compose user interface objects obtained from other aspect information objects. A list of user interface aspect names identifying interface elements to compose can be specified and used dynamically. For example, the map visualisation component specifies the "property sheet" of related components, such as those of the map and itinerary editor, are to be composed and displayed in its property sheet dialogue. When this dialogue is opened, references are acquired to the property sheet panel objects of any related components a composed user interface constructed with these.

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 constraints. A component may also make use of the user interface elements provided by a component in ways not anticipated by the original developer.

(a) Disabling of user interface elements.



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

Figure 6. User interface reconfiguration

Figure 6 illustrates examples of such reconfiguration from our collaborative itinerary planner. In Figure 6 (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 6 (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 characteristics of parts of their user interfaces. For example, the item being edited in the itinerary tree editor is highlighted, as is the field being edited in an item property sheet dialogue. The agent also uses the collaborative chat messaging tool for notification, sending "map monitor" messages.

When designing user interfaces, component developers may wish to allow parts of the interface to be adapted by other components or limit the ways they may be adapted. 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 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.

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 7. This is a standard configuration interface provided by our UserInterfaceAspect class 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 7. Example of user preferences.

We achieve user preference-based interface configuration for JViews components by having aspect information classes record these preferences as annotations and provide interfaces and data management methods 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 are appropriate and useful and others are not. Example task model and role assignments are illustrated in Figure 8, from the Serendipity-II workflow management system.

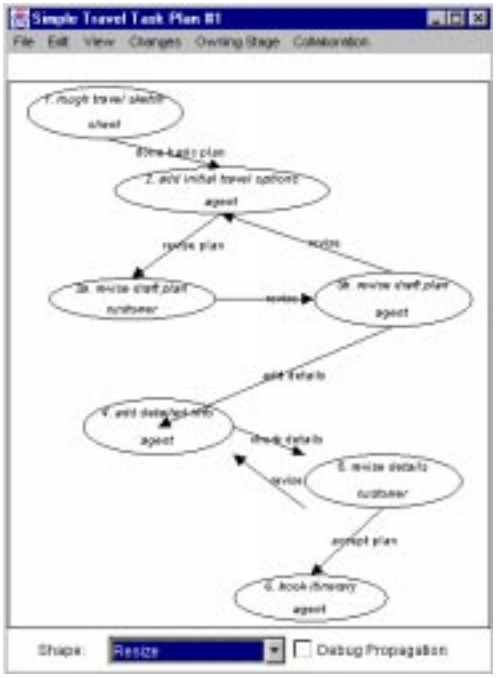


Figure 8. Simple itinerary planning process model.

Figure 9 illustrates some 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 in Figure 9 (a) has two forms: one for customers which hides some details, and one for agents. In Figure 9 (b), two quite different interfaces for itinerary item components are presented to the travel agent, depending on whether they are sketching a travel plan (subtask 1.2) or modifying a detailed travel plan (subtask 1.5). Similar adaptation can be employed for different subtasks for the travel 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 9. Examples of adaptation to task.

We achieve such role and subtask-based adaptation for 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.

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 interface via standardised methods and interfaces. Our approach has provided us components with interfaces that can be more suitably adapted in diverse reuse situations. Most of our JViews components now have better-integrated user interfaces than achievable using standard Java Beans.

Characterisation of user interface elements can be improved by adding more comprehensive layout, appearance and semantic constraint specification. Tool support for specifying user interface aspects is currently rudimentary, and generating aspect characterisations from the user interface specification tool of JComposer would 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. We also plan to investigate the application of our approach to 3D, Virtual Reality interfaces and ubiquitous user interfaces, which may provide a greater range of possible adaptation approaches. 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. 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.
4. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Environment*. Reading, MA: Addison-Wesley, 1984.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. Linton M.A., Vlissides J.M., Calder, P.R. 1989: Composing user interfaces with Interviews, COMPUTER, Vol. 22, No. 2, February 1989, 8-22.
11. 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.
12. Morch, A. Tailoring tools for system development, *Journal of End User Computing* 10 (2), 1998, pp. 22-29.
13. 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.
14. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
15. 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.
16. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
17. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
18. 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.