Preprint of paper published in International Journal of Software Engineering and Knowledge Engineering, vol. 10, No. 6, © World Scientific Publishing Co. 2000.

MULTI-PERSPECTIVE SPECIFICATION, DESIGN AND IMPLEMENTATION OF SOFTWARE COMPONENTS USING ASPECTS

JOHN GRUNDY

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

Submitted 3rd September 1999
First Revision 7th May 2000
Accepted 9th August 2000

Current approaches to component-based systems engineering tend to focus on low-level software component interface design and implementation. This often leads to the development of components whose services are hard to understand and combine, make too many assumptions about other components they can be composed with and component documentation that is too low-level. Aspect-oriented component engineering is a new methodology that uses a concept of different system capabilities ("aspects") to categorise and reason about inter-component provided and required services. It supports the identification, description and reasoning about high-level component functional and non-functional requirements grouped by different systemic aspects, and the refinement of these requirements into design-level software component service implementation aspects. Aspect information is used to help implement better component interfaces and to encode knowledge of a component's capabilities for other components, developers and end users to access. We describe and illustrate the use of aspect-oriented component engineering techniques and notations to specify, design and implement software components, report on some basic tool support, and our experiences using the approach to build some complex, component-based software systems.

Keywords: component-based development, aspect-oriented design, requirements engineering, software architectures, software components, software engineering environments

1. Introduction

As software systems and the software development process become ever more complex, developers require improved methods and technologies. Component-based systems are one example offering potential for better existing component reuse, compositional systems development, and dynamic and end user re-configuration of applications [1, 2, 3, 4]. Component-based systems compose applications from discrete, reusable, inter-related software components, which are often dynamically plugged into running applications [5, 3]. Various software architectures and implementation frameworks have been developed based on the notion of software components, including COM [6],

JavaBeans [5], and JViews [7]. Various development tools and methodologies have been developed to support component-based software construction [8, 9, 10, 7, 11, 12].

Most component-based development approaches, like traditional object-oriented analysis and design, focus on designing and implementing components that take “vertical slices” of overall system functionality, breaking systems into services grouped by data and operations on data [13, 14, 15, 16]. Most component-based techniques focus on the identification of interfaces supporting these vertical-slice, functional de-compositions [17, 6, 15], and encode low-level information about component interfaces for use at run-time [5, 6]. During development of several component-based design environments and collaborative Information Systems using these approaches we have found that they do not adequately help developers to capture, reason about and encode higher-level component capabilities [18, 19]. In particular these approaches are poor with respect to addressing issues cross-cutting component services, or the "horizontal slices" through systems. This makes component requirements analysis, specification, design, implementation and deployment difficult and components less reusable and more challenging to deploy and document.

To overcome these problems, we have been working on a new component development approach we call Aspect-Oriented Component Engineering (AOCE). AOCE focuses on identifying various horizontal slices, or “aspects”, of an overall system a component contributes (provides) services to, or services it uses (requires) from other components. Aspects are horizontal slices through a system, which typically affect many components identified by functional decomposition, of common system characteristics such as user interfaces, persistency and distribution and collaborative work. Component developers use aspects to describe different perspectives on systemic component capabilities during requirements engineering and design. Aspects also help to guide component implementation, particularly inter-component interface development, and aspect information can be encoded into component implementations for run-time use. Unlike most Aspect-oriented Programming approaches [20, 21, 22], AOCE avoids the concepts of "code weaving" and the use of run-time reflection mechanisms. Instead we focus on developing components whose cross-cutting systemic issues are carefully factored into the component interfaces so that components can be run-time re-configured and dynamically composed.

This paper motivates the need for AOCE and gives examples of using aspects during component requirements engineering, design and implementation. We begin with an overview of the concept of component aspects, using a component-based process management environment for illustration. We describe aspect-oriented component requirements engineering, and the refinement of component requirements codified by aspects into design-level aspects. Implementation of software components using design-level aspect information is described, along with various run-time uses of aspects. Tool support is briefly discussed, and we compare and contrast our approach with other component development methods and architectures. We conclude with an overview of current and possible future research directions.

2. Aspects of Software Components

We have developed several systems using software components [18, 19], one example being the Serendipity-II software process management tool, a screen dump from which is shown in Fig 1 (a). Serendipity-II provides multiple graphical and textual views of software process models, including stages (1), event connections (2), role assignments (3) and task resources. Enactment event histories (4) and shared to-do lists (5) facilitate work tracking and co-ordination. Fig 1 (b) shows some of the software components that make up Serendipity-II. Some are domain-specific, e.g. "Stage Icon", "Base Stage" and "Base Flow", and some are quite generic, e.g. "Editing History", "Collaborative Editing" and "View rel", reused in many diverse component-based applications [23].

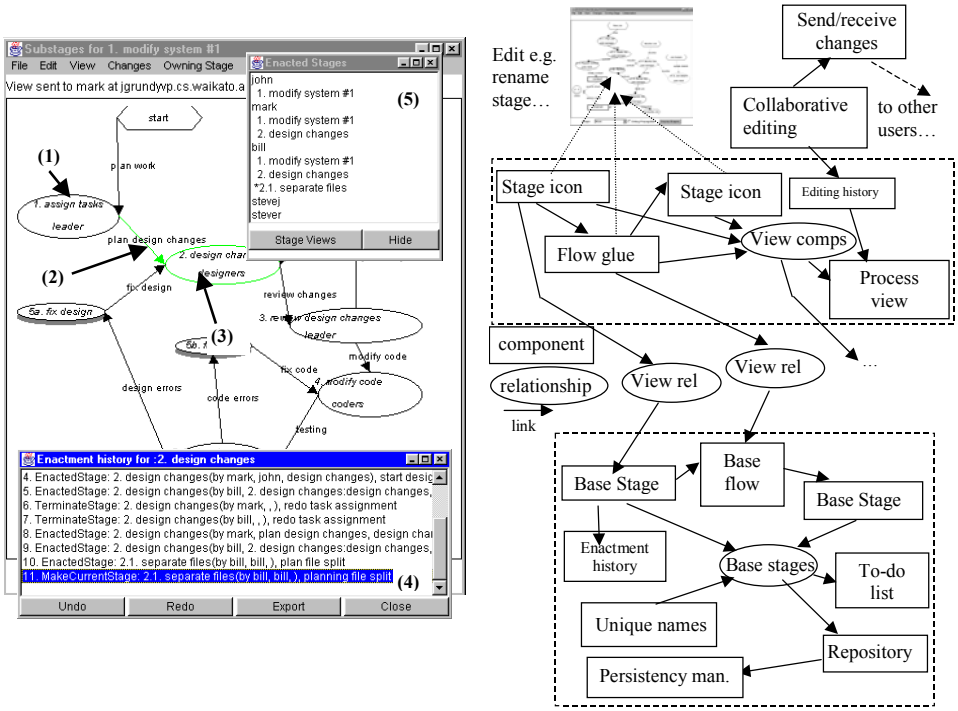

Fig 1. (a) Serendipity-II process management tool; and (b) example components.

Complex engineering issues arise when developing systems like Serendipity-II:

- Developers need to identify, describe and reason about inter-component relationships and capabilities. This includes identifying and describing the services a component requires as well as those it provides.
- Developers are often unaware of all the potential reuse situations of a component, and thus have to be very careful about assumptions made about related components.
- Components need to be appropriately configured for a particular reuse situation

- Developers may want to reuse 3rd party or commercial off-the-shelf (COTS) components, whose source code they may not have access to nor control over
- Components need to provide appropriately adaptable user interfaces, middleware capabilities and configuration capabilities, to be reused in many situations.
- Developers need to be able to refine component requirements to software component designs and implementations, ideally using a consistent metaphor for characterising component capabilities and inter-component relationships.
- Information about component capabilities and configurations needs to be available to both end users and other components at run-time, to facilitate plug and play.
- Users require appropriate support for finding, reusing and configuring components.

Most current component development methods, such as Select Perspective™ and Catalysis™ [1, 24], do not adequately support high-level description of and reasoning about component capabilities and focus on decomposing systems into "vertical slices" of functionality i.e. group functions with data using domain-specific categories. We developed the concept of component aspects to allow us to better categorise component capabilities according to a component's contribution to an overall component-based system's functions and to help organise non-functional constraints, particularly those cross-cutting components. Aspects are "horizontal slices" of a system's functionality and non-functional constraints, and include user interfaces, collaborative work facilities, persistency and distribution management, and security services. A key feature of this categorisation of component characteristics is the idea that some components *provide* certain aspect-related services for other components (or end users) to use, while other components *require* certain aspect-related services from other components.

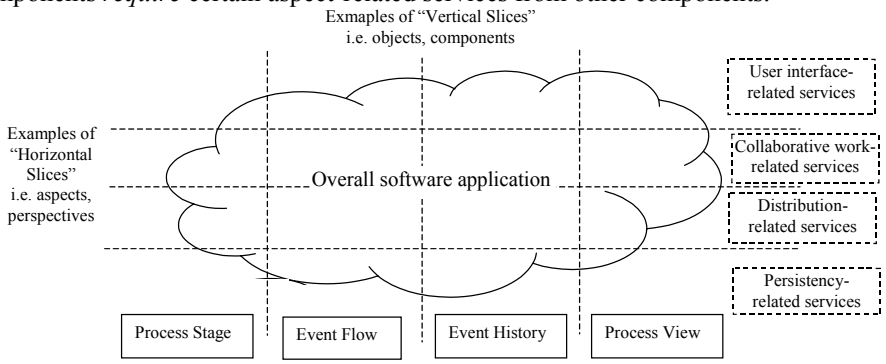

Fig. 2. General concepts of components vs. component aspects.

Fig. 2 illustrates the conceptual difference we make between software components (vertical slices of system data and functionality) [14] and component aspects (horizontal slices of system functionality and non-functional constraints) [13]. Usually each component in a system provides one or more aspect-related services for other components to use, and requires one or more aspect-related services from other

components in order to function. Identifying aspect-categorised services allows us to reason about components from various systemic perspectives cutting across the typical system vertical slicing into software components.

Each component aspect (perspective) has a number of "aspect details" that are used to more precisely describe component characteristics relating to the aspect. For example, at a requirements level we typically want to talk about general notions of data and event sources, event propagation and receiving, and concurrency control, in relation to the distribution aspects of component-based systems. At a design level, we typically want to talk about particular implementation patterns (e.g. observer, notifier) and technologies (e.g. TCP/IP sockets, CORBA, messaging systems etc).

Examples of some aspects and aspect details we have identified and found useful for applications we have developed are illustrated in Fig. 3 (a). This is not an exhaustive list and many other aspects might be appropriate in different domains e.g. real-time response, transaction processing, memory management, and so on.

Aspect	Aspect Details	Description
User interface	Views Affordances Feedback Extensible parts	Components supporting or requiring user interfaces, including extensible & composable interfaces for several components
Collaboration	Sync. editing Versioning Floor control Group awareness Messaging	Components supporting or requiring collaborative work by users
Persistency	Store/retrieve data Locate data Lock data	Components supporting or requiring data persistency management facilities
Distribution	Object Identification Method Invocation Transactions Event propagation Concurrency control	Components supporting or requiring distributed object management facilities
Security	Encoding model Key distribution Authentication Access control	Components supporting or requiring inter-component secutiry models
Configure	PEMs & Aspects Property sheet Wizard Scripting	Components supporting or requiring end user or dynamic configuration of component

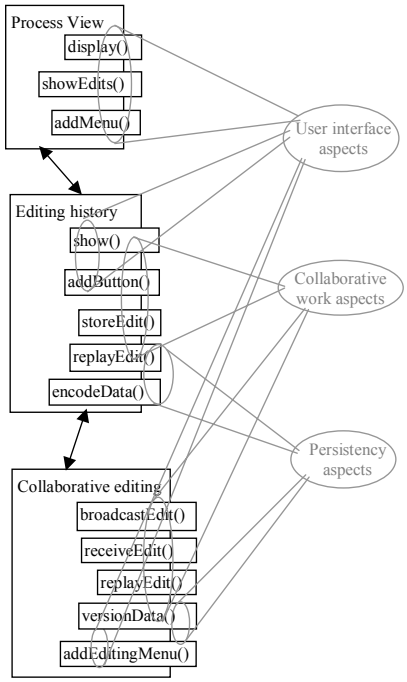

Fig. 3. (a) Some general aspects; and (b) an illustration of component aspects from Serendipity-II.

We identified these particular aspects and aspect details by carefully looking at the systemic, cross-cutting issues common to many components in our problem domains (visual design tools and Collaborative Information Systems [16]). In other problem domains, additional aspects are likely to be necessary (see [20, 40, 13, 22]). For example in real-time systems event response time, memory management and

concurrency aspects; safety-critical systems have redundancy and high assurance aspects; and security-critical systems have various additional security-related aspects. Developers need to identify the key cross-cutting concerns their application components have, develop suitable aspect details and agree on their aspects and aspect details so they can sensibly exchange component knowledge.

When reasoning about provided and required services of components we analyse these in terms of particular aspect details a component provides services relating to, or requires services related to from other components. Fig. 3 (b) illustrates how some aspects map onto some Serendipity-II component services. Our notion of an aspect is also used to capture information about non-functional constraints. For example, developers may not only wish to describe collaborative work or distribution aspects but note performance constraints, such as required network speed, maximum data transfer or transaction processing performance, security or robustness characteristics, and so on. Note some aspect categorisations may overlap e.g. the versionData() service might be considered a collaboratibve work aspect or persistency-related aspect of the Collaborative editing component. Our notion of an aspect is tolerant to this potential "overlapping of concerns", and we find it a very useful characteristic.

We have used the concept of component aspects during component requirements engineering, component design and implementation and component testing and deployment. Aspects aim to increase developers and end users knowledge about components by providing a more effective categorisation and codification mechanism for the cross-cutting of component services.

AOCE begins with component and/or system requirements engineering using aspects. Software component design refines abstract component specifications, choosing appropriate user interface, middleware and database technologies, and program design approaches and patterns. Component implementation using aspects produces deployable software components, with aspects used to guide interface implementation. Aspect information is encoded in components for run-time use by end users, developers, and other components. A key difference between AOCE and most conventional uses of OOA/D is that component requirements and designs may be reasoned about from an application perspective i.e. an overall system's requirements or design, 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. In the following sections we focus on the use of aspects for component requirements engineering, design, implementation and deployment, then compare our approach to other component development methods and technologies.

3. Aspect-oriented Component Requirements Engineering

Aspect-oriented component engineering begins by analysing a system's requirements, or one or more discrete components' requirements [16]. Typically candidate components are found from OOA diagrams, by reverse engineering software

components, or bottom-up consideration of individual, reusable components. For each component we identify, using desired component services and non-functional constraints, the aspects and aspect details for which the component provides services to or requires services from other components.

For example, consider the event history component, reused by Serendipity-II to provide view editing, processes stage enactment and collaborative editing histories. This can be identified from Serendipity-II requirements or in a bottom-up fashion as a commonly required design environment component. Event history functional requirements include event management (add, remove, annotate), history display and manipulation, multiple user sharing, and data persistency.

Fig. 4 illustrates some aspect-oriented requirements for the event history component and related components from Serendipity-II. Aspect details are categorised as being "provided" ("+" prefix) or "required" ("-" prefix).

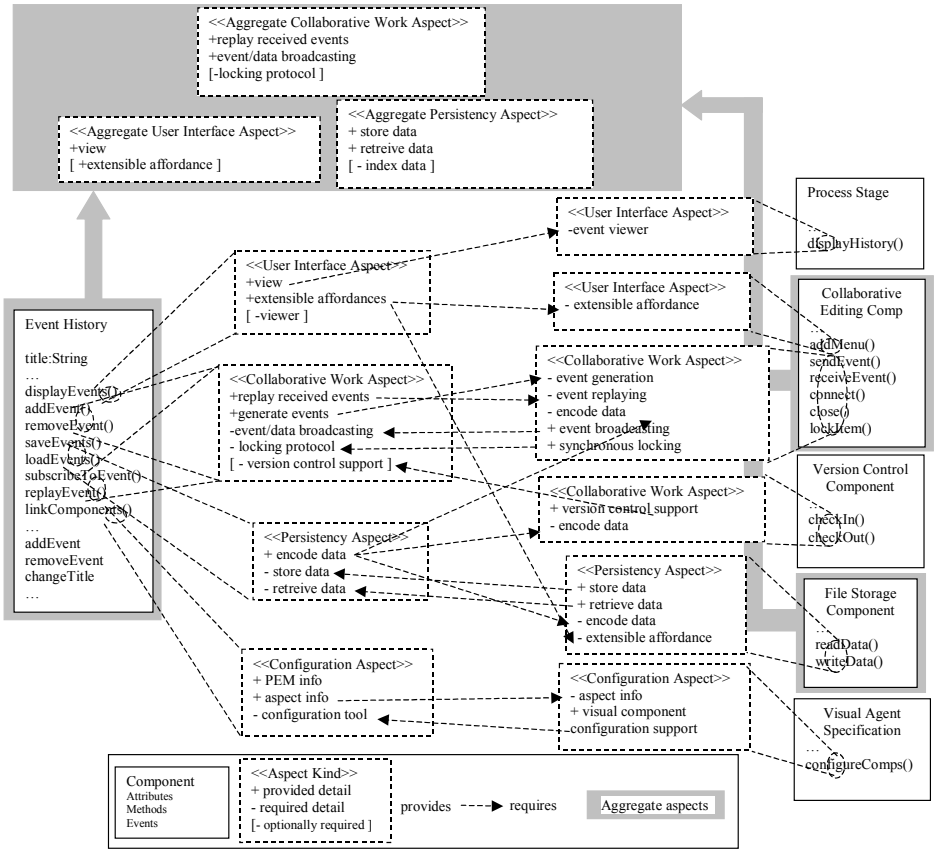

Fig. 4. Visual representation of Serendipity-II components and some of their requirements-level aspects.

Some "required" aspect details may be optional i.e. the component can still function without them being satisfied, although with some services unavailable. Serendipity-II's

requirements indicate the event history component must provide a user interface, must support collaborative viewing and editing, must be persistent, and allow configuration of history behaviour. We indicate user interface affordances must be “extensible” by other components, a need identified during Serendipity-II requirements specification, where a versioning component needs to extend event history affordances. We identified that collaborative work infrastructure should be provided by other components, to enable its reuse. Thus the event history provides basic collaborative work-related facilities, such as event editing, annotation, actioning received events and providing event listening and export facilities, but requires support for event and data broadcasting between environments, versioning facilities and data persistency from other components. Note that aspect details are kept quite general at the requirements level, and the eventual implementation strategies of these facilities is abstracted away.

Requirements (and design) level aspects can be reasoned about in groups, or “aggregate” aspects. Aggregate aspects are useful to identify, specify and reason about for groups of interrelated components. They also allow global, system-wide requirements to be captured, constraining more detailed aspects. Fig. 4 shows some aggregate aspects for the event history, collaborative work and file persistency components. The aspects of this aggregate are a constrained subset of the grouped components (no extensible affordance, no versioned history, requires indexing and so on). Developers choose whether to "show" provided aspect details in the aggregate i.e. make them accessible to other components. Required aspect details satisfied within the aggregate may be omitted, or if other components the aggregate is composed with can provide these services i.e. over-ride provision within the aggregate. Provides/requires relationships between aspect details allow developers to reason about the validity of component configurations. Consider an event history linked to a component providing event broadcasting but not data versioning. This configuration could be used but would not provide versioned event histories (acceptable in some situations but not others).

Each aspect detail has additional information encoding its functional and non-functional characteristics. These aspect detail *properties* are used to more formally describe aspects and inter-aspect relationships. A textual specification language defines components; aspects; provided and required aspect details; and detail properties with values or value constraints. Fig. **5** shows an example of some codified aspect information for event history and collaborative editing components. The event history's collaboration aspects provide EVENT_SOURCE services. Events are propagated before and after state changes to the event history (GENERATE=before, after), include events from aggregated components (AGGREGATE=true) and events generated in response to other events (TRANSITIVE=true). The event history requires EVENT_EXCHANGE services, provided in this example by a collaborative editing component. These should use the history's own event serialisation services (SERIALISATION=event_source) and must be able to propagate at least 3 editing events per second (NUM_PER_SECOND >= 3). Some properties are expressed as single or enumerated values, some as value constraints and some as values computed from other properties.

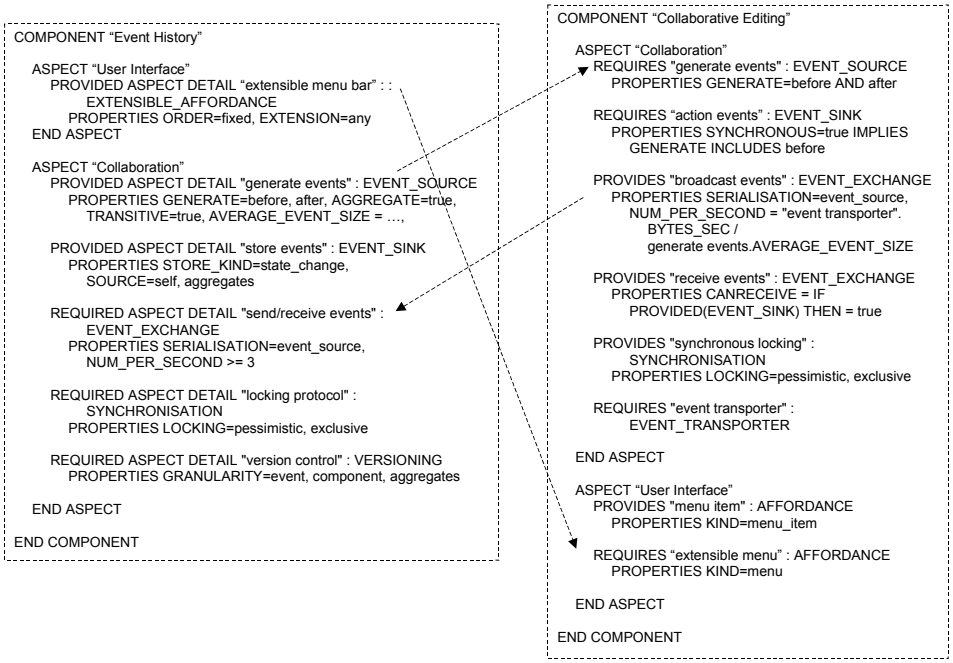

Fig. 5. Textual codification of some Serendipity-II aspect detail properties.

4. Software Component Design with Aspects

Aspect-oriented component requirements provide a focused set of functional and non-functional constraints a design can be refined from. As with refining object-oriented analysis models into object-oriented designs, requirements-level components can be refined directly to matching design-level software components, or can be split, merged or otherwise revised. Similarly, requirements-level aspect details can be refined into software component aspects that categorise design-level component services. Detailed design decisions about user interfaces, component persistency and distribution, collaboration facilities, security and transaction models, and component configuration facilities are examples of common design-level refinements. Developers also typically develop additional, design-level component specifications, such as collaboration diagrams, interface specifications and customisation policies, [25, 24].

The first step is refinement of requirements-level components to design-level software components. Developers may refine requirements-level components into e.g. user interface, data, and processing parts, allocate data and functionality accordingly, introduce various APIs (service objects) to support component implementation (e.g. databases, middleware, user interface, security etc). They may also aggregate smaller components to design implementations for larger ones. Component-based systems design tries to develop a set of interacting yet more or less stand-alone coarse-grained components versus a monolithic object-oriented design made up of objects.

Requiements-level		Design-level		
Comps	**Aspect Details**	**Comps**	**Aspect Details**	**Detail Properties**
Event History	+ viewer + extensbible affordance + generate events - event/data broadcasting - locking protocol - version control system + encode data …	Event History	+ viewer + extensible affordance + generate events - event/data broadcasting - locking protocol - version control system + encode data + classes + methods + events …	EDITABLE=true; KIND=frame; EXTENSIBLE=false KIND=button list; FUNCTIONS= { addMenuItem(),, removeMenuItem() , … } GENERATE=…; … SENDEVENTS={StoreChange, RemoveChange, …} TRANSPORT=socket; PROTOCOL=any; BYTESPERSEC >= 5000 ACQUIRE=getLock(); RELEASE=freeLock(); KIND=exclusive semaphore LOCKEVENTS={AcquireLock …} ENCODE=source; REMOTE=true ENCODING=text; ENCRYPTED=false CLASSES={EventHistoryList, EventHistoryDialogue} … …

Table 1. Refinement of component specifications to a software component designs.

As an example, consider the refinement of Serendipity-II's specifications to a set of component designs for the system. We want to produce a design made up of a group of interacting components that realises our process management environment, but where many of the components may potentially be reused via plug-and-play [19, 23]. We refine Serendipity-II specifications in a similar way to traditional OOA refinement: split them into parts which include user interface, data and processing division of responsibility; introduce service objects (APIs); and group to form units of functionality (traditionally programs, but groups of inter-operating components in our model). Note that in our component-oriented design many service object facilities (APIs) might be provided by software components (see Fig 1 (b) for part of Serendipity-II's design).

Design-level aspects refine implementation-neutral requirements into aspect details and properties that specify information relevant to selected implementation strategies software components embody. When refining requirements-level aspects to design-level developers specify aspect detail types and aspect detail properties more precisely, tieing them to component implementation design approaches. Developers can reason about implementation-specific component properties, and design-level aspects encourage implementation of de-coupled component interaction strategies.

As an example consider event history refinement, implemented by two classes. Extra aspect detail properties are introduced at design-level: the kind of viewer(s) provided and if viewer extensible; event transport mechanism, protocol and speed required; and encoding mechanism for version control storage. Some aspect detail

properties refer to design-level component services (events, methods, properties, interfaces etc), for example extensible affordance-related functions, events generated by component that should be sent to facilitate collaborative editing; locking acquire/release interfaces, function(s) and events required, and so on. The Event history incorporates both user interface, data management and persistency-related capabilities. The requirements-level Process Stage component on the other hand is refined to two distinct components - Stage Icon and Repository Stage. The framework architecture we use to build Serendipity-II makes a distinction between model and view components, connected via a reusable View Relationship component. The Stage Icon refinement includes most user interface aspects, the Repository Stage event processing, data management and data persistency aspects.

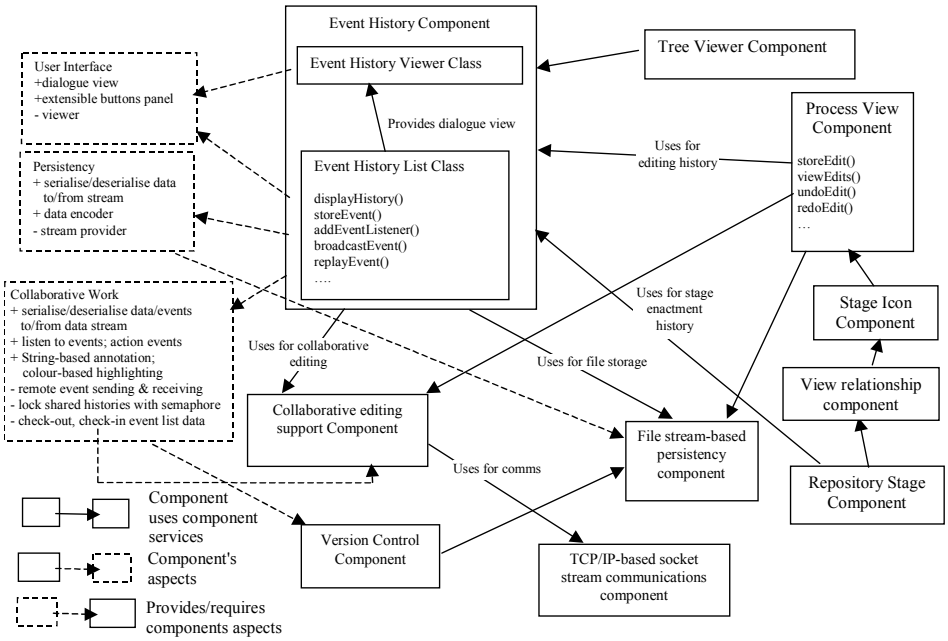

Fig. 6. Some design-level components and aspects for Serendipity-II.

Fig. 6 shows how the event history interacts with other design-level software components in Serendipity-II. We have refined the requirements components to a design which uses a synchronous/asynchronous collaborative editing support component, a TCP/IP socket-based event distribution component, a file-based version control component, and a tree-based viewer. All of these are reusable in many component-based applications. The domain-specific Serendipity-II process stage component has been refined to data management ("repository stage") and view-level ("stage icon") components. Grouping and management of these is done by repository and view components. This example of design-level software components and their aspects is the way that process view event histories are actually implemented for Serendipity-II [19].

There tends to be overlapping of aspects at design-time (and often at requirements level) in terms of the component services the aspects affect. For example, for the event history design, the persistency event data encoding uses the same implementation as collaborative work event data encoding. If constraints on the encoding provided aspect detail were changed e.g. to a binary format or to an encrypted format, related components using this provided detail, and its corresponding implementing services, may be incompatible or will not satisfy previously satisfied functional and/or non-functional requirements (e.g. if encryption algorithm slows down event transfer/storage rates below acceptable levels or binary-encoded data can't be stored by persistency component). Design-level component aspects record the services they affect, and may also record aspects whose concerns overlap with their own, allowing designers to track using these inter-perspective dependency links.

5. Component Implementation

Aspect-oriented component designs can be realised using any implementation framework for components. For example, Enterprise JavaBeans services map onto aspect characterisations reasonably well. However, we have found encoding aspect information in component implementations for run-time usage very useful. It can be used to introspect aspect-related services, provide de-coupled interaction, facilitate end-user understanding of components, and incorporate configuration validation.

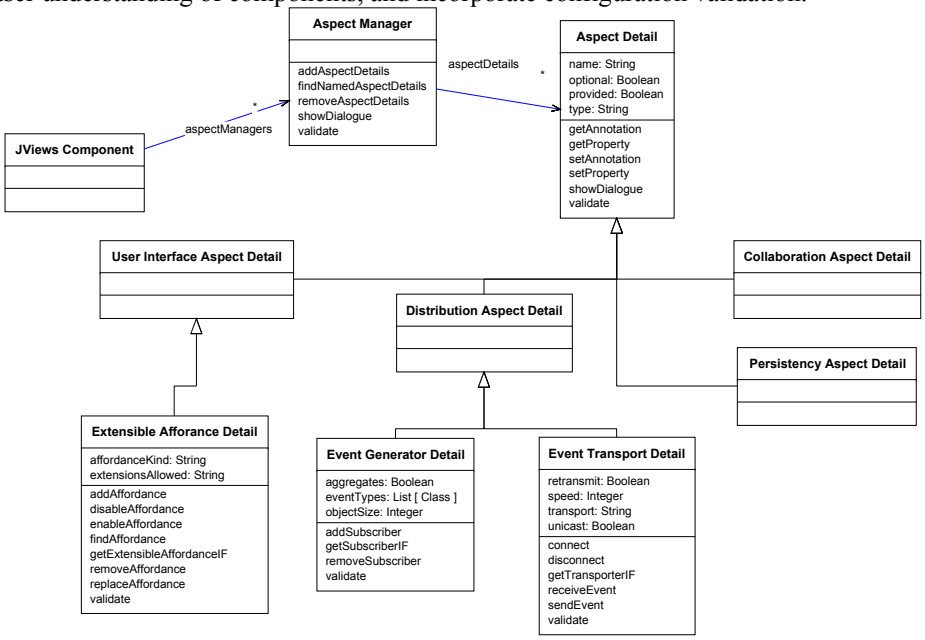

Fig. 7. Some AspectManager and AspectDetail classes from the JViews framework.

We have extended a component-based framework, JViews, to incorporate aspect information. JViews is an extension of the JavaBeans component-based framework [5],

and adds a more powerful event model, repository and view components and various reusable components for building user interfaces, middleware and data management services [23]. We added aspect codification AspectManager objects, one for each aspect category, managing various AspectDetail objects, each kind of AspectDetail class having appropriate aspect detail properties. As illustrated in Fig. 7, JViews component classes inherit from a JVComponent class that includes functions to access AspectManager and AspectDetail objects. The AspectManager classes provide functions to query, retrieve and modify their AspectDetail objects. The AspectDetail class provides generic functionality to identify (name) each aspect detail for a component, as well as common property management and annotation functions.

AspectDetail specialisations capture extra aspect detail properties and many provide aspect detail-specific component querying and manipulation functions. We have developed AspectDetail specialisations for collaborative work support, persistency, security, component configuration and transaction processing characterisation.

For example, the ExtensibleAfforanceDetail class, a UserInterface aspect detail specialisation, describes components that have extensible user interface affordances e.g. a pull-down menu or list of buttons. Its properties characterise the kind of extensible afforance, how the affordance can be extended and functions to carry out extension. Components providing extensible affordances advertise this via a ExtensibleAfforanceDetail object. Components requiring an extensible affordance advertise this, and use ExtensibleAfforanceDetail functions to discover capabilities of the provider component and dynamically extend the provider's user interface in a controlled, de-coupled fashion.

Various interfaces a component may implement can be accessed via AspectDetail functions. For example, the ExtensibleAffordanceIF interface might be implemented by a component, and another component discovers that it implements this via its ExtensibleAfforanceDetail object. The second component can access the ExtensibleAfforanceDetail functions, which know how to extend the first component's interface, or it can access the providing component's functions directly.

The EventGeneratorDetail class characterises components that provide or require event generation capabilities. It provides properties to characterise event generation as well as functions to establish and remove subscribers to events. The EventTransportDetail class describes distribution mechanisms for events and functions to carry out sending and receiving of events. AspectDetail objects may include validation functions that can be called at run-time to check components are correctly combined i.e. their aspect details and properties are sufficient to allow them to operate.

Programmers may provide functions in components that implement aspect-related services which are invoked directly, but where possible try to avoid this to minimise component coupling. AspectDetail methods allow JViews components to communicate in a generalised, de-coupled manner, producing far more reusable components.

6. Run-time Use of Aspects

AspectDetail objects are created when needed by other components. They can be used to introspect a component's capabilities at run-time, to provide a de-coupled access point for invoking functions of a component that implement aspect-related services, or be used to re-configure or validate a component.

Fig. 8 shows two examples of AspectDetail object usage in Serendipity-II. Fig. 8(a) shows a persistency management component, which needs to add extra affordances to the event history's user interface. The persistency management component accesses the event history's user interface manager (1) to obtain an extensible affordance aspect detail object. It then invokes the addAffordance() function (2), and addAffordance() calls appropriate functions implemented by the event history (3), and returns the new affordance objects. In Serendipity-II, the persistency management component extends the event history's buttons list to add "Export" and "Import" buttons, for event history saving and loading. If the event history only allowed a menu bar to be extended, it would add e.g. Export and Import menu item affordances. The persistency component knows nothing about the event history component and only interacts with it via the functions provided by the ExtensibleAfforanceAspectDetail object.

Fig. 8(b) shows a collaborative editing component using a distribution aspect manager (1) to discover the event generation and subscription interface supported by the event history (2), using this interface to subscribe to editing events (3, 4). When the collaborative editing component receives events (5), it sends these to another user's collaborative editing component via an event transport component (6, 7), illustrating a transitively provided aspect detail. Received events (8) are sent to the event history (9).

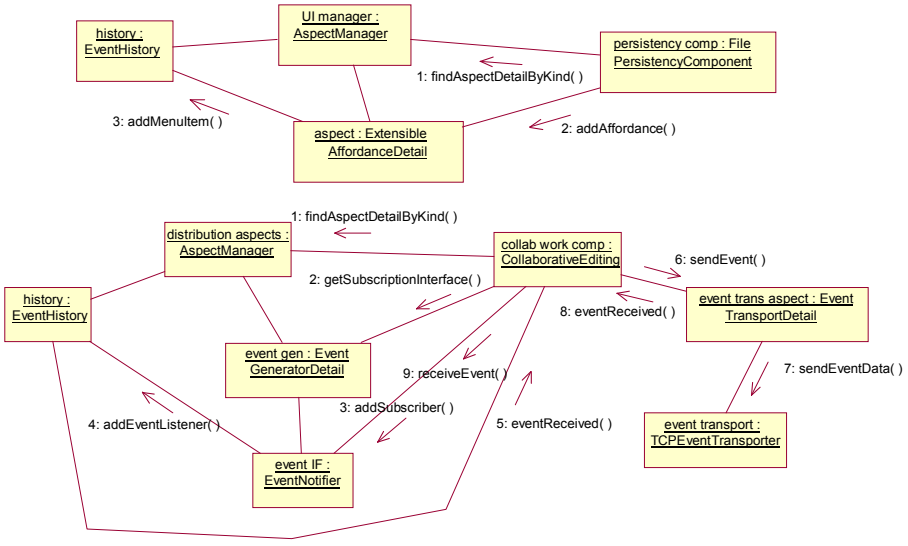

Fig. 8. (a) User interface extension; and (b) event subscription service access via AspectDetail objects.

End user support for accessing component aspect information is provided by tools that query components for their AspectInfo information. This allows developers and end users to use high-level, categorised knowledge of component capabilities. The AspectInfo information about component capabilities is also usable for indexing and locating components in a repository. Fig. 9 shows a simple example of end-user use of aspect information in Serendipity-II. The end user is building a simple process notification agent by reusing and connecting component representations in a visual agent specification tool [19]. The user can view component aspect information, and can request validation functions associated with aspects and aspect details be run to check the current component configuration is sensible and meets encoded aspect constraints. An example is shown of an end user locating and reusing the collaborative editing component from a component repository [26]. A query requested a component that provided a collaborative work aspect and an event broadcasting aspect detail.

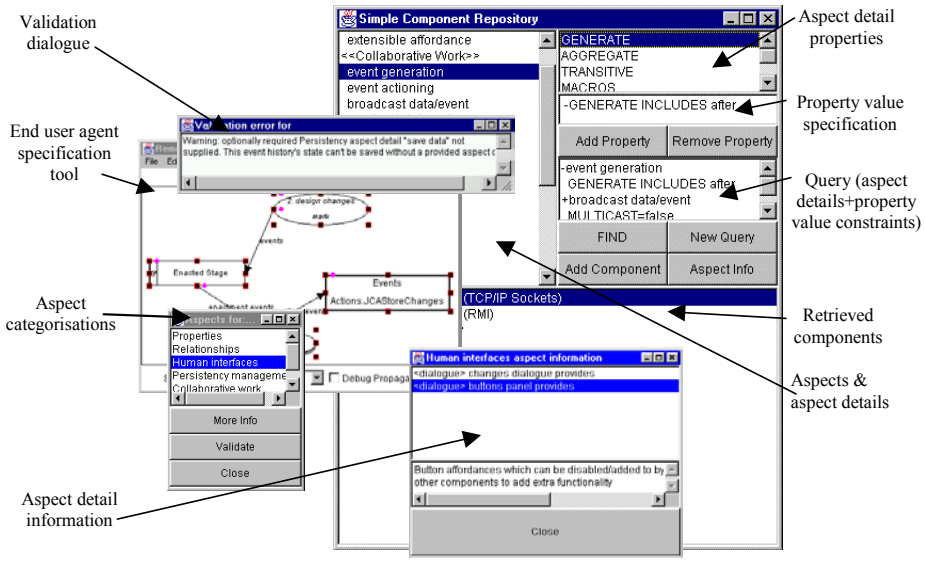

Fig. 9. End user access to aspect information (left) and component retrieval using aspects (right).

7. Development Tool Support

To support the aspect-oriented component engineering methodology we have extended a CASE tool for JViews, called JComposer [7], to support component aspects for requirements engineering, software component design and component implementation. JComposer provides multiple views of component-based software systems using the JViews ADL. It supports collaborative editing of these views and includes sophisticated inconsistency management support. We added additional constructs to JComposer to allow developers to describe component aspects, aspect details, detail properties and inter-component aspect relationships [16]. Requirements-level aspects and design-level

aspects can both be represented in JComposer, linked by simple refinement relationships. Checks can be run to ensure provided and required aspect details for related components are consistent. JComposer supports the generation of JViews component implementations, and we extended the tool to support the generation of AspectInfo class creation. Currently only basic support is provided for specifying component interfaces or use of design patterns in conjunction with aspect specifications.

8. Discussion

Current object-oriented and component-oriented development methods, such as the UML™ [10], Select Perspective™ [1], COMO [25], Enterprise-scale CBD [8], and Catalysis™ [24], tend primarily to focus on functional decomposition of requirements into objects and/or components [13, 15]. At design-level they focus almost exclusively on detailed component interface design and service implementation. We originally developed Serendipity-II and several other component-based systems using this kind of approach [19, 23], finding several problems with such approaches that other researchers and practitioners have also identified [4, 8, 2, 18, 25, 27, 17, 28]. The main problem is their tendency to produce components with capabilities and interfaces that are insufficiently adaptable. Other problems include lack of suitable notations to express component-oriented requirements and designs, lack of requirements and design processes and abstractions for current component implementation technologies, and difficulties in both developers and end users understanding components. The later is important in systems where users themselves need to extend their environment. As often no general framework is used to capture and reason about component requirements, or standardise interfaces, it is often hard to get third-party components to interact.

Some recent approaches take into account diverse component interface requirements [17] or system-wide properties [28], although they still focus on low-level component interface characteristics. Some extensions to the UML to express components include Catalysis, Enterprise CBD and COMO, [8, 24, 25, 27]. These still lack adequate structured characterisation of components, particularly their provided and required interfaces and non-functional properties. In contrast, component aspects provide a framework for multi-perspective specification, assisting developers to codify systemic characteristics, and they assist development of highly reusable, dynamically reconfigurable components. The need for reusable components that can be "trusted" to perform in appropriate ways in diverse situations has become apparent [29], and aspects with detailed property specifications and run-time validation begin to address this.

Current component technologies, such as Java Beans [5], CORBA C-IDL [30], and COM+ [6], and support tools, such as Visual Javascript [11], SYNTHSIS [9], and [12], focus on low-level component capabilities. The advertising of component capabilities using BeanInfo classes, C-IDL interfaces and type libraries does not lend itself to capturing high level knowledge about component capabilities. This adversely affects other components and end users ability to understand and appropriately use a component's facilities. Our higher-level aspect information greatly assists end users

understand reused component functionality. Enterprise JavaBeans containers [31] provide a similar capability to our JViews framework-level aspects for abstracting detailed systemic service provision from components. However, they restrict components to a containment model in order to achieve this, whereas component interaction and dynamic reconfiguration is more flexible but equally powerful.

A major aim of component-based systems is support for end user configuration of software [11, 12]. Various component-based systems support this [19, 12], as well as agent-based, workflow, adaptive user interface and end user computing systems [32, 33, 34, 35]. Most of these systems need third party agents to communicate knowledge [32, 33, 34], focusing on domain-specific data and functions, limiting the sharing of systemic user interface, collaboration and distribution mechanisms. Most component configuration tools utilise visual drag-and-drop metaphors, iconic component inter-connection or scripting languages [11, 12]. All require end users to be aware of component capabilities and what are “correct” configurations, but most tools and component architectures do not adequately capture and present such information. Aspects give end users a higher-level view of component capabilities, inter-component relationships and provided/required services.

Various techniques capture knowledge about software components, including IBROW [36], JBCDL [37], CDM [27], and [38]. IBROW uses multiple ontologies to describe problem solver components and adapatation of components at task and domain model levels. Oussalah and Messadia [38] also use task/process-solving method/domain-based descriptions. These go somewhat beyond aspects in addressing task and problem-solving issues, but do not address the kinds of cross-cutting functional and non-functional specifications as do component aspects. CDM and JBCDL use hierarchical categorisation, vertically grouping components based on component purpose, which is much more limited than cross-cutting systemic component characteristics. We have used aspects to index and support retrieval of components [26] and have found aspects form a better ontology for querying components.

Aspect-oriented programming (AOP) [13, 21, 20, 22] and adaptive programming [14, 39] are becoming popular approaches to handling cross-cutting concerns for object-based systems. To our knowledge aspects haven’t been directly applied to software component implementation aside from in our work. AOP [13, 21] uses a notion of systemic aspects of a system to "weave" code managing e.g. data persistency and object distribution [40, 13], with aspects codified independently to program classes. A key difference between AOP and AOCE is the concept of components providing services for one or more such systemic aspects and requiring one or more services from other components. Interfaces are designed to avoid AOP-style code weaving as source code may not be available for third party Commercial Off the Shelf (COTS) components, and run-time reconfiguration of component-based systems necessitates components being able to dynamically change interactions with other components’ aspect-based services.

Reflective techniques can avoid compile-time weaving [22, 41], though at the cost of expensive performance overheads and (currently) lack of design abstractions. Some

design-level approaches to codifying aspects have been developed [20, 42] though these typically adopt standardised UML-style notations which are textually annotated. Some approaches use "adaptive components" or "hyperslices" to isolate cross-cutting concerns and facilitate de-coupled component interaction [14, 15]. Such implementation strategies are compatible with our own using extended JViews components, though we have addressed a much wider range of aspects and aspect details, and aspect-oriented component requirements and design approaches provide higher-level abstractions.

We have used aspect-oriented component engineering to successfully reengineer a range of software components and component-based systems, including Serendipity-II and JComposer, and to develop new, reusable components for persistency management, collaborative work support and component distribution. Re-engineering Serendipity-II and JViews components using aspects has produced significantly better characterised component requirements, and more easily reused and reconfigured components. The main advantages aspect-oriented component engineering provides include the extra richness of multiple perspectives onto components, better structuring of component requirements and designs, encouraging implementation of better dynamic configuration and de-coupled component interaction, and run-time access to detailed component knowledge. Aspect-based perspectives give developers a set of alternative, richer viewpoints on component capabilities, and allow developers to document their components more completely. During design and implementation, aspects encourage more flexible coupling, dynamic configuration and dynamic deployment strategies. Aspect codification provides a far more powerful introspection and generic coupling mechanism, components can be indexed using their aspects, and aspect information even presented to end users.

Component aspects introduce added complexity, requiring developers think about their components from various perspectives, specify provided and required aspect details, and reason about component interaction from each perspective. A trade-off has to be found between this extra effort and AOCE benefits. Our experience indicates that for complex systems, or even single components that have several systemic aspects, AOCE is worth this extra effort with enhanced reusability, reconfigurability and understandability outweighing extra specification and reasoning effort. Problems identifying suitable aspects, choosing incomplete aspects and the lack of aspect support in current tools may mean the technique is less effective.

We are extending our set of component aspects, aspect details and particularly aspect detail properties and property constraints, allowing more formal reasoning about inter-component relationships and improved indexing and retrieval. We are extending JViews and its tools to support such formal specification and checking, and to include better support for aggregate aspect representation. We are improving JComposer's aspect generation capabilities and use of a wider range of UML modelling diagrams with aspect extensions. Use of Perceval [20] to codify aspects in an implementation-independent way and generate different component implementations is being

investigated. User studies of our requirements and design techniques are in progress using simplified extensions to the UML.

9. Summary

Several key challenges in building complex component-based systems include: the engineering of requirements for individual and groups of reusable components; refinement of requirements into software component designs; correct composition of components at compile- and run-time; and run-time access to component capabilities. Aspect-oriented component engineering addresses these by providing a new framework for describing and reasoning about component capabilities from multiple perspectives. Requirements engineering with aspects provides improved documenting of and reasoning about component functional and non-functional requirements. Requirements can be naturally refined to design-level aspects that categorise design decisions about component services and aid developers in choosing generic inter-component relationship implementations. Aspect information in component implementations allows developers, end users and other components to access high-level knowledge about a component's capabilities, and to perform basic configuration validity checks. De-coupled component interaction enhances component reusability and configurability. Tool support for aspect-oriented component engineering includes requirements to implementation use of component aspects in the JComposer CASE tool, providing run-time access to component aspect information, and an aspect-based component repository. Our experiences with aspect-oriented component engineering have been generally very positive, and a number of promising research directions exist.

Acknowledgments

Support for parts of this research from the New Zealand Public Good Science Fund and the many helpful comments of the anonymous reviewers are gratefully acknowledged.

References

1. P. Allen and S. Frost, Component-Based Development for Enterprise Systems: Apply the Select Perspective™, SIGS Books/Cambridge University Press, 1998.
2. A.W. Brown and K.C. Wallnau, Current state of Component Based Software Development, IEEE Software (Sept/Oct 1998), pp. 37-46.
3. C.A. Szyperski, Component Software: Beyond OO Programming, Addison-Wesley, 1997.
4. A.W. Brown and K.C. Wallnau, Engineering of component-based systems, in Proc. of the 2nd Int. Conf. on Engineering of Complex Computer Systems, Montreal, Canada (Oct 1996).
5. J. O'Neil and H. Schildt, Java Beans Programming from the Ground Up, Osborne McGraw-Hill, 1998.
6. R. Sessions, COM and DCOM: Microsoft's vision for distributed objects, Wiley, 1998.
7. J.C. Grundy, W.B. Mugridge and J.G. Hosking, Static and dynamic visualisation of component-based software architectures, in Proc. of 10th Int. Conf. on Software Engineering and Knowledge Engineering, San Francisco (June 18-20 1998), KSI Press.

8. A. Brown and B. Barn, Enterprise-Scale CBD: Building Complex Computer Systems from Components, in Proc. of the 9th Int. Workshop on Software Technology and Engineering Practice, Pittsburgh (30 August - 2 September, 1999), USA, IEEE CS Press.
9. C. Dellarocas, The SYNTHESIS Environment for Component-Based Software Development, in Proc. of 8th Int. Workshop on Software Technology and Engineering Practice, London UK (, July 14-18 1997), IEEE CS Press.
10. M. Fowler, The UML Distilled, Addison-Wesley, 1999.
11. Netscape Communications Inc, Visual Javascript™, 1998, http://www.netscape.com/.
12. B. Wagner, I. Sluijmers, D. Eichelberg and P. Ackerman, Black-box Reuse within Frameworks Based on Visual Programming, in Proceedings of the. 1st Component Users Conf., Munich (July 14-18 1996), SIGS Books.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, Aspect-oriented Programming, in Proc. of the 1997 European Conf. on Object-Oriented Programming, Finland (June 1997), Springer-Verlag, LNCS 124.
14. M. Mezini and K. Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development, in Proc. of OOPSLA'98, Vancouver, WA (October 1998), ACM Press, pp. 97-116.
15. P. Tarr, H. Ossher, H., W. Harrison and S.M. Sutton, Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns, in Proc. of the Int. Conf. on Software Engineering, Los Angeles (May, 1999), IEEE CS Press.
16. J.C. Grundy, Aspect-oriented requirements engineering for component-based software systems, in Proc. of 4th IEEE Int. Symp. on Requirements Engineering, Limmerick, Ireland (June 7-11 1999), IEEE CS Press.
17. A. Rakotonirainy and A. Bond, A Simple Architecture Description Model, in Proc. of TOOLS Pacific'98, Melbourne, Australia (Nov 24-26, 1998), IEEE CS Press.
18. J.C. Grundy, W.B. Mugridge, J.G. Hosking and M.D. Apperley, Tool integration, collaborative work and user interaction issues in component-based software architectures, in Proc. of TOOLS Pacific '98, Melbourne, Australia (24-26 November 1998), IEEE CS Press.
19. J.C. Grundy, J.G. Hosking, W.B. Mugridge and M.D. Apperley, An architecture for decentralised process modelling and enactment, IEEE Internet Computing **2**, (September/October 1998), IEEE CS Press.
20. F.A. Ariniegas, Introduction to Perceval: Aspect-oriented Design using XML Schema and Groves, in Proc. of the 5th Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vagas (June 26-29 2000), CSREA Press.
21. G. Kiczales and C. Lopes, Recent developments in AspectJ, In Proc. of the ECOOP'98 Workshop on Aspect-oriented Programming, Brussels, Belgium (July 1998).
22. J.L. Pryor and N.A. Bastan, Java Meta-level Architecture for the Dynamic Handling of Aspects, In Proc. of the 5th Int. Conf. on Parallel and Distributed Processing Techniques and Applications, Las Vagas (June 26-29 2000), CSREA Press.
23. J.C. Grundy, J.G. Hosking and W.B. Mugridge, Constructing component-based software engineering environments: issues and experiences, Information and Software Technology **42**, (January 2000), Elsevier.
24. D. F. D'Souza and A. Wills, Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998.

25. S.D. Lee, Y.J. Yang, F.S. Cho, S.D. Kim and S.Y. Rhew, COMO: a UML-based component development methodology, in Proc. of the Sixth Asia-Pacific Software Engineering Conf., Takamatsu, Japan (7-10 Dec. 1999), IEEE CS Press, pp. 54-61.
26. Grundy, J.C. Storage and retrieval of Software Components using Aspects, in Proc. of the 2000 Australasian Computer Science Conference, Canberra, Australia (Jan 30-Feb 3 2000), IEEE CS Press, pp 95-103.
27. R. Meling, E.J. Montgomery, P. Sudha Ponnusamy, E.B. Wong and D. Mehandjiska, Storing and retrieving software components: a component description manager, in Proc. of the 2000 Australian Software Engineering Conf., Canberra, Australia (April 2000), pp. 107 –117.
28. C.A. Szyperski and R.J. Vernik, Establishing system-wide properties of component-based systems: a case for tiered component frameworks, in Proc. of the OMG/DARPA Workshop on Compositional Software Architecture, Monterey, California (Jan 6-8 1998).
29. B. Meyer, C. Mingins and H. Schmidt, Providing Trusted Components to the Industry, IEEE Computer (May 1998), pp. 104-15.
30. T.J. Mowbray and W.A. Ruh, Inside Corba, Addison-Wesley, 1997.
31. R. Monson-Haefel, Enterprise JavaBeans, O'Reilly, 1999.
32. T. Finin, Y. Labrou and J. Mayfield, KQML as an agent communication language, Software Agents, MIT Press, 1997.
33. C. Fernström, ProcessWEAVER: Adding process support to UNIX, in 2nd Int. Conf. on the Software Process, Germany (Feb. 1993), IEEE CS Press, pp. 12-26.
34. G. Grunst, R. Oppermann and C.G. Thomas, Adaptive and adaptable systems, in Hoschka, P. (ed.): Computers As Assistants - A New Generation of Support Systems. Hillsdale: Lawrence Erlbaum Associates (1996), 29-46.
35. S. Rivard and S.L. Huff, Factors of successes for end user computing, Communications of the ACM **31**, (1988), 552-561.
36. E. Motta, D. Fensel, M. Gaspari and R. Benjamins, Specifications of Knowledge Components for Reuse, in Proc. of The 11th Int. Conf. on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany (June 1999), KSI Press, pp. 36-43.
37. W. Qiong, C. Jichuan, M. Hong, and Y. Fuqing, JBCDL: an object-oriented component description language, Proc. of the 24th Conf. on Technology of Object-Oriented Languages, (September 1997), IEEE CS Press, pp. 198 – 205.
38. M. Oussalah and K. Messaadia, An all-reuse methodology for KBS components library, in Proc. of The 11th Int. Conf. on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany (June 16-19 1999), KSI Press, pp. 187-191.
39. M. Mezini, L. Seiter and K. Lieberherr, Component Integration with Pluggable Composite Adapters. Software Architectures and Component Technology (M. Aksit, Ed), Kluwer, 2000.
40. K. Bollert, On Weaving Aspects, in Proc. of the ECOOP'99 Workshop on Aspect-oriented Programming, Lisbon, Portugal (June 1999).
41. I. Welch and R. Stoud, Load-time application of aspects to COTS, in Proc. of the ECOOP'99 Workshop on Aspect-oriented Programming, Lisbon, Portugal (June 1999).
42. W.N. Ho, F. Pennaneach, J.M. Jezequel and N. Plouzeau, Aspect-Oriented Design with the UML, in Proc. of the ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, Limerick, Ireland (June 6 2000).