Supporting aspect-oriented component-based systems engineering

John Grundy

Department of Computer Science, University of Waikato
Private Bag 3105, Hamilton, New Zealand
jgrundy@cs.waikato.ac.nz

Abstract

Current approaches to component-based systems development do not adequately capture high-level knowledge about component provided and required services for use during design, implementation and run-time deployment. We describe a new approach to engineering such systems that characterises components by the various "aspects" of the overall system each component provides to or requires from end users or other components. These aspects include user interface, persistency, distribution, collaboration, inter-component relationships. Appropriate software architecture and CASE tool support is needed in order to effectively describe, reason about and implement this encoding of high level knowledge about components. We motivate the need for aspect-oriented component engineering, describe and illustrate our approach and its current software architecture and development tool support, and report on our component-based system development experiences.

1. Introduction

Component-based systems, or "componentware", have become popular as an approach to systems development which promises improved reuse, compositional systems construction, and end user configuration support [10, 20]. Various software architectures and implementation frameworks have been developed based on the notion of software components, including OpenDoc [1], COM [18], JavaBeans [16], and JViews [5]. Various development tools and methodologies have been developed to support component-based software construction [2, 15, 17, 20, 22].

Currently component-based architectures and development approaches focus on designing and implementing component interfaces, and encoding low-level information about components for use at run-time, such as COM type libraries and JavaBeans introspection [16, 18]. We have found such approaches do not adequately capture knowledge about component provided and required services, for neither other components nor end users to use [6, 7]. This makes component design, implementation and deployment more difficult than if components could publicise more information about themselves, and this information was used to better guide component design and implementation.

To overcome these problems, we have been working on a new component development approach we call aspect-oriented component engineering. This focuses on identifying various aspects of an overall system a component provides to or requires from other components and end users. Aspects include systemic characteristics of systems such as user interfaces, component persistency and distribution, collaborative work and end user configuration support, and may include domain-specific characteristics. Component developers use aspects to describe more fully component required and provided services, and use aspects to guide component implementation. We describe extensions we have made to a software architecture, its ADL, implementation framework and run-time support to effectively support aspect-oriented component engineering. We also discuss enhancements to a CASE tool for this architecture. The overall aim of this research is to better capture, describe and reason with more knowledge about components than present approaches, during component desigm, implementation and deployment.

We begin with a motivating example application for this work. We then overview the concept of aspect-oriented component engineering, and discuss appropriate software architecture, ADL and implementation framework support for this methodology. CASE tool support and run-time usage of component aspects is discussed. We conclude with a report of our experiences with this approach, compare it to existing approaches, and overview possible future work.

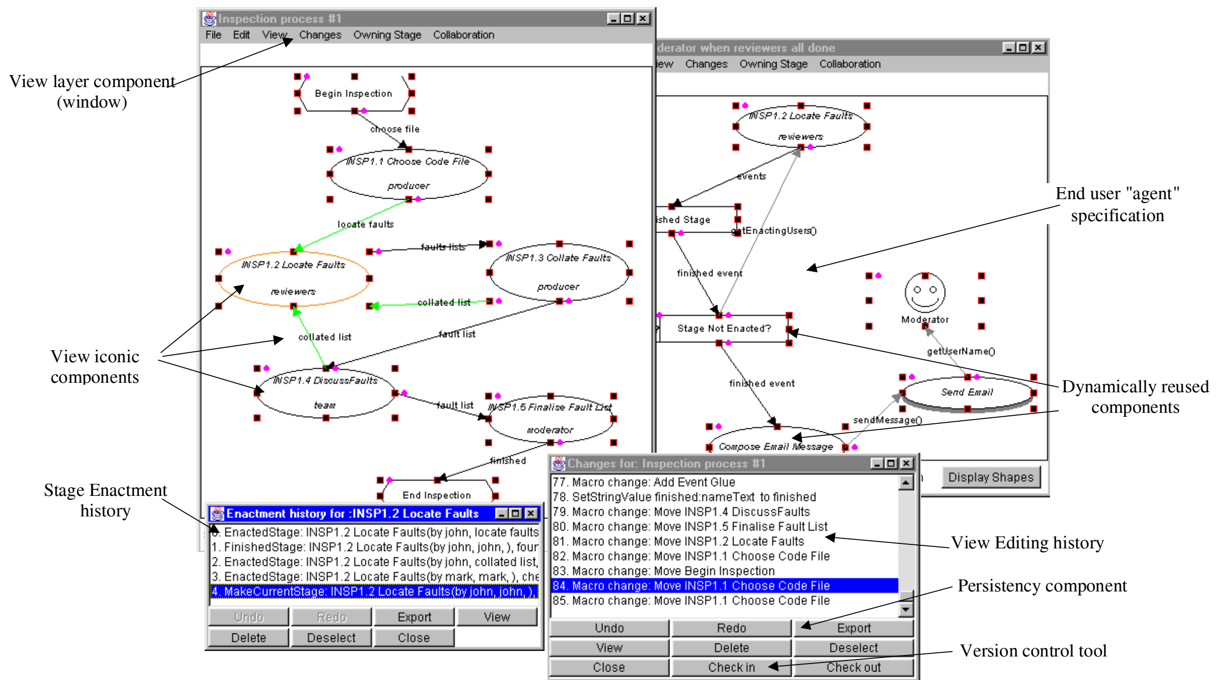

Figure 1. Example component-based application: Serendipity-II.

2. Motivation

Figure 1 shows an annotated screen dump from Serendipity-II, a software process modelling and enactment environment [7], that provides multiple, enactable views of a software process. Visual languages are used to describe work processes and end user deployed software agents, and multiple user support is provided with collaborative view editing and versioning.

Serendipity-II was developed using a component-based software architecture, where a variety of software components were composed to form the environment. The environment can also be extended at run-time with new components "plugged in" by end users to provide dynamically deployable software agents and tool integration. Many components have been reused from other systems, including the event history component (reused for both enactment and editing histories and reconfigured for use in Serendipity-II), and the version control and event broadcasting components, which have only some of their capabilities used by Serendipity-II. When developing component-based applications a range of complex engineering issues arise. These include:

- Designing software components to appropriately provide certain functionality to other components and end users, and require functionality from other components. Reasoning about complex provided and required inter-relationships of components is difficult in any non-trivial system.
- Reusing existing components when developing applications, particularly user interface, collaborative work and middleware components. Reasoning about how capabilities reused components provide and require fit with other components is very challenging.
- Component user interface facilities for end users. The disparate components that are composed to produce a component-based application should provide a consistent look-and-feel, despite many components not being designed with each other in mind. Some components should thus provide windows, menus, button panels, tool bars and dialogues which can be suitably extended and modified by other components.
- Components supporting middleware functions like data persistency, distribution and collaborative work need to be appropriately reused and configured. Such components often need to be used by all components in an application requiring such facilities, and matching middleware capabilities to component functional and non-functional requirements is crutial.
- End user application configuration is increasingly demand to achieve a major aim of plug-and-play component technologies. To support this, end users must have access to knowledge about component capabilities they can understand and use. Other components need similar knowledge to automatically reconfigure themselves and related components.

3. Aspect-oriented Component Engineering

3.1. Concept of Component Aspects

To overcome current problems of lack of knowledge about software component capabilities during design and deployment we have been working on a new component development methodology, aspect-oriented component engineering. Our approach uses a concept of "aspects" of an application for which components provide capabilities to or require capabilities from other components or end users. Aspects provide multiple perspectives onto the provided and required services of software components, allowing functional and non-functional requirements to be organised using a set of systemic categorisations.

We have developed a set of common aspects of an overall application components provide services to or require services from other components. These categories of systemic aspects include user interface, component persistency and distribution, collaborative work, inter-component relationship, and end user configuration. This is similar though more generalised concept of an aspect as used by aspect-oriented programming [11]. Other, domain specific aspects or different systemic aspects could also be used to categorise component services, such as process modelling and process enactment capabilities for Serendipity-II. Our systemic aspect categorisations have worked well in our problem domain of collaborative, multiple view design tools and Information Systems.

Aspect information is captured and documented during component requirements engineering and software component design, providing knowledge at differing levels of abstraction about component capabilities. Aspect details may relate closely to software component interfaces and services, or span multiple interfaces. Different aspects may share software component interface operations or be disjoint. For some aspects each component provides and/or requires certain capabilities to/from other components and end users. Each of these "aspect details" may describe particular interface or functional capabilities a component provides or requires, and may also describe non-functional constraints on the provision of or use of such capabilities. Some provided aspects may be transitive, i.e. indirectly provided by another component related to the "providor". Not all required aspects may necessarily need to be fulfilled to use a component, but non-provided aspects mean services dependant on these can't be used in the component configuration. Constraints associated with aspects control the matching of provided and required aspects, and may also be transitive across several related components. Groups of components can have aggregate provided and required aspects that relate to the group as a whole and which can be reasoned about. Aspect-related knowledge can be codified in component implementations and provided to other components or end users and used to validate run-time component configurations.

Figure 2. Some example components, aspects, aspect details and aspect usage from Serendipity-II.

3.2. Examples from Serendipity-II

Figure 2 illustrates some aspects for some Serendipity-II process model view-related components. Some components have several different kinds of aspects and associated aspect details they provide and/or require, for example view layer and event history components. Others have a more narrow focus and consequently less aspects of the overall system they contribute to, such as the event broadcasting and persistency management middleware components. Components are related by inter-component links (grey directed arcs), and most provided and required aspects of these linked components should be matched up if a system configuration is valid (solid directed arcs). Some provided capabilities of some components are not used by Serendipity-II, hence no other component requires such aspects.

3.3. Aspect-oriented Component Engineering Process

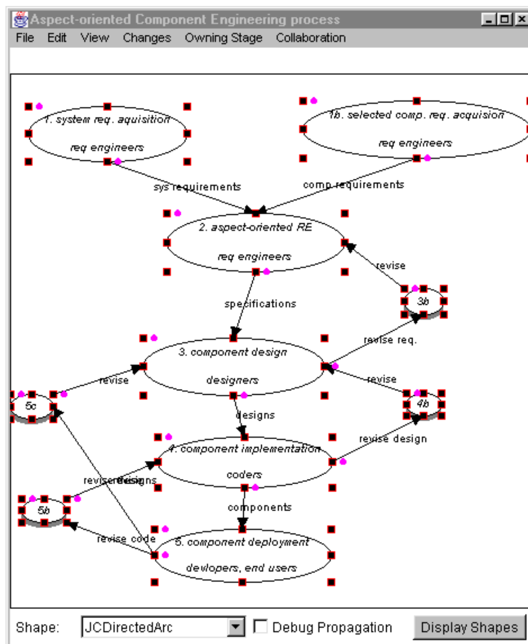

Figure 3. Basic AOCE process.

A key difference between aspect-oriented component engineering (AOCE), illustrated in Figure 3, and conventional OOA/D is that component requirements may be reasoned about from an application perspective i.e. an overall system's requirements, or from individual component or groups of reusable components. We have found using aspects to characterise component capabilities allows the relationship between domain-specific and generic, reusable components to be more easily reasoned about and specified. Software component design refines both component and aspect specifications, choosing appropriate user interface, middleware and database technologies to implement a specification. Component implementation codifies aspect information in implementation classes, and leverages design and requirements-level aspects to validate component implementations. Component aspect information is accessible an run-time by other components and end users.

4. Software Architecture Support

In order to effectively support aspect-oriented component design and implementation, we have enhanced an existing component-based software architecture, its architecture description language, and its implementation framework to support aspect description and codification. The JViews software architecture was used to develop applications like Serendipity-II, and provides a set of component-based abstractions for developing multiple view, multiple user design tools [5]. Figure 4 shows an example of how JViews is used to describe and implement the software architecture for Serendipity-II. JViews systems are comprised of components, inter-component links, and relationship components. JViews provides a set of novel event subscription, generation, propagation and response mechanisms that lead to highly reusable and extensible components. Additional abstractions provide multiple view, user interface, software agent, persistency, distribution, and collaborative work support [5].

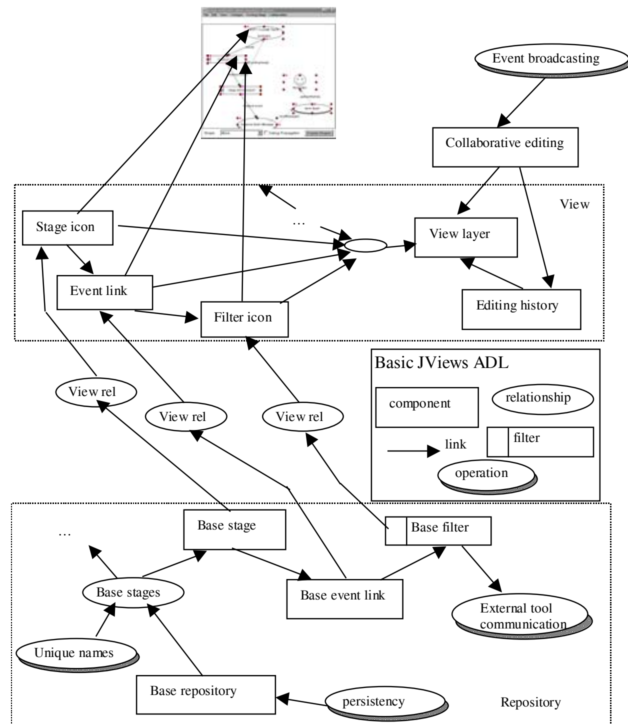

Figure 4. Serendipity-II modelled using JViews ADL.

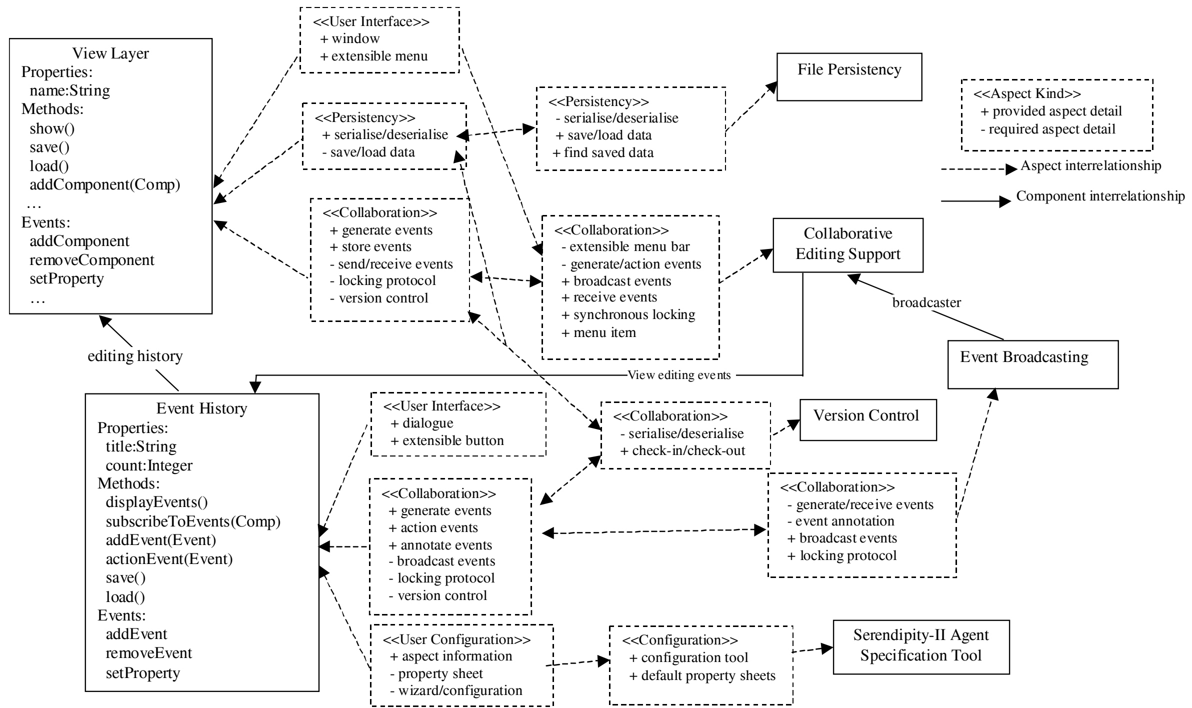

Figure 5. JViews visual Architecture Description Language extensions to support aspects.

```
component View Layer
properties
        name : String
        …
methods
        displayView()
        saveView(OutputStream)
        addButton(Button)
        …
events
        WindowEvent(ID)
        PropertyChanged(Component,Name,OldValue,NewValue)
        …
aspect User Interface
  provides extensible menu bar : EXTENSIBLE_AFFORDANCE
        -- other components can add additional menu items to main menu bar or menu bar items
        AFFORDANCE=menu_bar -- menu bar (c.f. pop-up, buttons etc.)
        ORDER=fixed -- can not change order of default view menu items
Aspect Collaboration
  provides "generate events" : EVENT_SOURCE
        -- generates events before and after change operation
        GENERATE=before, after -- before & after state change event generation
        AGGREGATE=true -- propagates all aggregate (view component) events
        TRANSITIVE=true -- all transitive events propagated
  provides "store events" : EVENT_STORE
        -- stores all non-transitive events
        STORE_KIND=state_change -- does not store non-editing events (open, close window etc)
        SOURCE=self, aggregates -- events stored must only originate from view or view aggregate components
  requires "send/receive events" : EVENT_EXCHANGE
        -- to support synchronous/semi-synchronous editing of view needs editing event transport between JViews environments
        SERIALISATION=event_source -- must use view layer/view component serialisation/deserialisation methods
  requires "locking protocol" : SYNCHRONISATION
        -- synchronous editing needs locking protocol activated on before state change event notification
        LOCKING=pessimistic, exclusive
  requires "version control" : VERSIONING
        -- asynchronous editing requires versioning of view editing history
        GRANULARITY=event, component, aggregates -- should support exchange of events or entire serialised view with other users
…
end View Layer
```

Figure 6. Aspect details in the textual JViews Architecture Derscription Language.

The visual JViews Architecture Description Language (ADL) was enhanced to add component aspects, aspect details, and aspect usage notational symbols. The textual ADL was enhanced to support the specification of aspect detail information and constraints. Figure 5 illustrates the additonal visual ADL notation used to describe aspects for JViews components. Each provided and required aspect detail has textually specified extra information and constraints, illustrated in Figure 6. Each kind of aspect detail has a set of properties whose values can be specified and constrained, describing detailed aspect information and used to reason about inter-component aspect usage.

JViews is implemented using Java, and its component model extends the Java Beans component API [5]. We have extended the JViews framework classes to support the encoding of aspect information using a set of AspectInfo

classes, in a similar way that BeanInfo and Service classes describe enterprise JavaBean low-level interfaces. Our aspect information encodes high-level, categorised knowledge about different kinds of component capabilities, however, which can be perused and understood by end users of component-based applications. AspectInfo classes are used to not only publicise component provided and required aspects, but to provide rules to validate component configurations.

Aspect information is used to assist component implementers in providing standardised access mechanisms to component capabilities, making components more reusable without hard-coded knowledge of other component interfaces. AspectInfo classes have methods for run-time configuration of components, providing a decoupled way of accessing component functionality. JViews also provides Java interfaces that can be used to provide standardised access to component aspects. We have developed some basic design patterns and reflection mechanisms that AspectInfo classes can use to access aspect-related component functionality at run-time.

5. Development Tool and Run-time Support

In order to effectively develop complex, JViews-based applications we have built a CASE tool for JViews called JComposer [5]. JComposer provides multiple views of systems using the JViews ADL, and supports collaborative editing of these views with sophisticated inconsistency management support. We have extended JComposer to allow developers to describe component aspects, aspect details and aspect usage. Basic aspect properties and constraints can also be specified, and inter-component aspect usage checks performed. Both requirements-level aspects and design-level aspects can be represented, with simple refinement relationships between each. Basic consistency checking is used when aspect details are modified. Figure 7 (a) shows an example of adding aspect information for some Serendipity-II components using JComposer. When design-level aspect information has been specified and basic aspect usage checks performed, JViews component implementation code is generated, including aspect codification.

When generating JViews class specialisations, JComposer encodes design-level aspect information for a component. This describes knowledge about the component's aspects that can be accessed and used at run-time by other components (through JViews AspectInfo classes), or by end users via a dialogue-based interface. End user support for accessing component aspect information is supported by tools that query components for their aspect specifications and present this to end users. This allows them to use knowledge of component capabilities categorised in standard ways and that is high-level in nature.

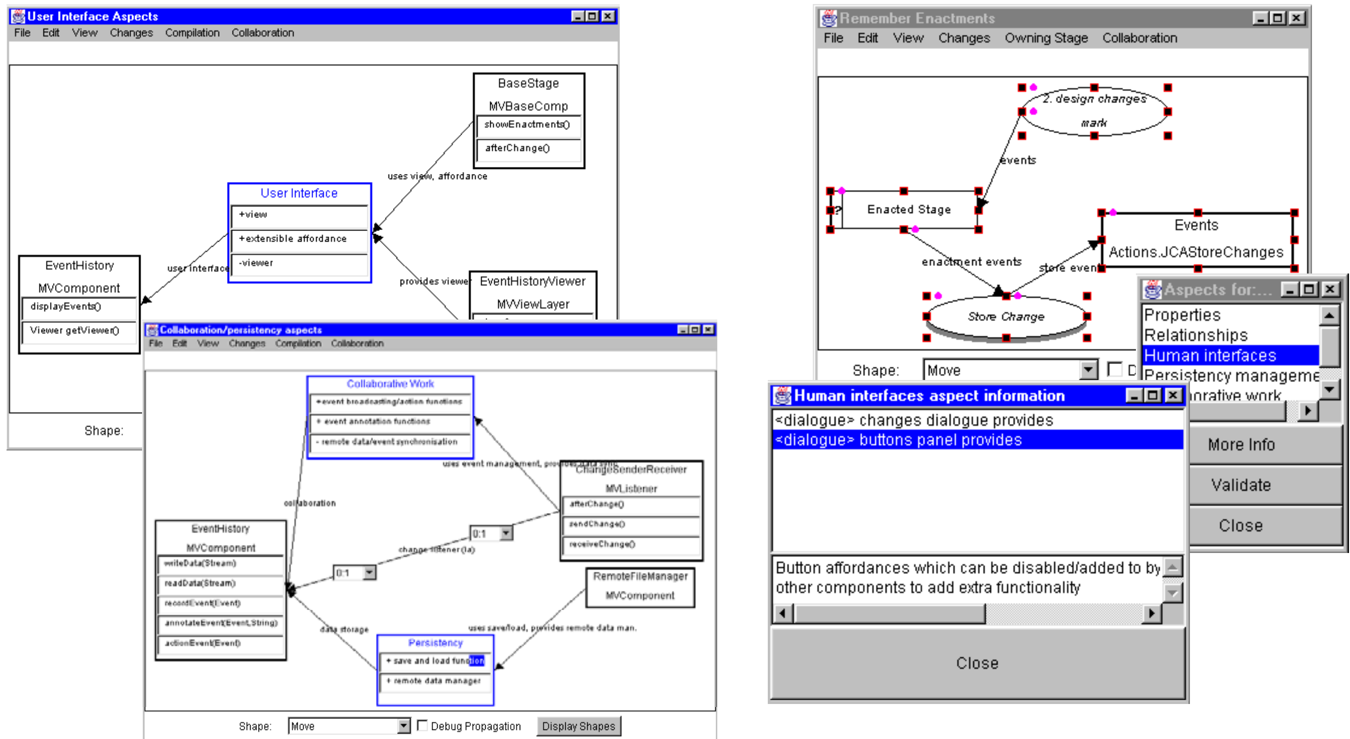

Figure 7. (a) Designing with aspects in the JComposer CASE tool. (b) Run-time use of Aspects in Serendipity-II.

Figure 7 (b) shows a simple example of end-user use of aspect information in Serendipity-II. The end user is building a simple notification agent by reusing and connecting component representations in the Serendipity-II agent specification tool. The user can view detailed information about components, using the aspect information encoded in these components by JComposer to inform them of the component's various provided and required aspects.

7. Discussion

Most current component-based system development methodologies focus on software component interface design, and tend not to consider component service requirements [15, 6]. We found during development of several component-based applications such approaches produce components that have user interfaces, end user configuration, persistency, distribution and collaborative work capabilities that are either not adaptable enough or are inappropriate in some situations the components could be reused. In our original Serendipity-II system, many component user interfaces and end user configuration facilities proved inadequate or inappropriate in the diverse situations they were reused. Many components with user interface facilities did not allow these to be extended or combined with those of other components, making the overall application interface poor.

Some component design approaches try to take into account component interface requirements [17] or system-wide component properties [21], but still tend to mainly focus on low-level component object interface characteristics. The need for reusable components that can be "trusted" to perform in appropriate ways in diverse situations has become apparent [13], due to the lack of dependability of component services using current design techniques and implementation architectures [23]. We have used our aspect-oriented component engineering approach and supporting software architecture and tools to reengineer many JViews and Serendipity-II components, and to develop several new, reusable components. Our aspect-oriented approach emphasises design for reuse, trustability and support for component extension. This dramatically improved most Serendipity-II component user interfaces and the overall application interface for end users.

Current component-based software architectures, such as Java Beans [16], CORBA [14] and COM [18], and support tools, such as Visual Age [9] and JBuilder [2], tend to be focused on the provision of low-level component-based system capabilities. The advertising of component capabilities in these architectures, using BeanInfo classes, interface specifications and type libraries respectively, do not lend themselves to capturing high level knowledge about component capabilities. This information is thus not generally suitable for allowing end users to understand how to reuse components dynamically nor support other components reasoning about a component's high-level characteristics. High-level aspect information greatly assists end user understanding of reused component functionality, and we found developers reusing third party components also appreciated this extra component design information. Matching up middleware (persistency, distribution and collaborative work) provided capabilities to required capabilities has proved much easier with aspects.

Various systems support end users or software agents reconfiguring system behaviour dynamically. These include agent-based systems [4], workflow systems [19, 3, 7], adaptive user interface systems [8], and end user computing systems [12]. Agent-based systems often need third party agents to communicate knowledge about their respective capabilities, and provide interfaces allowing reconfiguration, similar to Serendipity-II components. Often such inter-agent communication techniques focus on domain-specific data and functions, rather than common systemic functionality and constraints, limiting the sharing of inter-agent user interface, collaboration and distribution mechanisms. Most workflow and process-centred systems do not utilise component-based architectures, limiting the degree to which they can be extended [7]. Many component-based systems require similar user interface extension and end user configuration capabilities to adpative and end user computing applications, but access to knowledge about third party component capabilities needs to be improved for both software components and end users.

Aspect-oriented Programming uses a notion of systemic aspects of a system to "weave" code managing data persistency and distribution management, which have been independently codified as aspect information, into object-oriented programs [11]. However, components typically provide services to manage one or more such systemic aspects for a component-based application, and in addition require services from other components that manage other such aspects. If component interfaces are carefully designed, this avoids a need for code weaving that conventional object-oriented programs require to use aspect-oriented programming, and which can't usually be supported for COTS components.

We are extending our architecture, framework and support tools to support more formal specification and checking of aspect usage constraints. We are using aspect information to index and improve retrieval of component specifications in a component repository. We are improving JComposer's aspect information generation capabilities, including support for using aspect-related design patterns and Java interfaces. Mapping of component events and operations

using aspects relating to multiple view and tool integration support is planned, using the notion of aspect-oriented programming's "weaving" of aspect information.

8. Summary

Current component-based system development approaches do not adequately capture enough knowledge about component capabilities to guide component design and implementation, nor do current component implementation architectures provide this knowledge to end users and other software components at run-time. We have developed aspect-oriented component engineering to overcome these problems by categorising provided and required capabilities into system-wide aspects each component addresses. We have extended a component-based software architecture and its Architecture Description Language to describe this extended knowledge of component capabilities, and have extended supporting CASE tools and run-time environments to make use of this knowledge during design, implementation and deployment. Preliminary experiences with our approach and tools have been very positive.

Acknowledgements

Support from the New Zealand Public Good Science Fund and the helpful comments of the anonymous reviewers are greatfully acknowledged.

References

1. Apple Computer Inc., *OpenDoc Users Manual*, 1995.
2. Borland Inc, *Borland JBuilder™*, Borland Inc, http://www.borland.com/jbuilder/, 1998.
3. Fernström, C., "ProcessWEAVER: Adding process support to UNIX," in *2nd International Conference on the Software Process,* IEEE CS Press, Germany, Feb. 1993, pp. 12-26.
4. Finn, T., Labrou, Y., and Mayfield, J. KQML as an agent communication language, *Software Agents*, MIT Press, 1997.
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. 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.
9. IBM Inc, *VisualAge™ for Java*, 1998, http://www.software.ibm.com/ad/vajava.
10. Jell, T. *Component-based Software Engineering*, SIGS/CUP Publications, 1997.
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J. Aspect-oriented Programming, In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, Finland, June 1997, Springer-Verlag, LNCS 124.
12. Mehandjiev, N. and Bottaci, L. The place of user enhanceability in user-oriented software development, *Journal of End User Computing*, Vol. 10, No. 2., 1998, pp. 4-14.
13. Meyer, B., Mingins, C., and Schmidt, H. Providing Trusted Components to the Industry, *IEEE Computer*, May 1998, pp. 104-15.
14. Mowbray, T.J., Ruh, W.A. *Inside Corba : Distributed Object Standards and Applications*, Addison-Wesley, 1997.
15. Netscape Communications Inc, *Visual Javascript™,* 1998, http://www.netscape.com/.
16. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
17. Rakotonirainy, A. and Bond, A. A Simple Architecture Description Model, *In Proceedings of TOOLS Pacific'98*, Melbourne, Australia, Nov 24-26, 1998, IEEE CS Press.
18. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
19. Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B., and Irwin, K., A Business Process Environment Supporting Collaborative Planning, *Journal of Collaborative Computing*, Vol. 1, No. 1., Chapman-Hall, 1994.
20. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
21. Szyperski, C.A. and Vernik, R.J. Establishing system-wide properties of component-based systems: a case for tiered component frameworks, *OMG/DARPA Workshop on Compositional Software Architecture*, Monterey, California, Jan 6-8 1998.
22. 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.
23. Weyuker, E.J. Testing Component-based Sotware: A Cautionary Tale, *IEEE Software*, Sept/Oct 1998, pp. 54-59.