# A Method and Support Environment for Distributed Software Component Engineering

John Grundy

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

## Abstract

*Engineering component-based software systems is challenging, and made even more difficult when multiple developers are involved. A suitable software process for distributed component engineering is required, along with appropriate development notations and collaborative work supporting tools. This paper describes a component engineering methodology we have been developing, along with examples of its notations, development tools and tool collaborative work facilities. Key characteristics of the process and notations include their use of multiple perspectives on component capabilities and the extension of the Unified Modelling Language to capture these. We have integrated and extended several software tools to support the use of this new component engineering method. We have also provided a range of collaborative work facilities in these tools to facilitate collaborative component engineering. We illustrate some of these tools in use during multi-user development, and briefly discuss the architectural realisation of these tools and their integration.*

**Keywords:** software components, processes, methods, aspects, software engineering environments, collaborative work

## 1. Introduction

Component-based systems engineering shares many commonalities with traditional software systems engineering, and uses many common modelling and implementation approaches and tools. However, it brings some new challenges as well as preserving some common problems relating to managing distributed software development projects [29, 30, 13, 1]. These include the use of new, often highly dynamic and weakly defined software processes, the need to heavily leverage existing software component designs and implementations in a consistent way, and while it often allows developers to work more independently, at various times still requires developers to tightly collaborate and co-ordinate their work.

We have been working for some time on developing flexible, open, internet-based software engineering environments using component-based techniques [11, 13, 15]. Recently we have been developing a new component engineering methodology, and have been extending our software tools to help support large-scale component-based systems engineering with this method [12]. Assembling these tools into a component engineering environment that provides various collaborative work capabilities enables multiple, potentially widely distributed developers, to incrementally build and assemble complex applications using component technology. The main features our method and tools provide include:

- *Process modelling and enactment*. Software process management is necessary to both guide developers and track the work they have done by in a geographically and temporally distributed fashion. Even developers working together locally can greatly benefit from such support to better plan, manage and co-ordinate their work [1, 5, 11]. We have developed a basic process model for distributed component engineering and realised this in a sophisticated process modelling and enactment environment. Processes are used to plan, manage and co-ordinate multiple developers. Software agents can also be deployed to enhance developers' work co-ordination.
- *Component engineering notations and tools*. We have extended a UML-based CASE tool to better-support component modelling, particular to incorporate a notion of horizontal slices of component functionality (that we call component "aspects"). A software architecture modelling tool and run-time component visualisation tool complement this design and implementation tool's functionality.

- *Collaborative editing*. Developers need to exchange software development artifacts and collaboratively edit these in various ways, ranging from fully synchronous to asynchronous editing [7, 11]. They also need support to version artifacts and manage component-based systems configurations made up from multiplely versioned artifacts [20]. We have added collaborative editing support to our tools via plug-in software components.
- *Component management*. Component-based systems leverage existing software artifacts (components) heavily. A shared repository of such artifacts is necessary to facilitate distributed component-based systems development [16]. We have developed such a shared repository for our various modelling and implementation tools to use.

In the following section we overview related work in these areas, particularly identifying gaps and weaknesses in current development methods and tools for distributed component engineering. We describe a component engineering environment that includes: a component analysis, design and implementation tool; a software process management tool; a software architecture modelling tool; a component implementation and testing tool; and various 3rd party tools integrated by the use of component technology. We describe the collaborative work support features of these tools, particularly the recent work we have done enhancing their distributed usage and tool integration facilities. These include the integrated process support, collaborative editing, distributed work co-ordination and component repository facilities.

## 2. Related Work

Current component engineering methods, such as Catalysis [8] and Select Perspective™ [1], take varying views of what a "component" is, often focusing on software components as low-level design artifacts. None adequately characterise component capabilities from component function-independent perspectives. Similarly, most current component engineering technologies, such as DCOM, JavaBeans and CORBA C-IDL [23, 24, 29] generally focus only on characterising and describing vertical component provided service functionality (i.e. interfaces). Aspect-oriented Programming (AOP) [19] and Adaptive Programming (AP) [22] have become popular approaches to describing design-level aspects of programs, and incorporating support for these in object-oriented programs via code weaving techniques and component adaptors. Dynamic composition and re-configuration of systems is difficult in AOP, however, and component management challenging in AP. In almost all of these methods and technologies, it is difficult to design components for a wide variety of reuse situations, but also very difficult for different developers to share and talk about component characteristics. We found during development of several component-based applications such approaches produce components that have user interfaces, end user configuration, persistency, distribution, security and collaborative work capabilities that are either not adaptable enough or are inappropriate in some situations the components could be reused. Some component engineering methods [31] take into account component interface requirements and system-wide component properties when designing and deploying components. Such approaches have, however, typically only been used for characterising limited forms of component services.

Many tools have been developed to assist with component engineering. These include commercial tools like Rational Rose™ [25], JBuilder™ and Visual Age™, to a wide variety of research prototypes like Clockworks [10], MOOSE [9], Escalante [21], Argo/UML [27] and SYNTHESIS [6]. Most of these tools provide a variety of modelling notations and implementation support. Most of this tool support for software component development heavily focuses on low-level component interface design and component implementation. There currently exists very few tools for distributed component engineering that provide high level modelling and analysis support for complex component interactions and little support for distributed component-based system debugging and visualisation.

A wide variety of work has been done in building collaboraibve supporting software tools. Tools like FIELD [26] use a message-based approach to support some aspects of integration and synchronous or semi-synchronous work. Such approaches tend to work reasonably well, though rather restrictive limitations are imposed on the degree of synchronous support offered, the extensibility of the environments and the degree of process co-ordination support possible. Another approach is to develop synchronous editors for design-level or even code-level work, examples including Mercury [17] and ConversationBuilder [18]. Various toolkits to support building such applications have been developed, including GroupKit [28], MetaMOOSE [9], and Suite [7]. Unfortunately most collaborative editing tools and toolkits lack adequate support for process management and work co-ordination, and often focus heavily on synchronous work whereas most software development activity is asynchronous.

Process-centred environments aim to guide or enforce the use of a codified software process to control software development. Examples include EPOS [5], SPADE [2], and Serendipity-II [11]. Many of these environments do not integrate well with other tools for performing collaborative software development, however, though some have had limited collaborative editing support and tool integration support added [2, 11].

Shared software artifact repositories have been important in supporting primarily asynchronous work. These are also needed in order to support component-based software engineering where a large library of reusable components need to be shared. Examples of such systems include document-centric ones like and BSCW [3] and software component libraries like CodeFinder [16]. One of the main problems we have found with many of these existing approaches is their reliance on either too simple indexing strategies, or use of very complex and restrictive formal program semantics.

Various systems have taken an integrated approach to supporting distributed software development, where one tool supports a wide range of collaborative work capabilities, or multiple tools are integrated to provide a sophisticated environment. Those adopting the former approach include Argo/UML [27], Serendipity-II/JComposer [13], MOOSE [9], and TeamWave [28]. Unfortunately many of these approaches result in either monolithic, difficult to maintain and extend environments, or limit the kinds of tools that can be added into an environment.

To overcome some of these problems with current component engineering methods, tools and collaborative work facilities, we have developed a new method for engineering component-based systems. We have extended component engineering tools we have developed to support this method, and have integrated these tools to form an integrated development environment. We have added various collaborative work facilities in a seamless manner to this environment to support distributed component engineering.

## 3. Aspect-oriented Component Engineering

We use an example component-based application and some of its constituent components to illustrate our method and tools in this paper. Figure 1 shows a screen dump from a collaborative travel itinerary planner we have developed using software components. This application provides: a tree structure editor component, in this situation used to view and edit travel itineraries; a map visualisation component, used to illustrate an itinerary route; itinerary item property dialogues; a collaborative text chat; a web browser; and several collaborative work-supporting notification and task automation agents (e.g. multi user editing, multiple cursors, change notifiers etc.). The architecture of this application consists of these components related in various ways with reusable components configured to suit this particular reuse situation.

Different components making up this system provide services for other components or end users e.g. User Interface, distribution, persistency, security, collaborative work and so on. Other components require these services and thus developers can reason about the provides/requires relationships between components that comprise an application. We have been developing "Aspect-Oriented Component Engineering" (AOCE), a new methodology for component engineering that focuses on developing characterisations of the provided and required "aspects" of components, grouping them by the systemic aspects that they relate to [12].

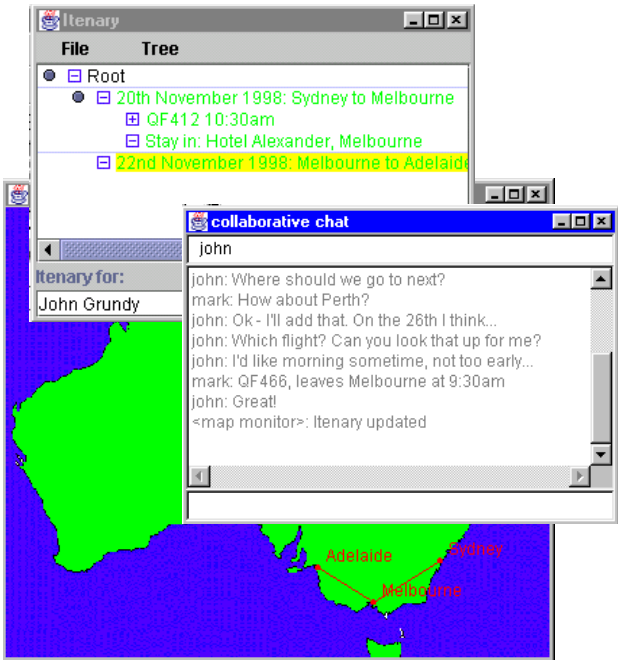


**Figure 1. Example component-based application.**

Figure 3 shows a simple example of such "horizontal slicing" of a component's capabilities. In this example, we are viewing multiple perspectives of the chat component i.e. its aspect characterisations, illustrating the different kinds of services such a component provides to other components and end users, and requires from other components. With each aspect category are grouped various "aspect details" e.g. provided panel and extendable menu, required frame, provided event generation and actioning, required event transport mechanism etc. Note some categorisations may overlap, for example the collaboration aspect also includes some of the interface and persistency-related services to support event display and serialisation/deserialisation for transport.

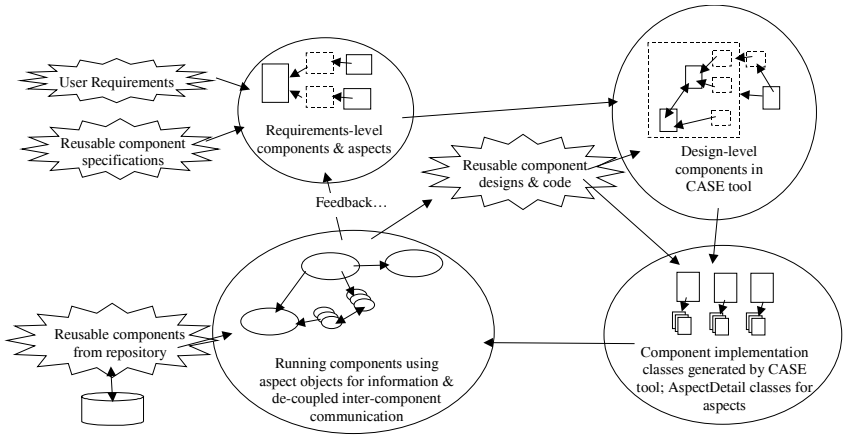


**Figure 2. Basic aspect-oriented component lifecycle.**

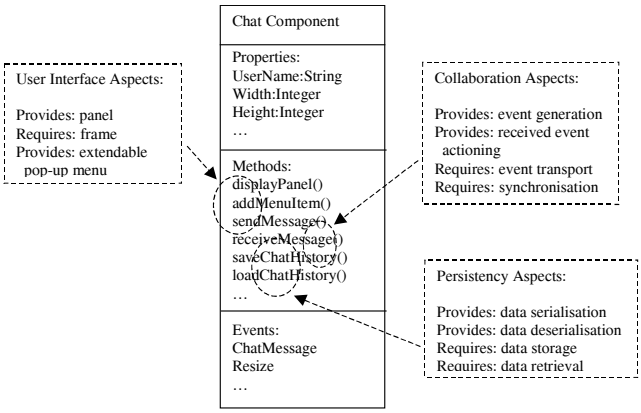


**Figure 3. Chat component's aspects.**

Each aspect detail has a set of "aspect detail properties" which further characterise it e.g. events generated before or after the component state is updated, event transport should be across WAN and support 100 event objects per second and so on. Aspect details and detail properties provide developers with a mechanism to specify the provided and required capabilities of components and allow them to specify both functional and non-functional characteristics of these capabilities.

AOCE is used throughout a component's lifecycle to describe, reason about and guide design and implementation of components. Figure 2 shows an example of the basic lifecycle of a component being engineered using AOCE. Requirements engineers develop abstract aspect characterisations of components based on user requirements. Developers may reuse existing components and aspects, reasoning about interactions with new component specifications using aspect details and properties. Refinements of these components and their aspects are developed to produce design-level system components and design-level aspects. Developers use software architecture characteristics and reusable design-level components and aspects to produce these designs.

They then have our CASE tool generate component-implementing classes that are specialised from a component architecture framework. AspectDetail class specialisations are also generated for each component to codify their aspects. Component implementers use aspect information to guide them in completing component implementations and interactions.

Some AspectDetail classes provide methods and interfaces which can be assumed by developers and used to implement highly de-coupled component interactions at run-time. Components may thus be coded to interact with each other directly, or use AspectDetail methods, Interfaces and/or patterns to support inter-component relationships. Components can also introspect the capabilities of other components at run-time, using AspectDetail methods and/or the capabilities of other components they can deduce from their publicised aspects to interact with them.

## 4. Supporting Notations

When designing software components, we have found aspects greatly assist developers in identifying component capabilities and ensuring designers carefully take into account the possibilities of reuse of components and necessary component interfaces and support for this. Aspects help designers to develop general approaches to providing aspect-based services and to supporting access by other components to discovering, using and tailoring such services. This is very important in successfully leveraging aspect-based approaches in component-based systems. Unlike aspect-oriented programming, we aim to avoid doing "code weaving" to distribute aspect-codified capabilities throughout component implementation code. Rather, we aim to have component methods implemented so that their aspect-related properties are handled in such a way that very general component provision and

requiring of aspect-related services is supported. This is crucial for COTS-based systems, where source code is not available or not changeable, and components can only reconfigure and interact with a component via its well-publicised interfaces. It also allows us to dynamically change the way aspect-related services are provided and used by components, not usually possible without recompiling aspect-oriented programming systems.

To model and implement components with aspects we have extended a component-based software architecture and its implementation framework, called JViews [13], to incorporate characterisations of component aspects, including aspect categories, aspect details and aspect detail properties. Aspects publicise component capabilities (both required and provided) according to various aspect categories, and each provided and required aspect detail has a set of property values further characterising the component capabilities. We use a UML-like notation to describe JViews components, and have augmented this notation to support aspect description.

Figure 4 shows an example of characterising a design-level component's capabilities using aspects, modelled with the JViews Architecture Description Language (ADL). This example shows the tree editor component from the collaborative travel itinerary planner application, along with some of the other components that make up this application and provide to or require from the tree editor aspect-related services. In this example, the tree editor: provides a user interface frame and configuration property sheet, may optionally require another form of structure viewer (e.g. outliner); provides event generation and actioning capabilities and requires event transport and synchronisation support; provides serialisation and deserialisation support but requires data storage and retrieval capabilities; and provides item locking and highlighting (for collaborative awareness) but requires collaboration event propagation between multiple users.

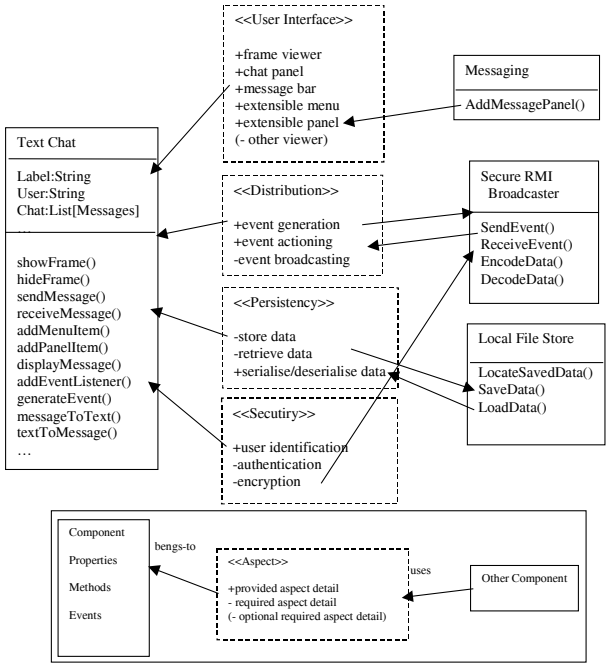


**Figure 4. Example chat component aspects.**

Aspects and aspect details may relate to one or more component object functions, and one component object function may have multiple aspect details that help characterise it. Thus designing components with aspects provides a set of multiple, partially overlapping perspectives onto the capabilities of components, with each perspective showing a particular systemic view of the component. These views also capture not only functional component capabilities, but non-functional properties as well e.g. performance, expected quality of service parameters, kind of user interface, integrity needs and so on.

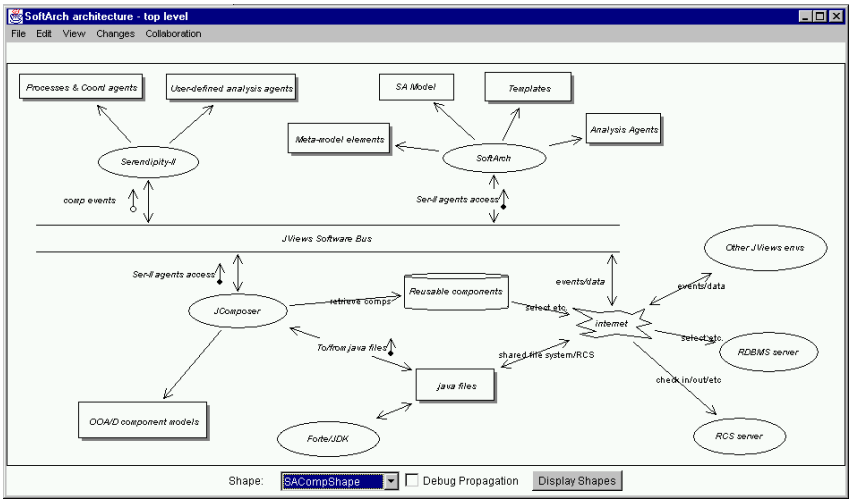


**Figure 5. The basic architecture of our component engineering environment.**

## 5. Development Tools

To support AOCE we have developed a distributed component engineering environment by integrating several tools, both our own and 3rd party. Figure 5 illustrates this environment. The main tools comprising this composite environment are outlined below:

- *Process management*. Software process management is necessary to both guide developers and track the work they have done by in a geographically and temporally distributed fashion. We use our Serendipity-II process management environment to provide distributed process modelling and enactment support, along with user-programmable and deployable task automation and tool integration agents [11].
- *Software Architecture design*. The architectures of complex component-based systems require careful design and analysis. SoftArch provides software architecture modelling and analysis facilities, including reusable software architecture templates and analysis agents [15].
- *Component design*. Software component designs must be captured, and suitable component implementations realised. We have extended our JComposer CASE tool to provide component specification, design and basic code generation support [13]. Developers describe components and their aspects in JComposer, which supports multiple views of components and aspects, allows both components and aspects to be inter-related, and provides some basic aspect-based static checking of component models. This includes checking all of a components required aspects have corresponding provided aspect details from related components and provided/required aspect detail properties are consistent.
- *Component implementation*. Component implementation and low-level debugging must be supported, along with higher-level component visualisation support. We allow developers to use their own preferred development tools, such as JDK, Forte or JBuilder. Our component visualisation tool, JVisualise, can be used to view running components, and we have added high-level dynamic architecture analysis tools to SoftArch to support dynamic system visualisation [15].
- *Component storage and retrieval*. A component repository allows multiple developers to share components in JComposer, and also supports the sharing Serendipity-II software agent components [14].
- *Other applications*. We have integrated these tools with Argo/UML [27] via XML-based import/export mechanisms, veral desktop applications (including Word™ and Netscape™), and a file server (a simple form of RCS) [13].

In order to effectively develop complex, component-based applications using our JViews architecture and framework we built the JComposer CASE tool [13]. 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 6 (1) and (2) show examples of adding aspect information to some JViews components modelled in JComposer. When design-level aspect information has been specified and basic aspect usage checks performed, JViews component implementation code is generated, including aspect codification.

Figure 6 (3) shows a SoftArch high-level view of a system architecture. SoftArch imports OOA-level component characterisations from JComposer (or Argo/UML) and supports a developer refining these to OOD-level component designs, via multiple refinement steps and views [15]. SoftArch-style models are very useful for large component-based systems design and analysis. Figure 6 (4) shows a JVisualise view visualising JViews component instances and inter-relationships in a running component-based system.

## 6. Distributed Work Support

To co-ordinate and support distributed component engineering we use the Serendipity-II process management tool, plug-in collaborative editing components, and a component repository. Figure 7 shows an example of a simple component engineering process (top view) and one of the process stages expanded (middle view) in Serendipity-II. Serendipity-II provides multiple, visual views of software processes and project plans which can be collaboratively edited. Process stages are enacted by each developer collaborating on a project, with automation of enactment as stages are completed or by software agents monitoring work done in development tools. Serendipity-II can monitor software artifact editing being carried out in tools are store these events (or synthesised summaries of multiple events) against process stages, providing a work tracking mechanism.

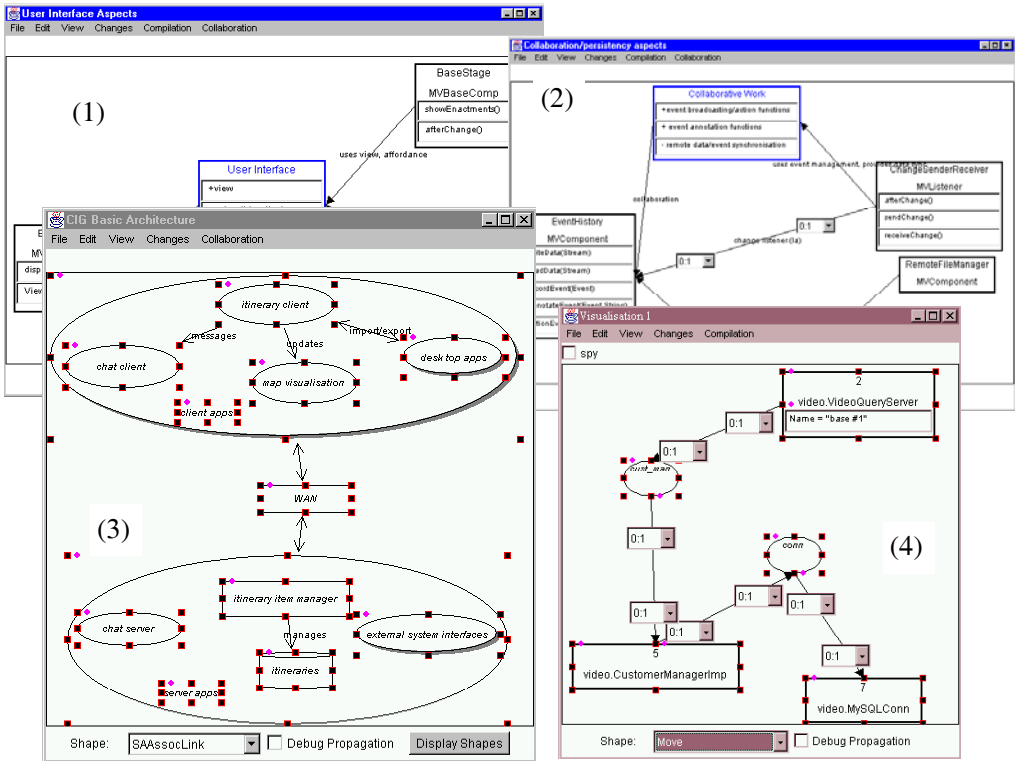


**Figure 6. JComposer, JVisualise and SoftArch in use.**

Distributed work co-ordination and tool integration is necessary when multiple developers need to be made aware of work others are doing/have been doing, or when a remote server needs to be used to centralise data or processing. The bottom view in Figure 7 shows an example of a task automation agent implementing a simple notification mechanism. In this example, when editing events made by user “john” occur while the process stage “4b. Determine Design ” is enacted, they are filtered to see if they apply to the component "CIGItenary", and if so are stored so user “mark” can review them later.

Developers collaboratively edit views in various tools in our environment, ranging from Serendipity-II process models, to SoftArch and JComposer architecture, analysis and design-level views, to JDK .java source code files. The only support we provide for source code file editing is asynchronous editing of different versions of the source code, checked out of a RCS-based distributed file management server. This has proved adequate, as developers almost always edit implementation code independently. This is particularly true for AOCE, where use of a component is ideally always via a well-defined, aspect-characterised interface specification.

Higher-level views of software development, such as process models, architectures and designs, sometimes need to be more tightly shared and collaboratively edited. Our JViews-based tools provide a flexible collaborative editing mechanism that allows developers to share views and edit them synchronously or asynchronously [11, 13]. For example, Figure 8 shows a JComposer component design view being semi-synchronously edited. Changes made by one user, ‘john’, are displayed as they are made but not immediately applied to the view (as with synchronous editing). The user whose view this is will at some stage select one or more changes and then ask JComposer to apply them to his/her view incrementally. Inconsistencies, such as syntactic or semantic errors that occur, generate error messages that are displayed in another dialogue. The user can also edit such views asynchronously, and have JComposer send them to collaborating developers at some later stage, where they can be incorporated or further discussed/refined. Whole views can also be exhanged, creating alternative versions for developers to edit and merge further changes with.

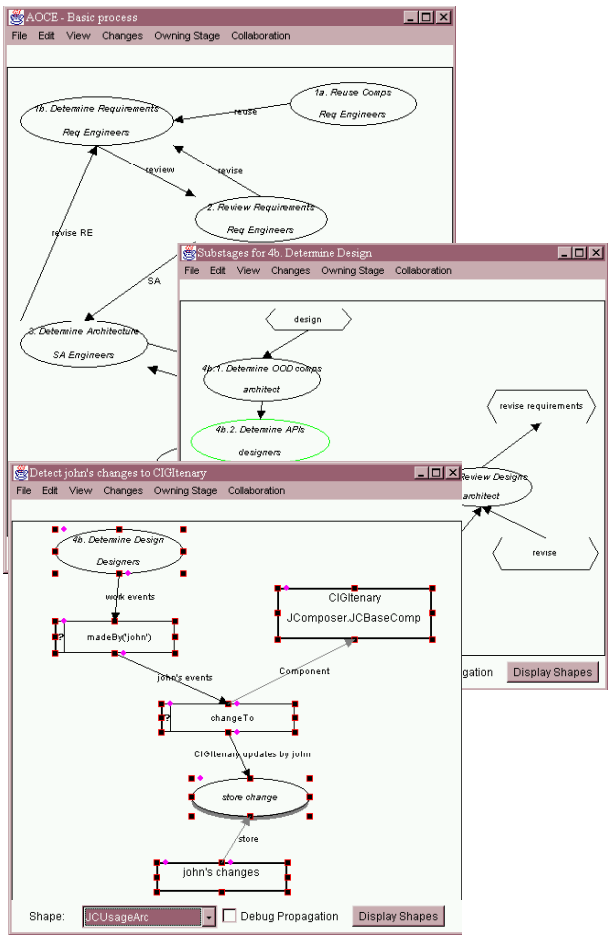


**Figure 7. Serendipity-II process views.**

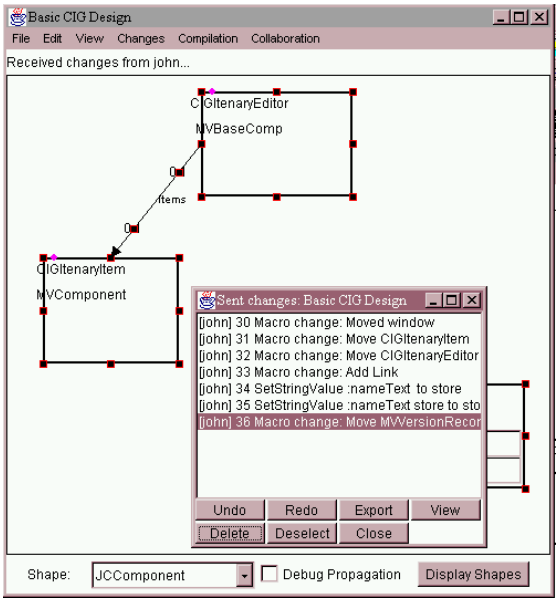


**Figure 8. Collaborative editing in JComposer.**

We have recently developed a component repository to allow distributed developers to share components, particularly commonly reused infrastructure components for building applications [14]. In order for developers to effectively share components, a common language or ontology for characterising and describing components must be used. We use AOCE component aspects to do this. Figure 9 shows an example of a developer searching our repository for a component supporting event broadcasting over the internet. The developer formulates a query using aspect, aspect detail and aspect detail properties of the component they are searching for. Retrieved components are displayed, and aspect-organised information about these components can be browsed by the developer. In this example, the developer has reused a socket-implementing collaborative editing component (actually part of the JViews infrastructure). A validation function has been run and the component is indicating it currently lacks a required relationship to another component in order to be able to function.

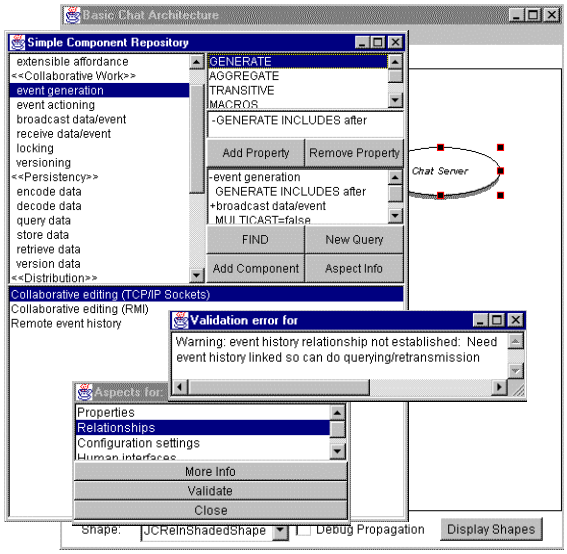


**Figure 9. Repository query and aspect details.**

## 7. Tool Architecture and Implementation

The basic approach we have taken to developing an integrated, distributed component management environment is shown in Figure 10. This illustrates the architectures of our decentralised software engineering environments, built using our JViews component-based framework. JViews provides abstractions for building tools supporting multiple views, component-based integration mechanisms, local persistency, and distributed event and data exchange.

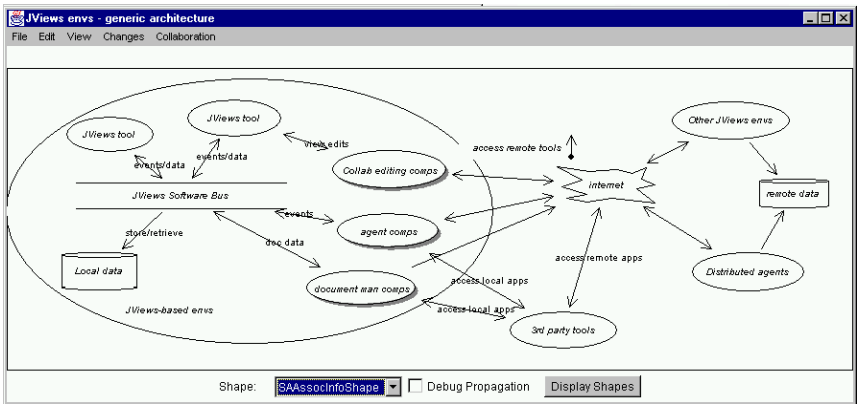


**Figure 10. Basic architecture of our decentralised environments.**

JViews-implemented tools are integrated on a developer's local machine using JViews infrastructure abstractions (basically component event and data interchange mechanisms). All data used by these tools is stored locally, with other users' tools also storing data they use locally. Any shared data is replicated on all machines, producing a decentralised architecture for the overall system. Such an architecture supports a robust software development environment, tolerant of network and machine failure, provides improved security for private work data, and scales up well to large multi-person projects [11, 13].

Distributed software development is support across any internet communications medium (LAN or WAN) by event and data exchange between users' environments. This involves tools exchanging events via JViews event distribution abstractions to reconcile updated data, notify of interesting events etc. Data can be exchanged with a version control and configuration management mechanism used to manage alternate versions.

Local and remote 3rd party tool access is support by wrapping tools (e.g. desktop applications like Word™ and Netcape™, or servers like RCS and http) with JViews components. Local and remote "software agents" provide both tool integration and task automation support. These allow the behaviour and composition of an environment to be extended and configured by developers as they require.

Our component engineering environment is supported by collaborative editing facilities plugged into our JViews-based tools, decentralised work and process co-ordination tools built and deployed in Serendipity-II, a component repository, and a distributed document (file) management tool.

We have implemented local JViews tool data persistency using the PSE Pro™ object store. This provides high-performance, robust local data management. We developed a custom event and data serialisation mechanism to support information exchange between JViews environments. Socket-based and RMI-based event communications are used, RMI more recently to assist developers in more easily building distributed environments. CORBA remote object interfaces are used to access some distributed software agents and 3rd party tools, though currently extensive use is made of JViews wrapper components to facilitate most tool integration.

## 8. Summary

We have described a methodology and support tools for engineering distributed component-based systems. Our methodology, aspect-oriented component engineering, provides a new approach to characterising component capabilities and reasoning about component interactions and provided/required services. Our JComposer, SoftArch and JVisualise tools support modelling, implementation and visualisation of complex component-based systems. We support a variety of collaborative work mechanisms, including process-based work co-ordination, collaborative editing and a shared repository of components.

We are working on a new infrastructure for building environments like the one described in this paper, to more readily facilitate their integration and configuration. We are extending AOCE to provide a richer range of aspects and aspect details for many problem domains. We are also further extending the UML to provide richer notational support for aspect-based component modelling and analysis.

## References


1. Allen, P. and Frost, S., *Component-Based Development for Enterprise Systems: Apply the Select Perspective™,* SIGS Books/Cambridge University Press, 1998.
2. Bandinelli, S. and DiNitto, E. and Fuggetta, A., Supporting cooperation in the SPADE-1 environment, *IEEE Transactions on Software Engineering,* vol. 22, no. 12, December 1996, 841-865.
3. Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system,

in *Proceedings of the 4th International WWW Conference,* Boston, MA, December 1995.

4. Brown, A. and Barn, B. Enterprise-Scale CBD: Building Complex Computer Systems from Components In *Proceedings of the 9th International Workshop on Software Technology and Engineering Practice*, 30 August - 2 September, 1999, Pittsburgh, USA, IEEE CS Press.
5. Conradi, R. Hagaseth, M., Larsen, J., Nguyen, M.N., Munch, B.P., Westby, P.H., Zhu, W. and Jaccheri, M.L., EPOS: Object Oriented Coopeartive Process Modeling, In *Software Process Modeling & Technology*, Finkelstein, A., Kramer, J. and Nuseibeh, B. Eds, Research Studies Press, 1994.
6. Dellarocas, C. The SYNTHESIS Environment for Component-Based Software Development, In *Proceedings of 8th International Workshop on Software Technology and Engineering Practice*, July 14-18 1997, London UK, IEEE CS Press.
7. Dewan, P. and Choudhary, R. A High-Level and Flexible Framework for Implementing Multiuser User Interfaces. *ACM TOIS,* vol. 10, no. 4, October 1992, ACM Press, 345-380.
8. D'Souza, D. F. and Wills, A., *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
9. Furguson, R.I., Parrington, N.F., Dunne, P., Archibald, J.M. and Thompson, J.B. MetaMOOSE – an Object-oriented Framework for the Construction of CASE Tools, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 19-32.
10. Graham, T.C.N., Morton, C.A., and Urnes, T., "ClockWorks: Visual Programming of Component-Based Software Architecture," *Journal of Visual Languages and Computing*, 175-19, July 1996..
11. 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.
12. Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In *Proceedings of the 4th IEEE Symposium on Requirements Engineering*, Limerick, Ireland, June 8-11 1999, IEEE CS Press, pp. 84-91.
13. Grundy, J.C., Hosking, J.G., and Mugridge, W.B. Constructing component-based software engineering environments: issues and experiences*, Information and Software Technology*, vol. 42, no. 2, January 2000, Elsevier.
14. Grundy, J.C. Component storage and retrieval using aspects, In *Proceedings of the 2000 Australian Computer Science Conference*, Canberra, Australia, Jan 31-Feb 3 2000, IEEE CS Press.
15. Grundy, J.C. and Hosking, J.G. High-level Static and Dynamic Visualisation of Software Architectures, In *Proceedings of 2000 IEEE Symposium on Visual Languages*, Seattle, WA, Sep 6-10 2000, IEEE CS Press.
16. Henninger, S. Supporting the Construction and Evolution of Component Repositories, In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996, IEEE CS Press, pp. 279-288.
17. Kaiser, G.E., Kaplan, S.M., and Micallef, J. "Multiuser, Distributed Language-Based Environments*," IEEE Software*, vol. 4, no. pp. 11, 58-67.
18. Kaplan, S.M., Tolone, W.J., Bogia, D.P., and Bignoli, C., "Flexible, Active Support for Collaborative Work with ConversationBuilder," in *1992 ACM Conference on Computer-Supported Cooperative Work,* ACM Press, 1992, pp. 378-385.
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., and Irwin, J. Aspect-oriented Programming, In *Proceedings of the European Conference on Object-Oriented Programming*, Finland. Springer-Verlag LNCS 1241. June 1997.
20. Magnusson, B. and Asklund, U. and Minör, S. Fine-grained Revision Control for Collaborative Software Development, In *Proceedings of the1993 ACM SIGSOFT Conference on Foundations of Software Engineering*, LA, 1993, pp. 7-10.
21. McWhirter, J.D. and Nutt, G.J., "Escalante: An Environment for the Rapid Construction of Visual Language Applications," in *Proceedings of the 1994 IEEE Symposium on Visual Languages,* IEEE CS Press, 1994.
22. Mezini, M. and Lieberherr, K. Adaptive Plug-and-Play Components for Evolutionary Software Development, OOPSLA'98, Vancouver, WA, October 1998, ACM Press, pp. 97-116.
23. Monson-Haefel, R *Enterprise JavaBeans*, O'Reilly, 1999.
24. Mowbray, T.J., Ruh, W.A. *Inside Corba : Distributed Object Standards and Applications*, Addison-Wesley, 1997.
25. Quatrani, T. *Visual Modelling With Rational Rose and UML*, Addison-Wesley, 1998.
26. Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 7, 57-66, July 1990.
27. Robbins, J. Hilbert, D.M. and Redmiles, D.F. Extending design environments to software architecture design, *Automated Software Engineering*, vol. 5, No. 3, July 1998, 261-390.
28. Roseman, M. and Greenberg, S. TeamRooms: Network Places for Collaboration. In *Proceedings of ACM CSCW 96,* pp. 325-333, 1996.
29. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
30. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
31. 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.