An implementation architecture for aspect-oriented component engineering

John Grundy

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

Abstract

Aspect-oriented component engineering (AOCE) is a new technique for engineering software components, using a concept of provided and required systemic aspects of a component's non-functional and functional characteristics to support component composition and interaction. These aspects include component user interfaces, collaborative work support, distribution and persistency, security, data management, processing, component inter-relationship and configuration characteristics. We describe support for AOCE in the JViews software architecture via the use of aspects, aspect details and detail properties. We describe implementation of this aspect information using the Java language, including the use of AspectDetail classes to augment JavaBeans, with different specialisations supporting decoupled component aspect querying and access to component services in a very de-coupled manner.

Keywords: aspects, software components, software architecture, introspection

1. Introduction

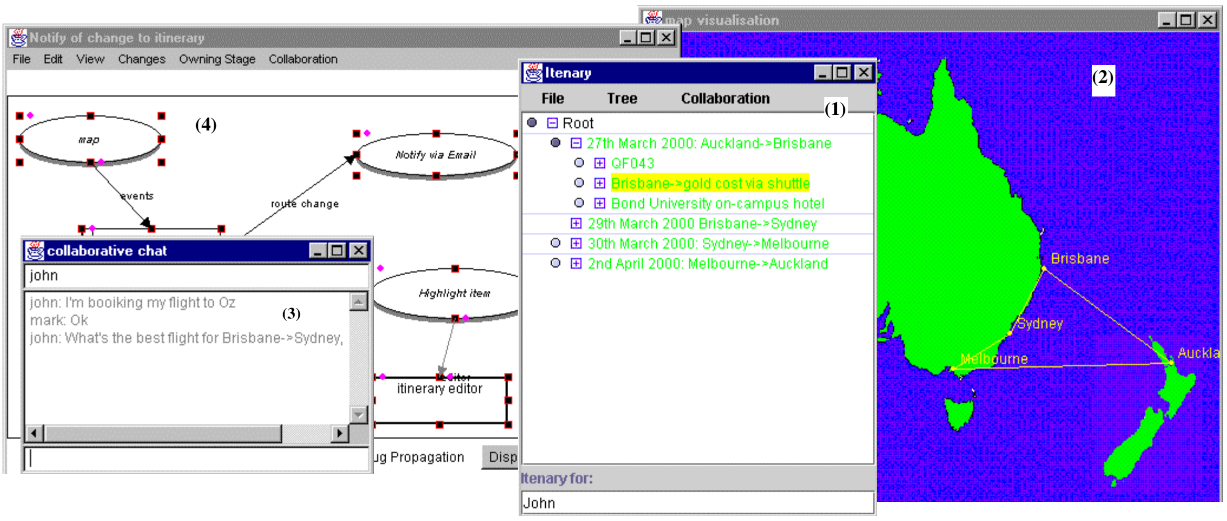
Component-based systems have become very popular as an alternative approach of composing systems from reusable software "building blocks", rather than developing monolithic applications [13], [12]]. Component-based development methods and technologies typically focus on packaging "vertical slices" of system functionality in components that are then combined and reused in diverse ways. Components are typically (though not always) coarser-grained than objects, provide introspection mechanisms and more flexible coupling mechanisms, such as event subscription, and can often be dynamically deployed.

A problem of component-based development is the spreading of systemic properties of a system throughout component code e.g. user interface, collaboration, distribution, persistency, security etc. We have developed Aspect Oriented Component Engineering (AOCE), a new methodology for component-based development that uses systemic aspect characterisation of component properties to help improve component design and implementation [3]. Component aspects take a "horizontal slice" across systemic features of applications (user interface, security, distribution, etc.) and capture the provided and required capabilities of components.

We describe the use of AOCE to support component design, and the architectural support for aspects in an implementation framework. Components encode, using a collection of AspectDetail specialisations, aspect information for run-time usage. At run-time components can access other components' aspects and use them to determine component capabilities, and sometimes can use AspectDetail object methods to invoke component functions in a loosely-coupled way.

2. AOCE

We use an example component-based application and some of its constituent components to illustrate AOCE and its architectural support in this paper.



john

john: I'm booiking my flight to Oz
mark: Ok
john: What's the best flight for Brisbane->Sydney,

Root
27th March 2000: Auckland->Brisbane
QF043
Brisbane->gold cost via shuttle
Bond University on-campus hotel
29th March 2000 Brisbane->Sydney
30th March 2000: Sydney->Melbourne
2nd April 2000: Melbourne->Auckland

John

Figure 1. Example of a component-based application.

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 (1), used to view and edit travel itineraries; a map visualisation component (2); itinerary item property dialogues; a collaborative text chat (3); a web browser; and several collaborative work-supporting notification and task automation agents (4) (e.g. multi user editing, multiple cursors, change notifiers etc.).

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. AOCE develops characteristics of these provided and required "aspects" of components, grouping them by the systemic aspects they relate to.

Figure 2 shows a simple example of such "horizontal slicing" of a component's capabilities. This provides multiple perspectives of the chat component, 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. 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. Aspects thus allow developers to specify the provided and required capabilities of components and allow them to specify functional and non-functional characteristics of these capabilities.

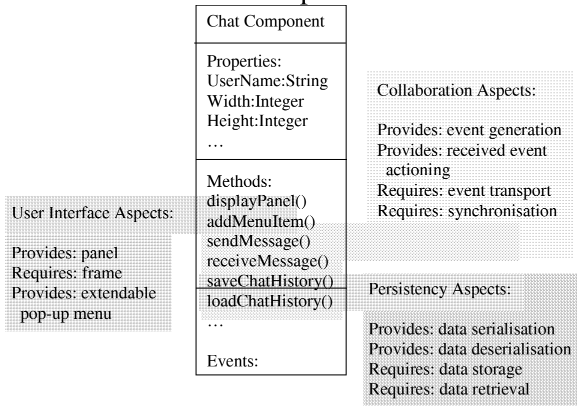


Figure 2. Example aspects.

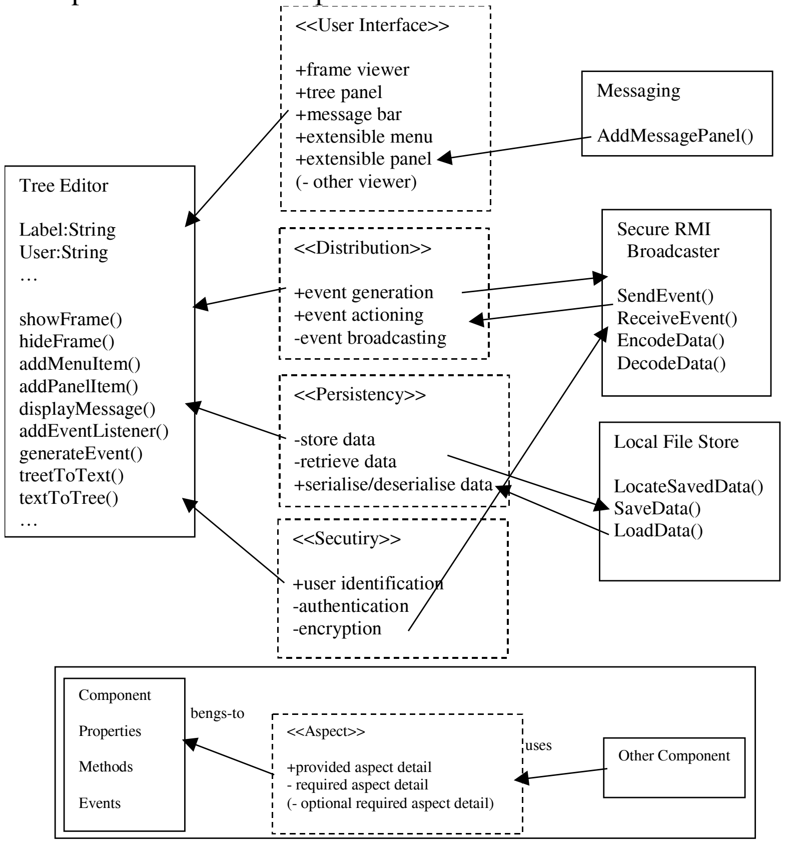
AOCE is used throughout a component's lifecycle. We have extended a CASE tool, JComposer, and component implementation framework, JViews, to incorporate the notion of component aspects to support component design and implementation. Requirements engineers develop abstract aspect characteristions of components based on user requirements. Refinements of these components and their aspects are developed to produce design-level system components and design-level aspects, and component-implementing classes are generated. AspectDetail class specialisations are also generated for each component to codify their aspects, and provide methods and interfaces used to implement highly de-coupled component interactions at run-time.

3. Architectural Support

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. 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 better supports COTS component reuse and dynamic system configuration.

We have extended our component implementation framework, JViews, to incorporate characterisations of component aspects, including aspect categories, aspect details and aspect detail properties. Components publicise their aspects (both required and provided aspect details) using aspect categories, and each provide and required aspect detail has a set of property values further characterising the component capabilities.

Figure 3 shows an example of characterising the tree editor component capabilities using aspects, modelled with the JViews Architecture Description Language (ADL). The tree editor: provides a user interface frame and configuration property sheet, and may optionally require another form of structure viewer (e.g. outliner).



(a) Visual representation of aspects and aspect details.

```
Component TreeEditor

  Properties
    …
  Methods
        showFrame()
        hideFrame()
        addMenuItem()
        …
  Events
    …
  Aspect UserInterface
    Provides frame viewer : Frame
      Default size=…
      Relevant methods=[showFrame, hideFrame]
      …
    End frame viewer
    Provides extensible menu : Extensible Affordance
      Affordance type=menu
      Extensions allowed=[add to end, insert item]
      Can disable=False
      …
    End extensible menu
    …
  End UserInterface

  Aspect Distribution
    Provides event generation : Event Generation
      Generation time=[before, after]
      Relevant Events=[all]
            Relevant methods=[addEventListener,   generateEvent]
        …
   End event generation
   Requires event transport : Event Transport
      Transport kind=[lan,wan,wireless]
      Event transport speed=between 10 and 100
      …
    End event transport
    …
  End Distribution
  …
End TreeEditor.
```

(b) Textual specification of aspects.

Figure 3. Aspects for the tree editor component, modelling using the JViews ADL.

It 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 users.

4. Implementation Support

In order to support AOCE at the component implementation level, we have extended our JViews component-based framework to incorporate aspect information. This has been achieved by providing each JViews component with a set of AspectManager objects, one for each kind of aspect category, with each manager having a set of AspectDetail objects of various kinds. Different AspectDetail objects have different sets of aspect detail properties relevant to the kind of aspect detail being represented. JViews component classes inherit from a JVComponent class that includes functions to access AspectManager and AspectDetail objects. Figure 4 shows a UML class diagram describing the representation of JViews aspect manager and detail objects. Each manager object organises and provides functions to access (and modify) aspect details of a particular aspect category e.g. User Interface, Distribution, Persistency and so on. The AspectManager classes provide functions to query, retrieve and modify AspectDetail objects they organise. The AspectDetail classes provide generic functionality to identify (name) each aspect detail for a component, as well as common property management and annotation functions.

A variety of AspectDetail specialisations are used to capture extra information (i.e. aspect detail properties) about aspect details, and many AspectDetail specialisations also provide aspect detail-specific component querying and manipulation functions. For example, the ExtensibleAfforanceAspectDetail class is used to characterise components that have a user interface affordance that can be extened e.g. an extensible pull-down menu or list of buttons. It provides properties to characterise the kind of extensible afforance (e.g. menu, button, panel etc.), how the affordance can be extended (e.g. add to end of list, insert item, replace item etc.), and functions that can be invoked to carry out user interface affordance extension (addAffordance(), etc).

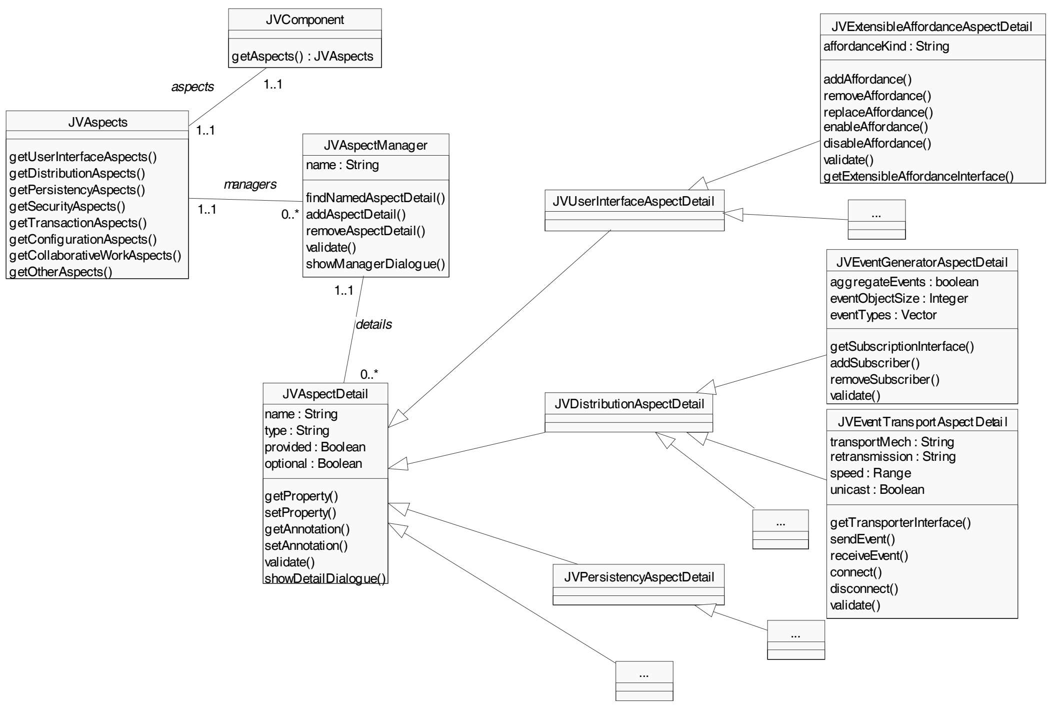


Figure 4. Some AspectManager and AspectDetail classes from the JViews framework.

Components providing an extensible affordance capability advertise that they provide this via a ExtensibleAfforanceAspectDetail object. Other components that require an extensible affordance advertise this, and they can use the ExtensibleAfforanceAspectDetail functions to both discover the capabilities of the provider component as well as dynamically extend the provider's user interface in a controlled, de-coupled fashion.

AspectDetail objects also include validation functions that can be called at run-time to check components are correctly combined i.e. their aspect details and detail properties are consistent and sufficient to allow them to operate. We have developed a range of AspectDetail specialisations, including various ones for collaborative work support, persistency, security, component configuration and transaction processing characterisation. We are continuing to develop more of these as they are needed in our component-based systems.

JComposer generates code in components to create and initialise their AspectManager and AspectDetail objects. Many JViews framework components provide standard aspect objects and implement interfaces, and specialisations for new components may also implement additional interfaces relating to their aspect-based capabilities. Our JViews components are currently specialisations of the JavaBeans component model, adding extra event subscription and notification support.

5. Run-time Aspect Usage

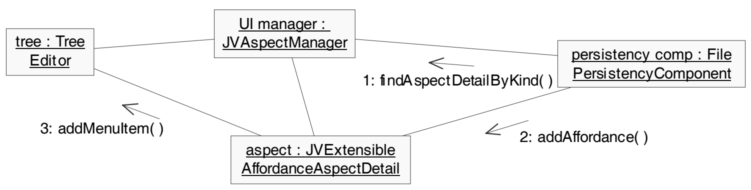
AspectDetail objects are used to introspect a component's capabilities at run-time, to provide a standardised access point for invoking functions of a component that implement aspect-related services, or be used to re-configure a component. Figure 5 shows some examples of AspectDetail object usage at run-time. Figure 5(a) shows a persistency management component, which needs to add "Save" and "Load" affordances to the tree editor's user interface. The persistency component accesses the tree editor's user interface manager (1) to obtain an extensible affordance aspect detail object; (2) invokes the addAffordance() function (2), passing new affordance labels, which calls appropriate functions implemented by the tree editor (3). In our implementation, the persistency manager extends the "File" menu, adding "Save" and "Load" menu items. The persistency component knows nothing about the tree editor component and only interacts with it via the functions provided by ExtensibleAfforanceAspectDetail object.

Figure 5(b) shows a collaborative editing component using a distribution aspect manager (1) to discover the event subscription interface of the tree editor (2), and uses this interface to subscribe to editing events (3, 4). When the collaborative editing component receives events (5), it sends events to a corresponding user's collaborative editing component via an event transport component (6, 7). Received events (8) are sent to the tree editor for actioning (9). This could be done via an event actioning aspect detail object, interface implemented by the tree editor (as here), or by a particular tree editor function being called.

Figure 5(c) shows the use of a third component to mediate inter-component communication. A persistency component locates (1) and accesses a security encoder aspect detail object (2), using it to obtain a specified data encoder component. When tree component data needs to be stored, the persistency component obtains the data (3, 4), and uses the data encoder to encrypt it (5). When restoring data, the encoder is used to decrypt it (6) before the persistency component passes it back to the tree component (7, 8) to restore the tree's state.

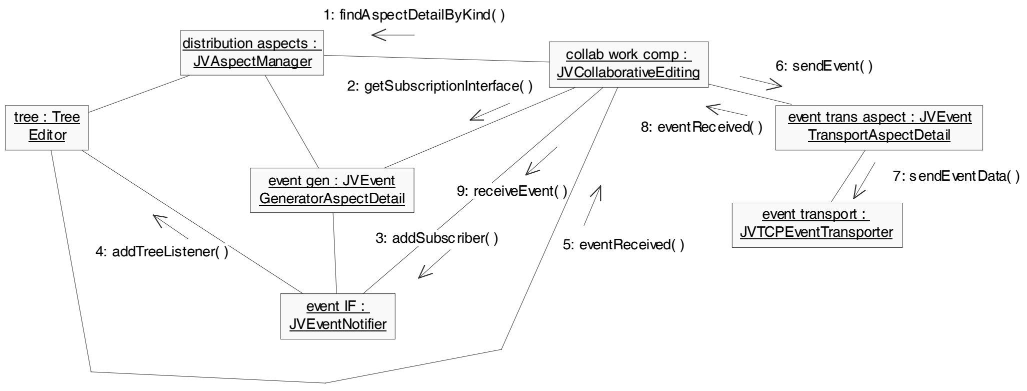
6. Discussion

Aspect-oriented Component Engineering aims to increase component developers' ability to engineer reusable software components that are more easily combined and reconfigured, both statically and dynamically. Most current component technologies and methods, such as DCOM, JavaBeans and CORBA C-IDL [12], [7], [9], do not adequately support component service characterisation, and generally focus only on vertical component provided functionality. Focusing on both provided and required horizontal component services, aspects help component engineers to build components with interfaces that support very de-coupled component interaction.



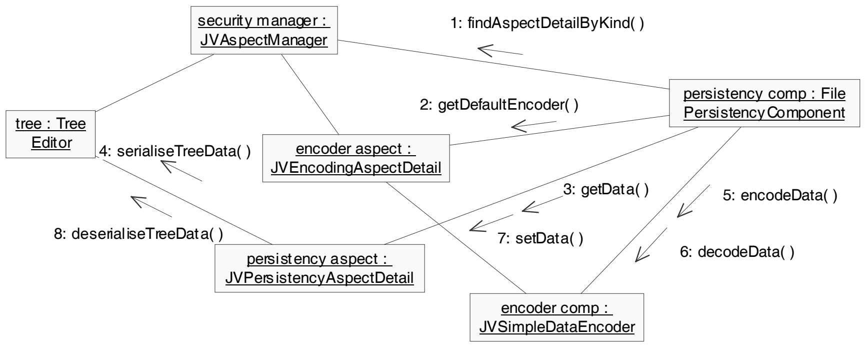
tree : Tree Editor
UI manager : JVAspectManager
persistency comp : File PersistencyComponent
aspect : JVExtensible AffordanceAspectDetail
1: findAspectDetailByKind( )
2: addAffordance( )
3: addMenuItem( )

(a) Tree editor and persistency component collaboration.



1: findAspectDetailByKind( )
distribution aspects : JVAspectManager
collab work comp : JVCollaborativeEditing
6: sendEvent( )
2: getSubscriptionInterface( )
tree : Tree Editor
8: eventReceived( )
event trans aspect : JVEvent TransportAspectDetail
event gen : JVEvent GeneratorAspectDetail
9: receiveEvent( )
7: sendEventData( )
event transport : JVTCPEventTransporter
4: addTreeListener( )
3: addSubscriber( )
5: eventReceived( )
event IF : JVEventNotifier

(b) Tree editor, collaborative work and event transport component collaboration.



security manager : JVAspectManager
1: findAspectDetailByKind( )
2: getDefaultEncoder( )
persistency comp : File PersistencyComponent
tree : Tree Editor
4: serialiseTreeData( )
encoder aspect : JVEncodingAspectDetail
3: getData( )
5: encodeData( )
7: setData( )
8: deserialiseTreeData( )
6: decodeData( )
persistency aspect : JVPersistencyAspectDetail
encoder comp : JVSimpleDataEncoder

(c) Tree editor, security component and event transport component collaboration.

Figure 5. Three examples of inter-component service access via AspectDetail objects.

Some component engineering methods ([11] and [14]) take into account component interface requirements and system-wide component properties. However, these are typically only used for characterising limited forms of component services, such as distribution. Most also focus only on component functionality and often not non-functional characteristics.

Aspect-oriented Programming [5] and Adaptive Programming [6] have become popular approaches to describing design-level aspects of programs, and incorporating support in programs via code weaving and component adaptors. AOCE differs in that its intention and architectural support are aimed at characterising component services and providing interfaces to combine provided and required services in flexible ways, rather than specifying aspect information separately to OO programs and weaving support into them, or wrapping components and linking interfaces with plug-in adaptors. Another key difference is our support for dynamic system composition and the ability to reconfigure and use different aspect-related services at run-time.

Dynamically reconfigurable systems have become important in many domains, including end user computing systems [8], agent-oriented systems [10], and various software tools [2].. Most of the architectures used to build such applications adopt ad-hoc solutions to reconfiguration and inter-component interaction [8], or use domain-specific solutions, such as agent communication languages, workflow and software data interchange formats [2], or basic component interface usage models [1], [4].

Compared to our component aspects and their realisation using aspect detail objects, these approaches are less flexible, do not capture the range of systemic system services of aspects, and result in generally less reusable and reconfigurable components.

We have deployed our aspect-oriented components in a range of complex problem domains, including a CASE tool, process management system and collaborative travel itinerary planner [3], [4]. We have identified a wide range of systemic aspects for which a number of aspect details and properties can be identified for characterising component provided and required services. The use of our component aspects architecture has allowed us to engineer components which are highly reusable and reconfigurable. Aspects ensure these components are implemented in such a way that their interfaces support appropriate interaction and configuration functions that other components can leverage without needing hard-coded knowledge of component types.

7. Summary

We have been developing a new methodology for engineering software components, and a corresponding design and implementation architecture. Our aspect-oriented component implementation architecture allows developers to describe various component services using systemic aspect categorisations, and have specialisations of AspectDetail objects generated to represent many of the provided and required services of components. AspectDetail objects provide properties to specify the nature of provided and require services, and functions which can be used to configure components and access their aspect-related services in a very de-coupled manner, obviating the need for many components to have hard-coded type and interface knowledge of other components they might be reused with. At run-time, AspectDetail objects are queried to determine component capabilities, and their functions called to obtain aspect service-related interfaces a component supports or even to access and configure services.

References

[1] Brown, A.W. and Wallnau, K.C. Engineering of component-based systems, In Proceedings of the 2nd *International Conference on Engineering of Complex Computer Systems*, Montreal, Quebec, Canada, October 21-25, 1996, IEEE CS Press.

[2] Gray, .J.P., Liu, A. and Scott, L. Issues in software engineering tool construction, Information and Software Technology, **42** (2), Elsevier, 73-77.

[3] Grundy, J.C. Supporting aspect-oriented component-based systems engineering, In *Proceedings of 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 388-395.

[4] Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences, *Journal of Information and Software Technology*, **42** (2), January 2000.

[5] 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.

[6] 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.

[7] Monson-Haefel, R *Enterprise JavaBeans*, O'Reilly, 1999.

[8] Morch, A. Tailoring tools for system development, *Journal of End User Computing,* **10** (2), 1998, pp. 22-29.

[9] Mowbray, T.J., Ruh, W.A. *Inside Corba : Distributed Object Standards and Applications*, Addison-Wesley, 1997.

[10] Nwana, H.S. Software Agents: An Overview. *The Knowledge Engineering Review,* **11** (3) (1996), 205-244.

[11] 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.

[12] Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.

[13] Szyperski, C.A. *Component Software: Beyond OO Programming*, Addison-Wesley, 1997.

[14] Szyperski, C.A. and Vernik, R.J. Establishing system-wide properties of component-based systems: a case for tiered component frameworks, *Workshop on Compositional Software Architecture*, Monterey, CA, Jan 1998.