# Aspect-oriented Requirements Engineering for Component-based Software Systems

John Grundy

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

## Abstract

*Developing requirements for software components, and ensuring these requirements are met by component designs, is very challenging, as very often application domain and stakeholders are not fully known during component development. We introduce a new methodology, aspect-oriented component engineering, that addresses some difficult issues of component requirements engineering by analysing and characterising components based on different aspects of the overall application a component addresses. We give an overview of the aspect-oriented component requirements engineering process, focus on component requirements analysis, specification and reasoning, and briefly discuss tool support.*

## 1. Introduction

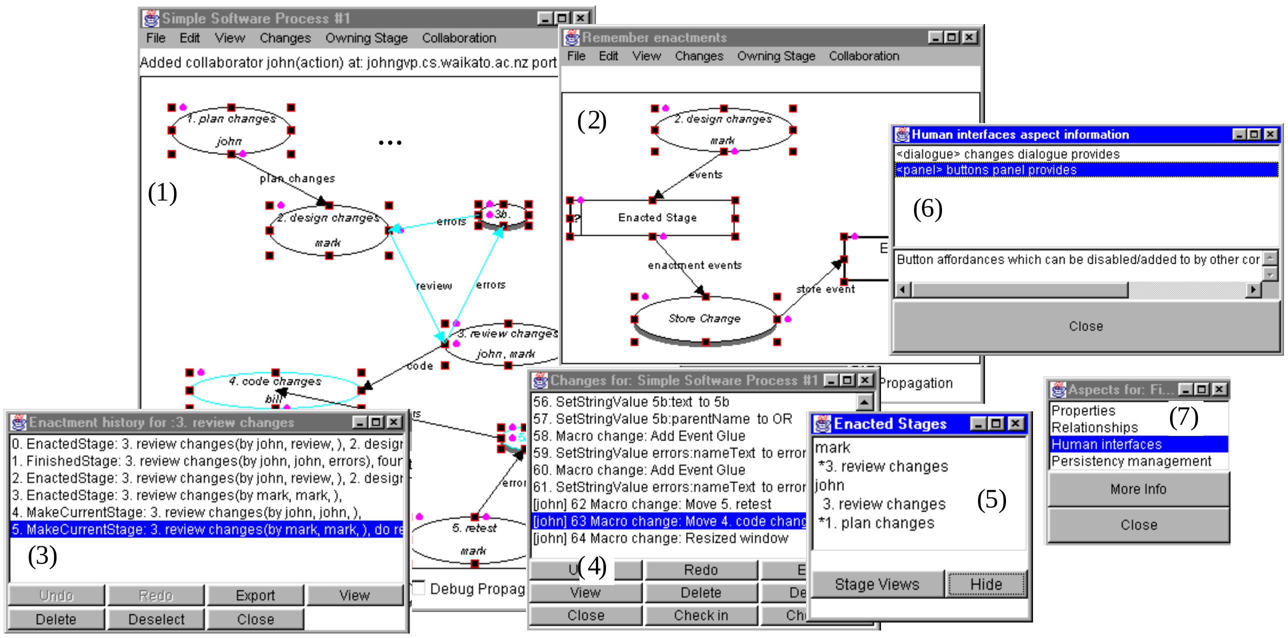
As software systems become ever more complex, developers use new technologies to help manage development. Component-based systems are one example offering potential for better existing or third party component reuse, compositional systems development, and dynamic and end user reconfiguration of applications [1, 16]. Component-based systems build applications from discrete, inter-related software components, often dynamically plugged into running applications and reconfigured by end users or other components [10, 16].

While some processes, notations and tools used for traditional Requirements Engineering [12] are useful for component development, we have found deficiencies during development of component-based design tools [2, 3, 4]. Stakeholders are often not clearly identifiable when analysing component requirements, and include end users, developers, and other components. Components typically provide and require services to and from end users and other components, which can usually be classified by different systemic aspects of an application. Traditional requirements capture techniques don't usually achieve this for individual components. The "requires" and "provides" relationships are captured by notations like Object-Oriented Analysis (OOA), but with insufficient detail. A key aim of software components is to allow components to be interchangeable, but traditional analysis techniques don't adequately identify and describe generic interfaces for extensible user interfaces, persistency, distribution and collaborative work. Lastly, a suitable codification of requirements is needed by end users and other components at run-time. We have developed *aspect-oriented component engineering* using the notion of "aspects" of a system (e.g. user interface, persistency and distribution, user configuration, collaborative work), for which components provide or require services. Aspects help identify, categorise and reason about component requirements.

Section 2 motivates our work using an example application and deficiencies with traditional RE methods and existing component-based methods and tools. Sections 3 to 5 overview the concepts and process of aspect-oriented component engineering, illustrating requirements specification for our example application and discuss reasoning with aspect-oriented requirements. Section 6 describes how aspect-oriented requirements are used during component design, implementation and deployment, and discusses tool support for aspect-oriented component engineering. We conclude with a summary of contributions and overview of future research directions.

## 2. Motivation

Our need for improved component requirements engineering grew from experiences developing multiple view, multiple user design environments. An example of such a system, Serendipity-II, is shown in Figure 1 [2].



Added collaborator john(action) at: johngvp.cs.waikato.ac.nz port

0. EnactedStage: 3. review changes(by john, review, ), 2. desigr
1. FinishedStage: 3. review changes(by john, john, errors), four
2. EnactedStage: 3. review changes(by john, review, ), 2. desigr
3. EnactedStage: 3. review changes(by mark, mark, ),
4. MakeCurrentStage: 3. review changes(by john, john, ),
5. MakeCurrentStage: 3. review changes(by mark, mark, ), do re

56. SetStringValue 5b:text to 5b
57. SetStringValue 5b:parentName to OR
58. Macro change: Add Event Glue
59. SetStringValue errors:nameText to error
60. Macro change: Add Event Glue
61. SetStringValue errors:nameText to error
[john] 62 Macro change: Move 5. retest
[john] 63 Macro change: Move 4. code chang
[john] 64 Macro change: Resized window

mark
*3. review changes
john
3. review changes
*1. plan changes

<dialogue> changes dialogue provides
<panel> buttons panel provides

Button affordances which can be disabled/added to by other cor

Properties
Relationships
Human interfaces
Persistency management

**Figure 1. An example component-based application: the Serendipity-II process-centred environment.**

Serendipity-II's main functional requirements include visual, collaborative process modelling and software agent specification views (1, 2), process enactment and view modification histories (3, 4), to-do lists (5), and agent component information (6, 7). Key non-functional requirements include supporting novice and experienced users, platform independence and mobile computer support, robustness, and security [4].

Serendipity-II was developed using component-based techniques, with many of the components making up the environment reused elsewhere. Examples include enactment and editing event history management and interfaces, collaborative view editing, persistency management, event broadcasting between environments, and version control and configuration management.

We found that traditional approaches to Requirements Engineering [12] are not ideal for developing component requirements. They tend to assume stakeholders and requirements are known, most parts of a system are used in one application (or are not dynamically configurable), and end users don't significantly reconfigure applications. Unfortunately existing component-based development methods usually focus on component design and implementation [1, 15, 16], and usually only provided services are documented. We found this leads to less-reusable components, particularly with regard to component user interfaces and support for distribution and collaborative work. Some have suggested provides-requires relationships between components be reasoned about [13, 17], though have focused on low-level interface specification. Determining customer requirements for product development [8] shares similar issues to component engineering, with stakeholders and usage not well-known. Techniques of ensuring diverse specifications are consistent, use of multiple perspectives and careful refinement of requirements to designs could thus be useful in component RE. Aspect-oriented programming (AOP) [7] uses systemic "aspects" of objects (particularly data distribution and concurrency), augmenting traditional object classes and "weaved" into code. We view weaving from an inter-component view, rather than intra-method, with some components unchangeable COTS parts.

Existing component development tools, such as Visual Age™ [3], focus on design and implementation, as do many CASE tools, such as Rational Rose™ [14] and Software thru Pictures™ [6]. We have found such tools unsatisfactory for analysing and documenting component requirements. Similarly, the component characterisation used by component architectures, like JavaBeans [10] and COM [15], are too low level for describing requirements. Enterprise JavaBeans use a high-level service framework, though currently focus only on service provision. Tools supporting component deployment [7, 18], lack high-level information about components, making run-time configuration difficult. Similarly, most component repositories [11] utilise indexing mechanisms that don't adequately characterise components for retrieval and reuse.

## 3. Component Engineering with Aspects

We have been developing an aspect-oriented component engineering methodology. Aspect-oriented Component Requirements Engineering (AOCRE) within this focuses on identifying and specifying the functional and non-functional requirements relating to key “aspects” of a system each component provides or requires. For example, a developer may identify user interface, collaborative work and persistency-related functional and non-functional aspects of a component, and document provision and required services of the component for each such aspect. Aspects may be decomposed into aspect “details”, for example the data transfer, event broadcasting and version management provides/requires aspect details for collaborative work support. We have developed some useful categorisations of component aspects for design environments, in Table 1. While these categories have been useful for systems we have developed, other categorisations may be better for other domains. Domain-specific aspects can also be identified for specialised components e.g. process modelling for Serendipity-II, which we have found useful for documenting and reasoning about domain-specific component characteristics.

| Aspect | Aspect Details | Description |
|---|---|---|
| User interface | Views<br>Affordances<br>Feedback<br>Extensible parts | Aspects supporting or requiring user interface, including extensible & composable interfaces for several comps |
| Collaboration | Sync. editing<br>Versioning<br>Locking protocol<br>Awareness | Aspects supporting or required for collaborative work by users |
| Persistency | Save/load data<br>Find data<br>Locking<br>Versioning | Aspects supported or required for data persistency management |
| Distribution | Obj. Identification<br>Oper. Invocation<br>Transaction Man.<br>Robustness | Aspects supported or required for distributed object management |
| Configuration | PEMs & Aspects<br>Property sheet<br>Wizard | Aspects supported or required for end user or dynamic configuration of component |

**Table 1. Some useful component aspects.**

Some components may have many aspects and others a few. Unlike traditional object-oriented analysis object services, aspects may share component services, required aspects are as important to characterise as provided aspects, and often more than one other component may provide or require a component's aspects. These “overlapping” aspects are a natural consequence of high-level categorisation of the systemic properties of components, and help requirements engineers gain understanding of related component characteristics.

We thus view aspect characterisation as a way to take multiple, systemic perspectives onto components, and thus better understand and reason about component data, functionality, constraints and inter-relationships. Note that some systemic aspects identified for a component may be redundant in some usage scenarios, and different aspect categorisations may be used depending on both the aspects of reused components and those identified for the system as a whole being developed.

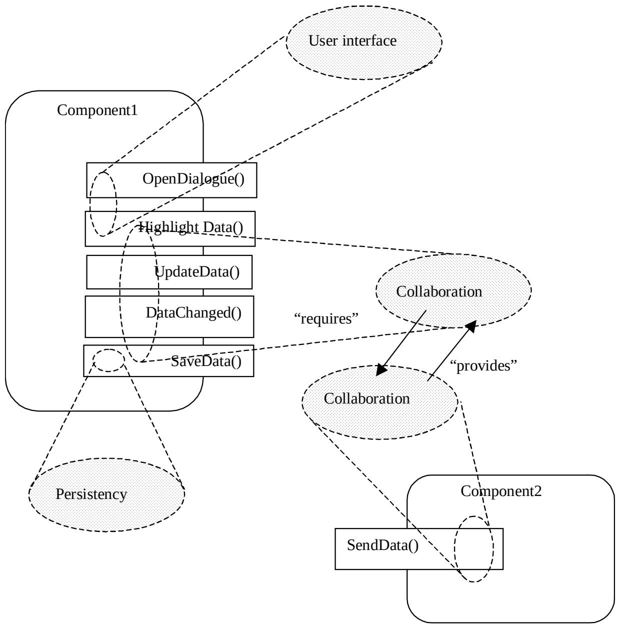


**Figure 2. Basic notion of component aspects.**

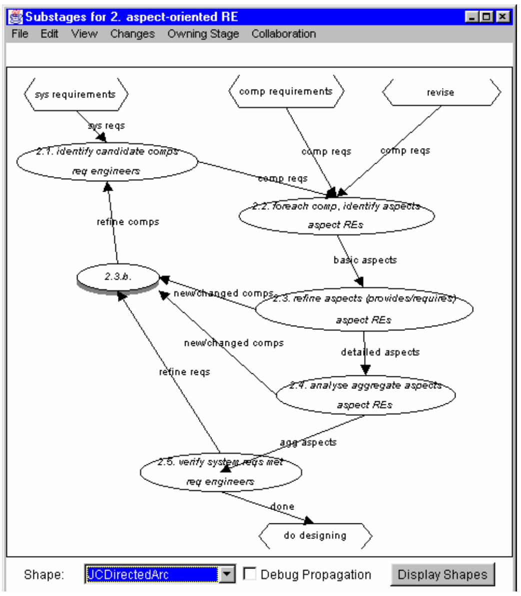


**Figure 3. Basic AOCRE process.**

Figure 3 shows the basic AOCRE process, which begins after analysing general application requirements or individual or groups of components requirements. This allows iterative top-down and bottom-up requirements refinement. Engineers characterise a component's aspects, aspect details, provided and required details, functional and non-functional properties, and reason about inter-related components’ aggregate aspects. Components and aspects identified are refined into detailed component designs. For Serendipity-II we analysed requirements for some reusable components, then designed and implemented these. Serendipity-II requirements were developed and refined into components and aspects. Aspects were reasoned with to determine component composition, configuration and reuse scenarios. Component design and implementation was carried out using these requirements, with feedback evolving reusable and Serendipity-specific requirements.

## 4. Describing Requirements Aspects

Candidate components are found from OOA diagrams, by reverse engineering software components, or bottom-up consideration of individual, reusable components. We have found "perfect" identification of components is not essential during AOCRE, with requirements-level "components" acting as groupings of related services and aspects. These can be split, merged or otherwise refined at design-time, in a similar way to OOA objects being refined into classes. For each component, we identify (using possible stakeholder requirements and object services) aspects for which the component provides services or requires services from other components.

For example, consider the event history component used in Serendipity-II, reused to provide view editing histories, processes stage enactment histories and collaborative editing histories of exchanged events. This component is identified from Serendipity-II requirements, which call for various event histories, or can be considered 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. Components may need to reconfigure event history user interfaces to enable/disable affordances or add their own (see Figure 1).

Figure 4 illustrates aspect-oriented requirements we have identified for the event history component, and some related components used with in Serendipity-II. Components are in solid rectangles, aspect characterisations in dotted rectangles. Aspect details are categorised as being "provided" by a component (denoted by a "+" prefix, e.g. dialogue, basic event management, data serialisation for the event history, or "required" (0"-" prefix), e.g. extensible affordance, event broadcasting and data storage. The aspects provided by the event history are shown in Figure 4, and the usage of provided aspect details and provision of required aspect details indicated between aspects and other Serendipity-II components.

When considering aspects for the event history we identified it must provide a user interface, provide collaborative work support, must be made persistent, and allow configuration of history behaviour. We made user interface affordances "extensible" by other components, avoiding a common problem of inconsistent user interfaces built from mis-matched parts. This need for extension was identified during Serendipity-II requirements specification, where a reused versioning component needs to extend event history affordances. We identified that collaborative work support infrastructure should be provided by other components, as these facilities are reused often by applications.

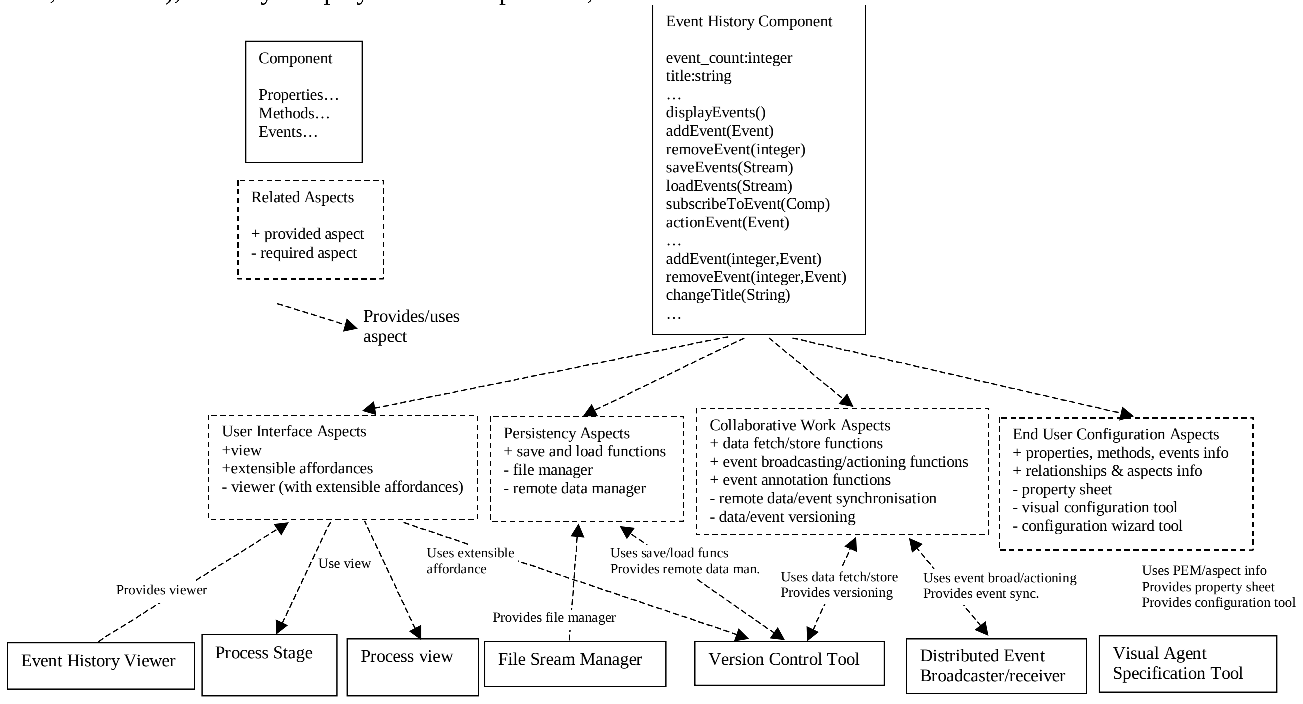


**Figure 4. Example Serendipity-II components and some of their aspects.**

II. Collaborative Work Aspects : COLLABORATION

II. 1) +data fetch/store functions : DATA_MANIPULATION

-- Provides services for getting some/all of event history data and for updating some/all of event history data. Used by components providing collaborative work infrastructure to keep distributed data synchronised or partially synchronised.

QUERY=true; UPDATE=true

II 2) +event broadcasting/actions functions : EVENT_MANAGEMENT

-- Provides services allowing other components to detect event history update events and to action (replay) events received by other components. Used by components providing collaborative work infrastructure to keep distributed event history synchronised or support deltas of event history version changes.

DETECT=true; ACTION=true

II 3) + event annotation functions : AWARENESS

-- Provides services for annotating, selecting, highlighting events. Used by components providing collaborative work infrastructure to support basic group awareness facilities for updated event history events. Other components should use these to annotate events with remote user name, colour them with a colour associated with a particular user, etc.

HIGHLIGHT=colour; ANNOTATE=text

II 4) - remote data/event synchronisation : LOCKING

-- Requires component(s) that supports remote data/event synchronisation. Could support fully synchronised data or semi-synchronous update. This should be robust if network connections fail, and should work over low or high bandwidth networks.

SYNCHRONOUS=true OR false; SEMI_SYNCHRONOUS=true OR false; NETWORK_SPEED=any; STORE=true

II 5) - data/event versioning : VERSIONING

-- Requires component(s) providing data versioning. Should support both event history data and event history update event recording/versioning. This should be a simple-to-use facility for end users. Should extend the viewer affordances to provide at least check-in/check-out capabilities via +extensible affordance aspect.

DATA=true; EVENT=true; INTERFACE=extensible affordances; CHECKIN=true; CHECKOUT=true

**Figure 5. Detailed aspect-oriented component requirements specifications.**

The event history provides basic collaborative work facilities, such as event editing, annotation, actioning received events and providing event listening and export facilities. It requires event and data broadcasting between environments and versioning facilities. Aspect details are kept quite general at the requirements level, and the eventual implementation of these facilities is generally unimportant. During AOCRE generalised aspect details are specified to characterise event history collaborative work-related services. Note event serialisation and deserialisation services are used by collaborative work and persistency aspects, illustrating aspects may overlap.

Detailed textual specifications of aspects provide additional documentation of functional and non-functional requirements. We are developing a set of properties for each aspect detail kind used to more formally describe aspects and aspect usage. Figure 5 shows an example of some codified aspect information for the event history.

## 5. Reasoning with Aspects

After identifying a component's provided and required aspects, related components and aspects can be reasoned about. Inter-component relationships inferred by provided and required aspects allow Engineers to reason about the validity relationships and aspects specified. For example, an event history linked to a component providing only event broadcasting collaborative work aspect doesn't have versioning. The component could be used but would not provide end users or the target application versioned event histories. If versioning is mandatory, the specification is invalid. If a history requires high bandwidth, encrypted data transfer, and is linked to a component providing only modem connection and no encryption, this is invalid.

Aggregate aspects can be identified and specified for groups of interrelated components, allowing Engineers to reason about aspect-oriented requirements for a set of related components, or even global requirements for a whole application. Figure 6 shows an example of a group of interrelated components providing an event history with asynchronous collaboration (via version control), persistency using files, and no synchronous collaborative support or extensible user interface. The aspects of this aggregate are a constrained subset of those of the event history and related components. Global application requirements can be specified using aspects, and then be migrated down to groups of related components or individual components.

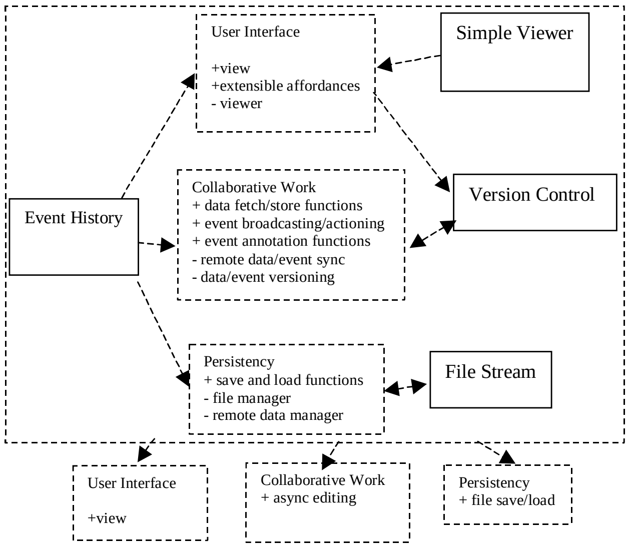


**Figure 6. Example of aggregated aspects.**

Aspects aid in handling evolution of requirements by assisting in categorising requirements changes and localising effects of these changes to relevant aspect categories and aspect details. Changing overall system requirements impacts on aspect-oriented requirements specifications by: changing properties associated with aspect details, adding details or removing details; introducing new aspects and aspect details, or changing inter-component relationships and aspect provides/requires associations; and introducing new components or refining candidate components (merged, split), with modification of associated aspects. Aspects assist in reasoning about modified requirements by aiding requirements engineers in reformulating components, component aspects and provided/required aspects.

## 6. Design, Implementation and Run-time

Aspect-oriented component requirements assist when designing and implementing components. They provide a focused set of functional and non-functional constraints a design can be refined from, and provide a specification that an implementation can be tested against. Requirements-level components can be refined directly to matching design-level software components, or can be split, merged or otherwise revised, as can requirements-level component aspects. They also allow for design decisions to be influenced by weakening or strengthening aspect-level constraints.

Detailed design decisions about the user interface design and behaviour, component persistency and distribution strategies, technologies and available services, collaboration and awareness support facilities, and component configuration tools are usual refinements. Figure 7 shows an example of the refinement of event history component requirements-level aspects to more detailed design-level software component aspects. Some aspects become more specific as e.g. user interface design decisions are made.

Aspects can provide a standardised mechanism for related components to describe and access each other's functionality, or be used to guide inter-component interface definition. A component may thus indirectly invoke other component functionality via operations provided by aspect implementations, or may invoke component operations directly. The former results in more generic, reusable inter-component relationships, while the later is sometimes easier to implement. Aspects can be implemented via interfaces, language reflection or design patterns. We have used all three approaches when implementing components with JViews, our component-based software architecture [8].

Aspect information can be encoded in component implementations for use at run-time by components or end users. Components may query other components for the aspects they provide or require, ask them to perform consistency checks for a configuration, or use their aspect information to reconfigure themselves. For example, a version control tool component queries the event history component for its user interface aspects, locates its preferred extensible affordance aspect (if there is more than one) and requests Check-in and Check-out affordances be added, and is notified when these are accessed. End users can peruse encoded aspect information to determine what functional and non-functional requirements a component has, as shown in Figure 1 (6, 7).

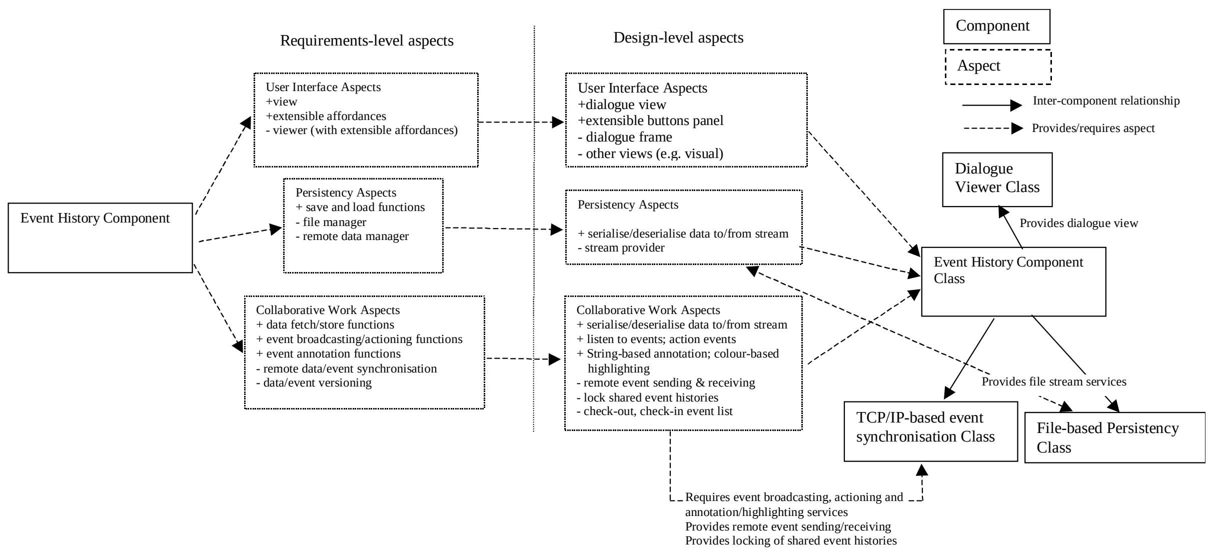

Requirements-level aspects
Design-level aspects
Component
Aspect
Inter-component relationship
Provides/requires aspect
Event History Component
User Interface Aspects
+view
+extensible affordances
- viewer (with extensible affordances)
Persistency Aspects
+ save and load functions
- file manager
- remote data manager
Collaborative Work Aspects
+ data fetch/store functions
+ event broadcasting/actioning functions
+ event annotation functions
- remote data/event synchronisation
- data/event versioning
User Interface Aspects
+dialogue view
+extensible buttons panel
- dialogue frame
- other views (e.g. visual)
Persistency Aspects
+ serialise/deserialise data to/from stream
- stream provider
Collaborative Work Aspects
+ serialise/deserialise data to/from stream
+ listen to events; action events
+ String-based annotation; colour-based highlighting
- remote event sending & receiving
- lock shared event histories
- check-out, check-in event list
Dialogue Viewer Class
Provides dialogue view
Event History Component Class
Provides file stream services
TCP/IP-based event synchronisation Class
File-based Persistency Class
Requires event broadcasting, actioning and annotation/highlighting services
Provides remote event sending/receiving
Provides locking of shared event histories

**Figure 7. Refining requirements-level component aspects into design-level aspects.**

## 7. Tool Support

We have developed some basic tool support for specifying aspects of components in a component-based software development environment, JComposer [3]. Aspects of a software component are grouped and associated with the component, and inter-component aspect usage documented. Some basic validation checking ensures related component aspect requirements are correctly met. Detailed aspect specifications are specified using a hierarchical notation in MS Word™ and aspect detail properties in JComposer. Basic inconsistency management techniques help manage evolving aspect-oriented requirements and include highlighting of changed aspect information in views, change histories for all views and each individual component and each of its aspects, and consistency checks that test if provide/require links between components match.

Requirements-level aspects can be refined into design-level aspects that are associated with classes used to implement a component. JComposer supports generation of information describing a component's aspects that developers and end-users can access at run-time. Basic reverse-engineering capabilities allow components with aspect information to be reverse engineered in JComposer, preserving their aspects.

Aspect-oriented Requirements Engineering can be used to analyse and refine the requirements of new or COTS components. We have used JComposer to characterise the aspects of various software engineering and office automation tools, including MS Excel™, MS Word™, Eudora™, JComposer itself, and Netscape™, and these have been integrated with Serendipity-II [8]. When characterising the aspects of such third party components, it is only necessary to characterise those services or requirements of these systems that are to be used with other components.

To date we have reengineered many Serendipity-II and JViews components and developed several new components using our aspect-oriented approach. Requirements for these components have been documented using aspects and code part-developed using JComposer's support for design-level refinement of aspects. Previously we had used conventional requirements engineering and design approaches when developing environments like Serendipity-II. Our preliminary experience with AOCRE has been very positive, with improved documentation and understanding of component requirements resulting, along with an improved ability to reason about related component requirements using our aspect-oriented perspective. Generally we have found components that have been developed using AOCRE exhibit improved reusability and extensibility, and systems built with these components exhibit improved allocation of responsibility for data and behaviour among both reused and application-specific components.

We are exploring additional visual language support for aspects and aggregated aspects, including better indication of provides/requires aspects spanning several components. Extending our property/value aspect descriptions will help better-describe aspects and provide more formal, rigorous checking. We are developing a repository using aspects to index components for reuse.

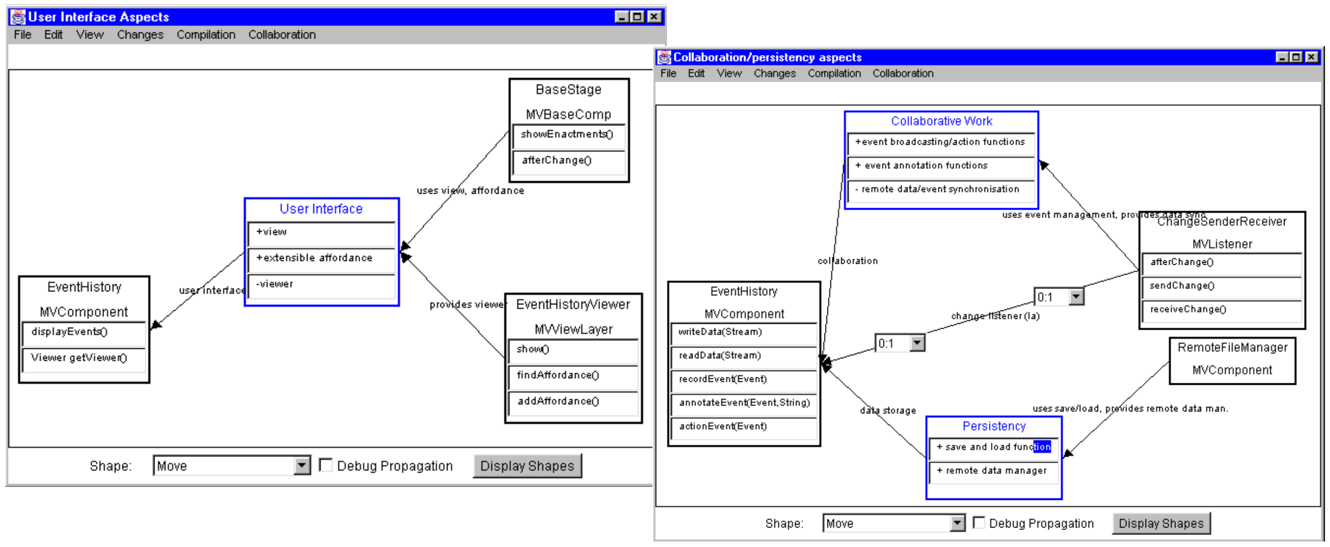


**Figure 8. Specifying aspects in JComposer.**

## 8. Summary

Requirements for component-based software systems are difficult to analyse and specify. We have developed an approach that characterises different aspects of a system each component provides to end users or other components, or requires support for from other components. This allows Requirements Engineers to reason about inter-component relationships that exist for a selected component or for a group of related components using categorised perspectives onto a component's data and behaviour. These aspect-oriented views of a component  Aspect-oriented component requirements can be refined naturally into design-level software component aspects, and can be encoded into component implementations for use at run-time. We have developed basic tool support for aspect-oriented requirements engineering and used the approach for the reengineering of many components and the development of several new components. The resultant requirements are more easily understood, inter-component relationships reasoned about and component specifications more readily reused than if using traditional requirements engineering approaches.

## Acknowledgements

Financial support from the Public Good Science Fund is gratefully acknowledged, as are the many helpful comments of the anonymous reviewers.

## References

1. Brown, A.W. and Wallnau, K.C. Engineering of component-based systems, In Proceedings of the *2nd Int. Conference on Engineering of Complex Computer Systems*, Montreal, Canada, Oct 1996, IEEE CS Press.
2. 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.
3. Grundy, J.C., Mugridge, W.B., Hosking, J.G. Visual specification of multiple view visual environments, In *Proceedings of 1998 IEEE Symposium on Visual Languages*, Halifax, Canada, Sept 1998, IEEE CS Press.
4. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
5. IBM Inc, *VisualAge™ for Java*, 1998, http://www.software.ibm.com/ad/vajava/.
6. IDE Inc., *Software thru Pictures™ 7.0*, 1998, http://www.ide.com/Products/SMS/core7.0.html.
7. 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.
8. Litva, P.F., Integrating Customer Requirements into Product Designs, *Journal of Product Innovation Management*, Vol. 12, No. 1, pp. 3-15.
9. Netscape Communications Inc, *Visual Javascript™*, 1998, http://www.netscape.com/ compprod/products/.
10. O'Neil, J. and Schildt, H. *Java Beans Programming from the Ground Up*, Osborne McGraw-Hill, 1998.
11. Park, Y. and Bai, P. Retrieving software components by execution, In *Proeedings of the. 1st Component Users Conference,* Munich, July 14-18 1996, SIGS Books.
12. Pressman, R. *Software Engineering : A Practitioner's Approach*, McGraw-Hill, 4th Edition, 1996.
13. Rakotonirainy, A. and Bond, A. A Simple Architecture Description Model, *In Proceedings of TOOLS Pacific'98*, Melbourne, Australia, Nov 24-26, 1998, IEEE CS Press.
14. Rational Corp., *Rational Rose 98*, 1998, http://www.rational.com/products/rose/prodinfo.html.
15. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
16. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
17. Szyperski, C.A. and Vernik, R.J. Establishing system-wide properties of component-based systems, *OMG/DARPA Workshop on Compositional Software Architecture*, Monterey CA, Jan 6-8 1998.
18. Wagner, B., Sluijmers, I., Eichelberg, D., and Ackerman, P., Black-box Reuse within Frameworks Based on Visual Programming, In *Proeedings of the. 1st Component Users Conference,* Munich, July 14-18 1996, SIGS Books.