Developing Software Components with Aspects: Some Issues and Experiences

John Grundy and John Hosking

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

Abstract

Engineering software components is quite a challenging task. Existing approaches to component-based software development are for the most part focused on functional decomposition. All have a number of weaknesses as they do not take into account cross-cutting concerns impacting components. In this chapter we outline a method we have developed called "aspect oriented component engineering". Our approach uses aspects to help engineer better software components. We motivate our work with a simple distributed system example. We then describe how component specifications and designs can use aspects to provide additional information about components. We describe how aspects can be used to help implement more de-coupled software components. We show how encoded aspect information can be used at run-time to support component plug-and-play, retrieval and validation. We compare and contrast our approach to other component engineering methods and aspect-oriented software development techniques.

1. Introduction

Component-based systems development is the composition of systems from a set of "software components". These components encapsulate data and functions. They often provide events, are self-describing, and many can be dynamically "plugged and played" into running applications [1, 5, 44, 47]. Often we make use of a mixture of newly built and existing COTS (Commercial Off-The-Shelf) components. For the later, we usually have no access to source code. Engineering software components is quite a challenging task. Components must be identified and their requirements specified [1, 44]. Component interaction is crucial, so provided and required component behaviour needs identification and documentation [37]. Ideally, components are implemented using a component technology that supports a high degree of component reuse. Users of components may want to be able to understand and correctly plug-in components at run-time.

We have found problems with most component design methods [5, 1, 24] and implementation technologies [38, 31, 47, 21]. In our experience, these do not produce sets of components with sufficiently flexible interfaces, run-time adaptability or good enough documentation [12, 13]. A major weakness is the inability to describe functional and non-functional characteristics and inter-relationships of the components. We initially trialed the use of *aspects* (cross-cutting concerns) at the requirements level to improve the description of our components [10]. When this proved to be successful, we applied the concept to component design and implementation [12, 11]. This involves using aspects to better describe the cross-cutting concerns impacting on components at the design level. We then make use of these aspect-oriented component designs to help build components with more reusable and adaptable functionality. We have also used encodings of aspects associated with software components at run-time [13]. This uses aspect information to support dynamic component adaptation, introspection, indexing and retrieval, and validation.

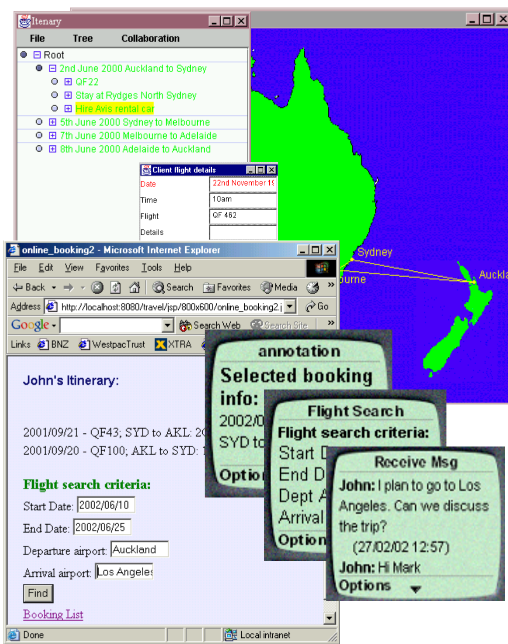
Most aspect-oriented software development uses aspects in similar ways to us. Aspects are used to identify and codify cross-cutting concerns impacting on objects. Some reflective systems use aspect information to support run-time adaptation [35, 28]. Aspect programming systems weave code into join-points of programs

[22, 23, 3]. Some design approaches use aspects (or "viewpoints" or "hyper-slices") to provide multiple perspectives onto the object designs [45, 19, 33, 15, 7].

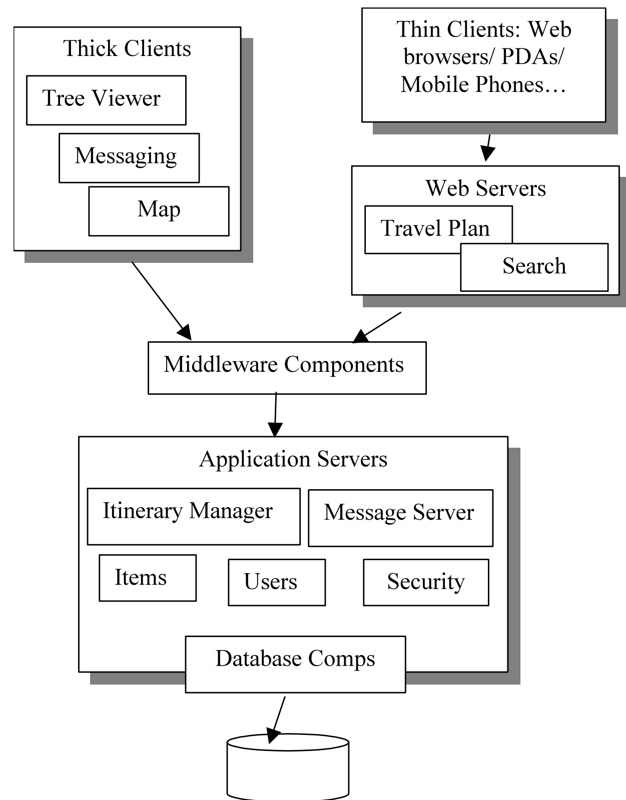
In this chapter, we provide a summary for non experts of our work applying "aspects" to the development of software components. We first motivate our work with key issues developers face when building component-based systems, using a simple example to illustrate. We then overview our use of aspects to help build software systems. In later sections, we, in turn, discuss the use of aspects for component specification and design, implementation and at run-time. We compare and contrast our work with other component engineering, component specification and aspect-oriented software development methods. Readers interested in more technical detail or more comprehensive evaluation and comparison of our work can find this in our previously published papers [10, 11, 12, 13].

2. Motivation

Consider a collaborative travel planning application that is to be used by customers and travel agents to make travel bookings [13]. Examples of the user interfaces provided by such a system are illustrated in Figure 1 (a). Some of the software components composed to form such an application are illustrated in Figure 1 (b).



(a) Travel planning application



(b) Example travel planner components

Figure 1. Example component-based application.

We built this component-based system by composing a set of software components to provide the facilities required by the users of the system. These include travel itinerary management; customer and staff data management; system integration with remote booking systems; and various user interfaces. Some components are quite general and highly reusable. For example, the map visualisation, customer data manager, chat and email message clients and server, and middleware and database access components. Others components, such as the travel itinerary manager, travel item manager, travel booking interfaces and integration components, are much more domain-specific.

When building such an application, a developer needs to identify and assemble a large number of components. These have usually been built using “functional decomposition”, organising system data and functions into components based on the vertical piece(s) of system functionality they support. However, many systemic features of an application end up cross-cutting, or impacting on, many of the different components in the system [22, 45]. For example, user interfaces, data persistency, data distribution, security management and resource utilisation all impact on a wide variety of components including their methods and state. Some components provide functionality relating to these system features. Others require it from other components in order to be able to operate. We use the term “aspects” to describe these cross-cutting, horizontally-impacting concerns on the software components that make up a system [10, 12].

To give an idea of how different systemic aspects impacting on components works, Figure 2 shows three components from the travel planner system, Tree viewer, Travel itinerary and Database, User interface, Persistency, Collaborative Work and Transactions are the aspects cross-cutting the components’ methods and state. The Tree viewer provides user interface and collaborative work support. The Travel itinerary component requires user interface and persistency support in order to work, but provides data to render and store itinerary items. A Database component provides data storage and transaction co-ordination support but requires transaction co-ordination. The three components must work with each in order to provide the travel plan viewing, business process and data management required by the system. Note that each of these components is thus impacted by several “aspects” in different ways.

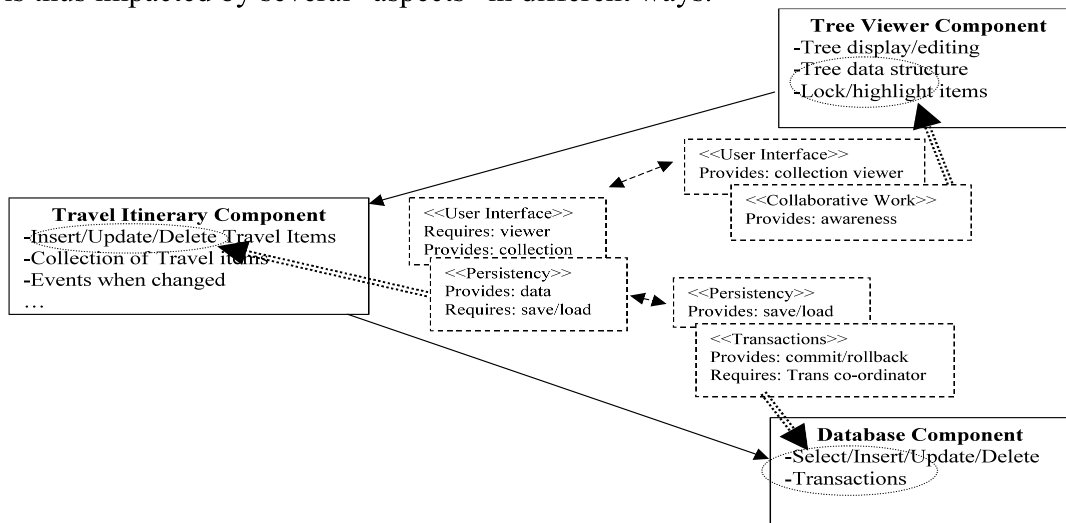


Figure 2. The concept of aspects cross-cutting inter-related software components.

For each aspect impacting a component, further information also needs to be specified in the form of a set of *aspect details* that the component provides or requires. Each aspect detail may also be constrained by one or more *aspect detail properties* that describe detailed functional or non-functional constraints. In our example, we may specify that the Itinerary manager is impacted by the Persistency aspect. We may then specify that the nature of this Persistency impact is that it requires a component providing data storage (a Persistency aspect detail). We may further specify that the data storage provided to it must meet some level of performance constraint. For example, 100 insert() and update() functions must be supported per second (an aspect detail property constraint for the data storage aspect detail). Other aspect details might specify the kind of awareness supported by the tree viewer (e.g. highlight of changed items); the kind of authentication or encryption used (Security aspect details); the upper bounds of resources used, performance required, or concurrency control techniques they enforce [12, 11].

In aspect-oriented programming languages [23], aspects are used to support code injection into methods. For example, in Aspect/J point-cuts might be used to specify where to add persistency management, memory utilisation, user interface and distributed communication code. In aspect-oriented design [19, 42, 43], the aspects are used to describe cross-cutting concerns impacting on the components. In dynamic aspect-oriented

programming [35], components might be modified at run-time using the aspects to change their parameters or their running code.

3. Our Approach

We have developed aspect-oriented component engineering, a new method for developing software components with aspects. Essentially the use of aspects provides us with “multiple perspectives” on software component designs. Figure 1 shows our approach and its use of aspects. Component specifications and designs, typically UML diagrams, are augmented with aspect information (1). These aspects are used to reason about the component specifications, architectures and designs. We use them to determine whether required aspects are met in proposed component configurations. We also check whether component configurations are “consistent” with respect to the aspect constraints. When implementing designs we use the aspect information to help us develop more de-coupled component interaction and dynamic component configuration. This enables us to maximise the amount of component reuse and dynamic component adaptation possible (2). We encode the aspect information about software components in a run-time accessible form (3). At run-time, this information allows components to be introspected i.e. understood by end users and other components. We use this encoded aspect-based information to support dynamic run-time component re-configuration and adaptation. We have also used it to support component storage and retrieval from a repository and component validation by dynamic test generation and execution (4).

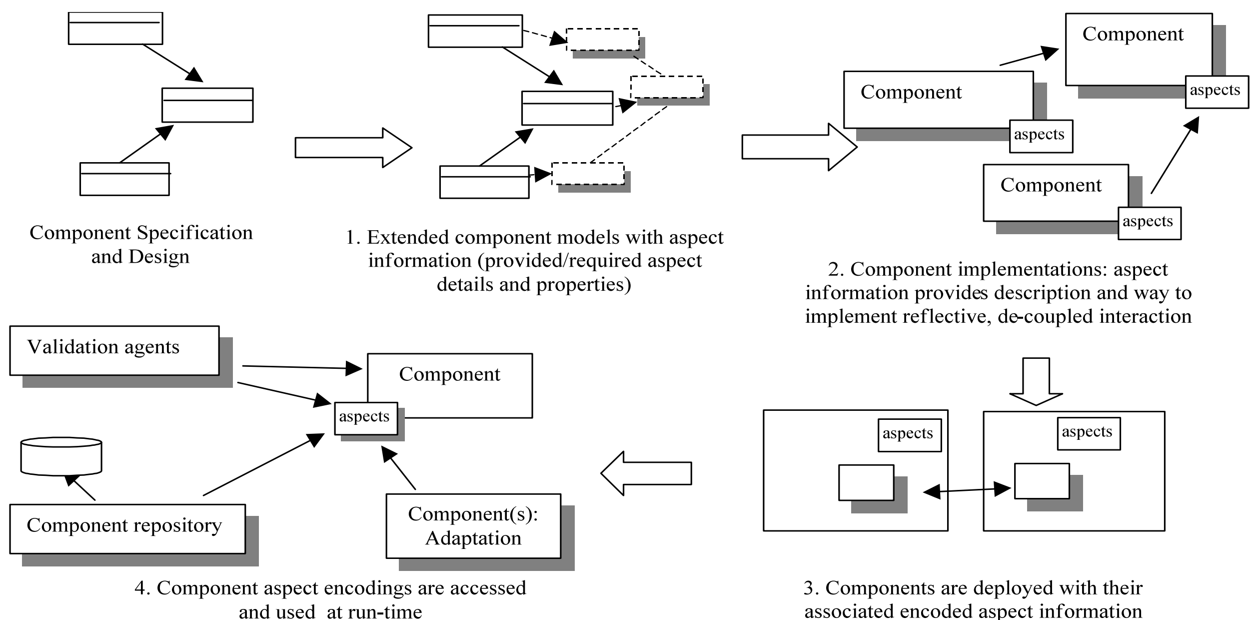


Figure 3. An overview of aspect-oriented component engineering.

In the following three sections, we briefly illustrate how we use aspects for component design, de-coupled and configurable component implementation, and at run-time. We use a few simple examples from the travel planner outlined previously. We then compare and contrast our work with that of others.

4. Component Specification and Design with Aspects

In our approach to component development one of the key uses of aspects is to give developers a way to enrich their component specifications and designs. They do this by capturing cross-cutting impacts of systemic functionality and associated non-functional constraints as aspects. Note that using aspects is only one approach of doing this. Approaches using some form of multi-perspective or viewpoint representations are also common [7, 9, 15, 33, 45]. We chose to use aspects to give developers a way to categorise these concerns impacting different components and different parts of components.

During requirements engineering we use aspects to document the functional and non-functional properties of a component. These are then grouped using a set of aspect categories. Common categories that we have used are User interface, Collaborative work, Component configuration, Security, Transaction processing, Distribution, Persistency and Resource management. Domain-specific aspects can also be used. These might be parts of the system that provide or require services relating to Travel itinerary management, Payment and Order processing.

Figure 4 illustrates a simple use of aspects when specifying two inter-related software components. In this example the travel planner's requirements identify that a number of components must provide "extensible" user interfaces. By this we mean that a component may provide a user interface and another component at run-time adapt this interface in some way. This is usually by adding a new user interaction "affordance" to it in some structured way. For example, the travel itinerary construction use case says that new "travel item" construction facilities must be able to be added dynamically to the itinerary planning user interfaces.

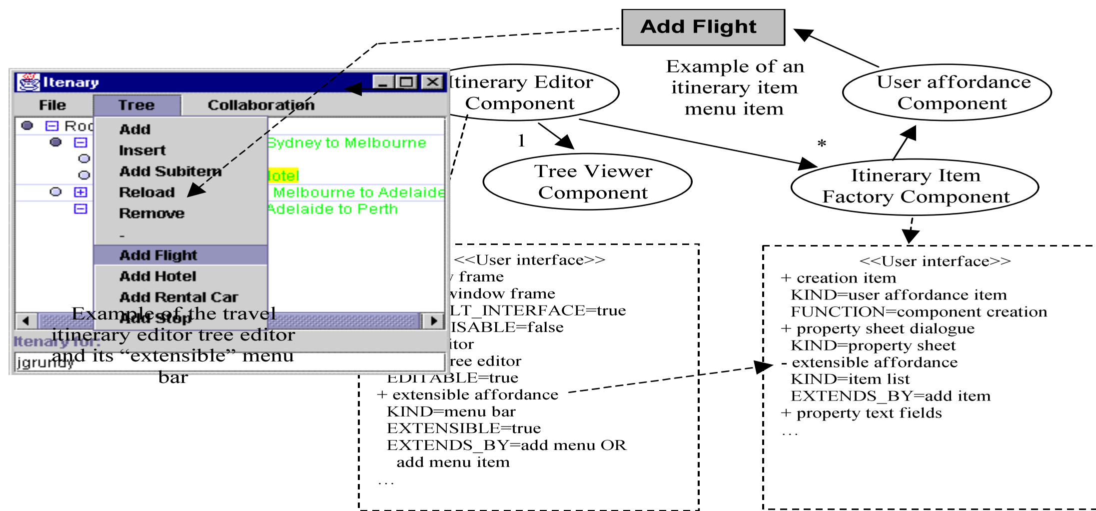


Figure 4. A simple component specification example using aspects.

Figure 4 shows the itinerary editor component, which uses a tree editor and multiple itinerary item creation (factory) components. The aspects of the itinerary editor specify that it provides (among other things) an "extensible affordance" user interface facility. This means other components can extend its user interface in certain, controlled ways. Another component, an itinerary item factory, is used to create particular kinds of itinerary items - flights, hotel rooms, rental cars. This factory requires a component with an extensible affordance so it can add a button, menu item or drop-down menu item to this user interface to allow the creation of different kinds of travel items. In this example, the provided user interface extension aspect detail in the itinerary editor aspects satisfies the required one in the factory's aspects. This approach can be used to describe a large range of provided and required component functional and non-functional properties. Developers can then reason about the inter-relationships of components using this information.

During design, developers refine component specifications into detailed designs and then implementation solutions. They also refine the aspect specifications to a much more detailed level. To describe their designs developers create additional design diagrams. Each diagram focuses on particular aspects impacting on a group of related components. An example from the travel planner system is shown in Figure 5, an annotated Unified Modelling Language (UML) sequence diagram. This describes component interactions in the travel planning system as a user constructs part of a travel itinerary. In the example, the user interacts with a web-based thin-client interface for the itinerary editor. This in turn interacts with application server-side

components. Additional middleware and database components provide the infrastructure for the architecture of this design.

In this example, we have used UML stereotypes to indicate the aspects impacting on each component. Both method invocations and component objects have been annotated in this way. We have used UML note annotations to further characterise particular method invocations between components. These indicate provided and requires aspect details. Some details are further characterised by adding non-functional constraints. In this example, they include the type of security and data storage required and required performance measures of the distributed system communication.

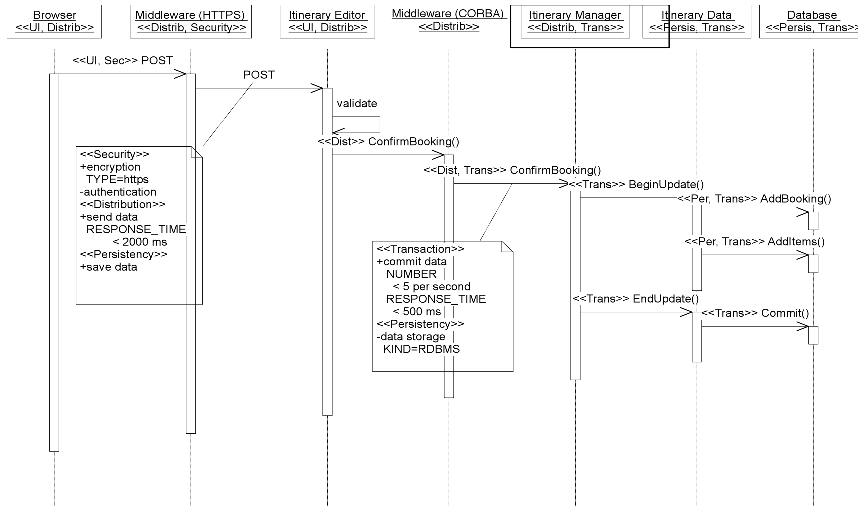


Figure 5. Component interaction design example.

Such aspect-augmented design diagrams allow developers to capture richer information about their component designs. Developers can augment existing diagrams or create new ones. New diagrams allow them to focus in on particular parts of a system or particular component interactions. Note that our aspects give developers a way to capture and document both functional and non-functional constraints during design.

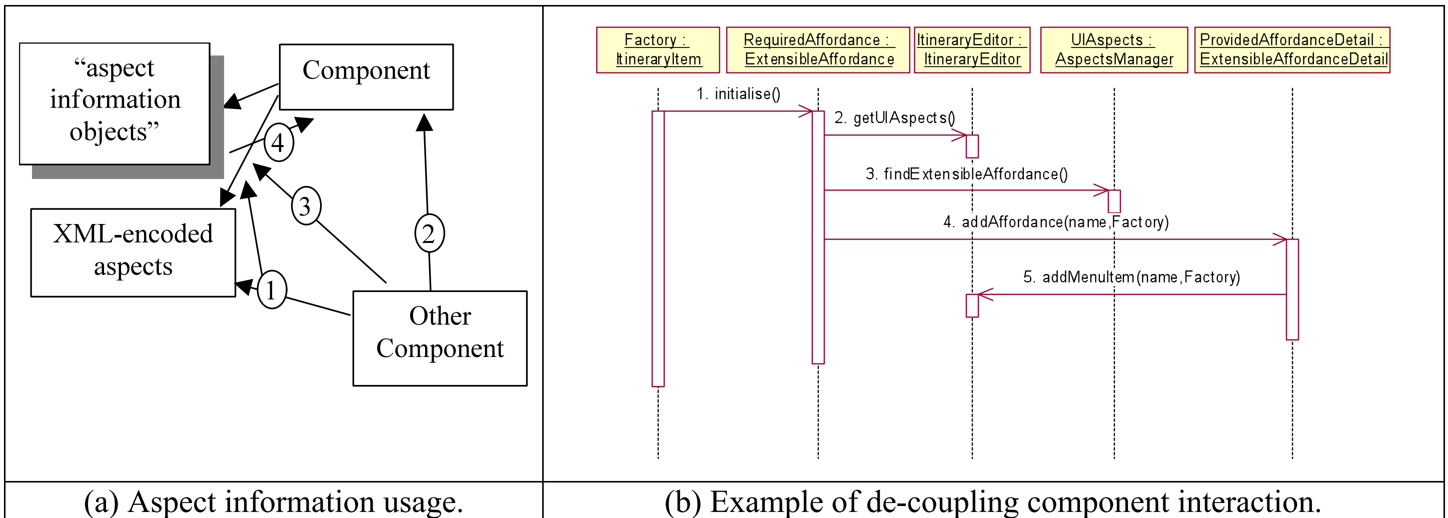
5. Component Implementation with Aspects

When implementing software components some key issues arise:

- How can reusability of components be maximised? This is desirable so the component-based development philosophy of "building systems from reusable parts" can be realised.
- How can component interaction be de-coupled so that knowledge of other component types and particular interfaces and methods can be minimised? This allows greater compositional flexibility.
- How can run-time introspection of components be supported? This allows components at run-time to be understood by other components and by developers (or even end users) who may be reconfiguring a system (for example, plugging in new components).
- How can run-time adaptation and composition be best supported?

In our work, we have used aspects to help achieve these goals by developing two approaches that make use of aspect information when implementing software components. Figure 6 (a) illustrates how information about the aspects impacting on a component can be obtained. The aspect information associated with a

component can be queried by other components (1). We have developed two ways of doing this. One is where the aspect information is encoded using a special class hierarchy of “aspect information objects”. The other is where aspect information is encoded in XML. After obtaining aspect-based information about another component, a client component can then invoke the component’s functionality. This can be done by dynamically constructing method invocations (2). It can also be done by calling “standard” adaptor methods implemented by the aspect objects (3). These translate “standard” method calls into particular component method calls (4). This approach provides a way of greatly de-coupling many common component interactions by the use of a set of “standard” aspect-oriented interactions.



(a) Aspect information usage. (b) Example of de-coupling component interaction.

Figure 6. Using aspects to de-couple components.

A simple example of using this de-coupled approach is shown in Figure 6 (b). An itinerary item factory component instance wants to add a user interface affordance to the itinerary editor’s user interface. The factory component knows nothing about how the editor’s user interface is implemented. Neither does it know how its affordance will actually be realised (menu item, button etc). Firstly, it queries its own aspect information objects for an extensible user interface affordance object (1). This queries the components related to the factory (in this case the itinerary editor) for its aspect information (2). It then locates a provided extensible affordance aspect object (3) and requests this object to add an affordance to the itinerary editor’s user interface (4). The affordance will notify the factory when it should create a new travel item of a particular kind. The provided affordance aspect object knows how to create and add this affordance for its owning itinerary editor component. In this example, it adds a menu item (5). If the factory were associated with a different component that extended its interface with buttons, a new button would be created and added. This would be done without the factory having any knowledge of how this is done. We have used this approach to implement a wide variety of software components with highly de-coupled interaction.

6. Using Aspects at Run-time

As indicated in the previous section, during component implementation aspects are codified either using special aspect objects or using XML documents. We make use of these encoded aspects in numerous ways after component deployment. This is illustrated in Figure 7. Client components obtain aspect information from a component and use this to understand the component’s provided or required services impacted by a particular systemic aspect. They can dynamically compose method calls to invoke component functionality or call aspect object methods to indirectly invoke the component’s functionality (1). We have developed a component repository that uses aspect information to index components (2). Aspect-based queries are issued by users (or even other components) to retrieve components whose functions and non-functional constraints meet those of the aspect-based query. We have developed validation agents that use aspect information to

formulate tests on a deployed component (3). The agent then compares the test results to the aspect-described component constraints and informs developers if the deployed component meets its specification or not in its current deployment context.

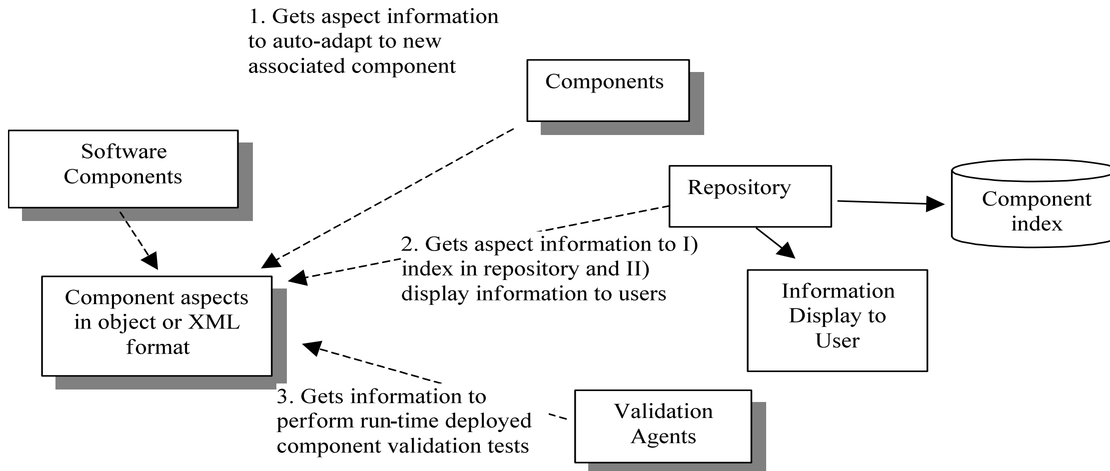
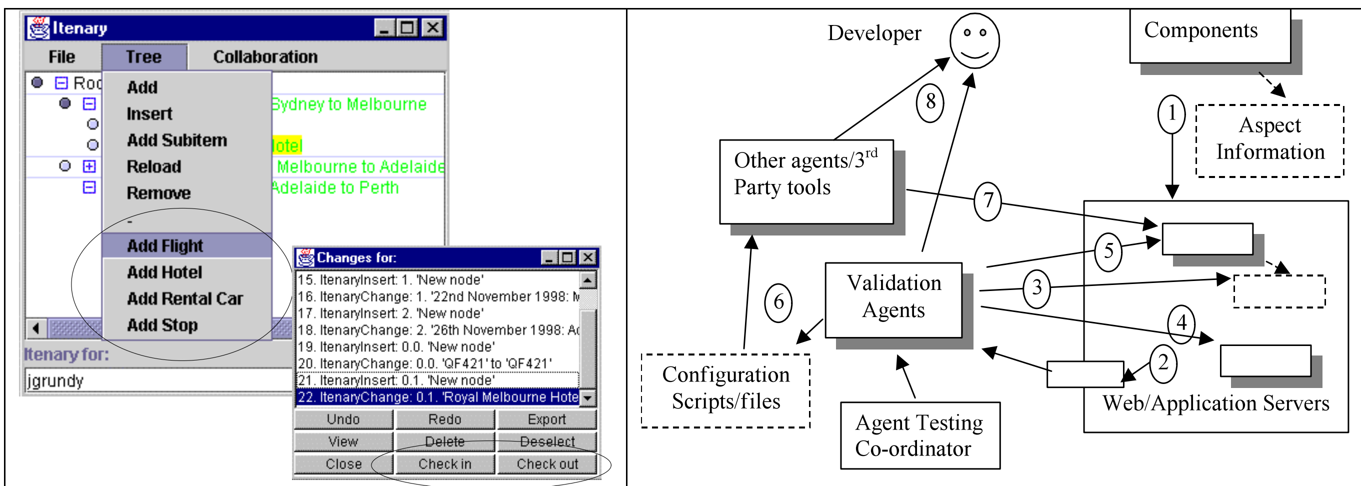


Figure 7. Examples of using aspects at run-time.

Figure 8 (a) shows how several user interface adaptations in the collaborative travel planner have been realised using dynamic discovery and invocation of component functions. The itinerary editor menu has been extended using the mechanism described in the previous section. Each itinerary item factory component obtains itinerary editor component's extensible affordance object and requests it to add (in this case) a menu item allowing the factory to be invoked by the user. The dialogue shown at the bottom of the figure is a similar example where a reusable version control component has added check-in and check-out buttons to the button panel of a reusable event history component. The version control component also obtains the distribution-providing component and persistency-supporting component of related components in its environment. It uses their facilities to store and retrieve versions and allow sharing of versions across users. The same mechanism is used to achieve this but different aspect objects are introspected and invoked.

The use of aspect information at run-time in this way is an alternative to some of the other dynamic aspect-oriented programming approaches. These use run-time code injection or modification to achieve similar results. However, usually software components are self-contained and their source code is often not available. Thus, we have tried to provide a way of dynamically changing running components by using aspects to understand component interfaces and behavioural constraints. The implementation of de-coupled component interaction by the aspect information then allows run-time adaptation via this flexible mechanism.



(a) User interface adaptation examples	(b) Deployed component validation process

Figure 8. Two examples of run-time usage of aspect information.

As a final example of the use of aspects with software components, consider the issue of checking whether deployed components are correctly configured. The operation of most software components is impacted by a variety of deployment scenario conditions, particular the other components they are deployed with. We have made use of aspect characterisations of components to enable run-time test construction and validation of component behaviour. This approach is useful because many components cannot be adequately validated until they are actually deployed.

Figure 8 (b) illustrates how we use have used aspect information to support a concept of "validation agents". These validation agents obtain aspect information about a component (1) that has been deployed in web or application servers. Different validation agents query parts of this aspect information (3) to work out the required constraints on the component's operation. Some agents also make use of deployment-specific test data (4) to formulate tests on the components. Some tests simply check the component is accessible or its functions work when invoked. Some check performance of components, transaction support or resource utilisation (memory, CPU or disk space). Some validation agents run tests (5) by invoking deployed component functionality. Others configure and deploy 3rd party testing tools (6), which run tests against the components (7) and return results to the validation agent. The agent then analyses the test results and informs the developer if the component's deployment situation means it doesn't meet its aspect-encoded constraints (8).

7. Discussion

8.1. Related Work

A great deal of work has been done to address the problem of separation of concerns in software development [16]. Examples of such work include viewpoint-based requirements, designs and tools [7, 9], subject-oriented programming [15], hyper-slices [45] and aspect-oriented programming [20, 23, 19]. Viewpoints, or partial views on parts of software artefacts, have been used for various purposes. These include requirements engineering, specification and design, user interface construction and in various software tools [7, 16]. Aspects are in essence a specialisation of the general notion of a viewpoint. An aspect captures particular cross-cutting concerns on objects or components, and thus provides a certain partial perspective on a software system design or implementation. Viewpoints of one form or another have been used in all development methods. These include the many component development methods like Catalysis™, SelectPerspective™ and COMO [1, 5, 24]. However, in almost all of the current component design methods and implementation technologies a function decomposition-centric approach is used. Such an approach results in the tangling of systemic, cross-cutting concerns in both the component designs and in their implementation code [12, 22, 28]. This is the same kind of problem that aspect-oriented programming tries to address for object-oriented programs. We have used characterisations of cross-cutting concerns to help design and implement software components in similar way to other UML extensions with aspects [43, 19]. We have used aspect information in a more novel way at run-time to provide a mechanism to dynamically understand and interact with other components.

Hyper-slices and subject-oriented programming are similar to aspect-oriented design and programming. They attempt to provide developers with alternative views of cross-cutting concerns [15, 45]. Our aspect-oriented component engineering views are specialised kinds of hyper-slices. We have deployed this viewpoint mechanism to assist component development in this work. Much recent work has gone into developing techniques to characterise software components. Our work is but one such approach. Some component development methods have introduced specialised views of component characteristics. These notably include

security and distribution issues [14, 1]. Our aspects adopt a similar approach but provide a uniform modelling approach for components' cross-cutting concerns in general. Some approaches use formal specifications of component behaviour [32]. Others make use of characterisations of services that components provide [36]. Some approaches focus on both provided and required functional component services [37]. Most now adopt varying forms of XML encoding of specifications. Our aspects provide a design, implementation-encoded and run-time accessible characterisation of software components. We have focused using common cross-cutting concerns as the "ontology" to describe components and some of their interactions. However, we feel that this approach is ultimately complimentary to other description approaches.

Little attention has so far been paid to applying aspects to component-based systems. Adaptive plug-and-play components make use of components that implement something similar to the concept of our use of aspect-based de-coupled interfaces. These are mixed to help realise the separation of various concerns from component implementations [28]. Component design methods currently provide a very limited ability to identify overlapping concerns between components. However, the isolation of systemic functions e.g. communications, database access and security, into reusable components is common in component technologies [2, 31, 47]. This partially addresses the problems of components encapsulating these systemic services. It also enables isolation of these services and access via well-defined, component-based interfaces. However, not all aspects can be suitably abstracted into individual components. This is due to overlaps and the eventual over-decomposition of systems.

One of the main motivations for the use of reflective techniques and the run-time composition and configuration of components is to try to avoid compile-time weaving [35, 48, 46]. This allows running systems and their components to have aspects imposed on them after their deployment. This is done typically as before/after method processing. Some technologies also support intra-method code incorporation and component reconfiguration at run-time. Aspects in such systems can be formulated at run-time and added or removed from programs and components dynamically. The cross-cutting concerns are encapsulated inside the introduced aspect code. Currently most code incorporation-based techniques have a high cost of expensive performance overheads. A further issue is a current lack of design abstractions. Our aspect-oriented design approach does not preclude implementation with any of these technologies. However, our aim with de-coupled interaction and introspection via aspect information was to produce software components that make use of aspect-related services in other components via well-defined component interfaces. This approach allows for controlled and efficient dynamic reconfiguration support via standardised component-supported or delegate aspect object-supported functionality access. In addition, our aspect-oriented component designs provide a consistent set of design abstractions. Our dynamic discovery approach using aspect information has some similarities with the UDDI discovery mechanism developed for web services [27]. A major difference with current UDDI registries is the ability to use categorised functional and non-functional information with our aspect-based approach.

Our use of aspect information at run-time for component repositories and deployed component validation contrasts with most other approaches. Most software repositories use type-based, keyword-based or execution-based indexing [16, 32, 49]. The component interface, comments or behaviour when executed with same data is used to index and retrieve components. Using aspects in addition to one or more of these techniques gives further perspectives on components that can be indexed and queried. Current component testing and validation techniques mainly focus on exhaustive functional interface testing [25, 14, 20, 26]. Using aspect-encoded information associated with components allows validation agents to query for expected functional behaviour and non-functional constraints. Tests can then be automatically assembled, run and feedback given to developers on whether or not a component meets its aspect-codified constraints in its current deployment situation.

8.2. Evaluation of Aspect-Oriented Component Engineering

We have used aspect-oriented component engineering on a range of problems. These have included the construction of adaptive user interfaces, multi-view software tools, plug-and-play collaborative work-supporting components and several prototype enterprise systems. We have built some of these systems using our custom JViews component architecture and some with Java 2 Enterprise Edition software components. Using aspects to assist in engineering these components has helped us to design and build more reusable and adaptable components. We have carried out a basic empirical evaluation of aspect-oriented component engineering by having a group of developers design and prototype a set of components. These included experienced industry designers and post-graduate OO technology students. Feedback from the evaluation indicated that the designers found the aspect-based perspectives on their UML designs useful. This was both when designing components and when trying to understand others people's components and their compositions. Using aspects to assist in developing de-coupled components is effective though needs good tool support. Run-time validation of software components with aspects is seen as a potentially very important long-term contribution of this work.

We have identified a number of key advantages of component development with aspect-oriented techniques.

- Cross-cutting properties of a system can be explicitly represented in design diagrams. This provides a way for developers to clearly see their impacts on the software components in a system.
- Developers can clearly see the impact of these cross-cutting concerns relating to the components in a system, as well as their impact on the component interfaces, operations and their relationships.
- Adding aspect information allows cross-cutting behaviour (aspect details) and related non-functional constraints (aspect detail properties) to be expressed together. This allows these to be more easily seen and reasoned about when building component compositions.
- Using aspect-oriented designs when implementing software components can result in greater de-coupling of components.
- Using encodings of aspect information at run-time can provide a useful approach to providing run-time adaptability and run-time accessible "knowledge" about component behaviour and constraints.

However, a number of potential disadvantages to adopting our approach also exist.

- There is possibly considerable added complexity to the specifications and designs. Some of our component designers found the additional diagrams and aspect annotations come with a high over-head. Others were unclear what aspect details and properties they should use and were unsure whether adding new aspect details and properties to their model was a good or bad thing.
- Currently we have developed limited tool support for AOCE. Our current tools are tuned to producing our custom JViews architecture's components, rather than more general J2EE or .NET components. This means that while developers can use aspect annotations in conventional CASE tools these lack formal foundations and checking. The aspect designs are not yet supported by good code generation or reverse engineering tools. Our developers identified good tool support for aspect-oriented component engineering as being very important.
- A considerable amount of effort must be put in by implementers to encode aspect information for association with software components. This can be overcome to a degree by extended tool support and by use of a component architecture that directly supports aspect encoding and de-coupled interaction.
- The use of different aspect ontology by different developers or teams is an issue. Some developers might want to make use of differently named aspect details and properties that express the same information. Of course, developers could agree on the same set of aspects. However, third-party sourced components using different characterisations would need a mapping of one ontology to another. This is a very difficult problem in general.

8.3. Future Research Directions

We are developing some further extensions to our method and prototype supporting tools to overcome some of these problems. Aspects can be prefixed with an ontology "name space" (much as can XML namespaces)

and transformations may be defined between different aspect ontologies. This is to support the translation between different component descriptions used by different people. We are investigating the use of an adaptable commercial CASE tool with notations that are more tailorable and meta-models to enable us to provide integrated support for aspects and the UML. This would also include some formal correctness checking support. In addition, we have been exploring code generation from XML-encoded component characterisations using XSLT transformation scripts. This would generate component skeleton code. We are investigating the use of new .NET reflective technologies [38] to enable efficient run-time weaving of aspect-implementing code with .NET components. This would enable us to support third party component run-time extension.

8. Conclusions

We have been working on using the concept of an "aspect" – a piece of cross-cutting systemic functionality – to clearly identify the impact of these concerns on parts of software components. Developers are then able to represent this information using augmented component specification and design diagrams. These extended component descriptions then allow them to more easily reason about inter-component provided and required functionality and constraints. We have found these aspect characterisations provide a useful way of de-coupling implemented component interaction. They also provide a useful component description approach. Using encodings of aspect information and making these available at run-time enables more sophisticated component introspection, dynamic component adaptation. It also enables better component dynamic validation, storage and plug-and-play. Developers must balance the potential advantages of this approach with the overhead of describing aspects for components.

References

1. Allen, P. and Frost, S. *Component-Based Development for Enterprise Systems: Apply the Select Perspective™,* SIGS Books/Cambridge University Press, 1998.
2. Allen, P. *Realising E-Business with Components*, Addison-Wesley, October 2000.
3. Ariniegas, F.A. Introduction to Perceval: Aspect-oriented Design using XML Schema and Groves, In *Proceedings of the 5th International Conference on Parallel and Distributed Processing Techniques and Applications: Special Session on Aspect-oriented Programming*, Las Vagas, June 26-29 2000, CSREA Press.
4. Bichler, M., Segev, A., Zhao, J.L. Component-based E-Commerce: Assessment of Current Practices and Future Directions, *SIGMOD Record* Vol. 27, No. 4, 1998, 7-14.
5. D'Souza, D.F. and Wills, A., *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
6. Fingar, P. Component-Based Frameworks for E-Commerce, *Communications of the ACM*, October 2000.
7. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B., Inconsistency Handling in Multiperspective Specifications, *IEEE Transactions on Ssoftware Engineering*, vol. 2, no. 8, 569-578, August 1994.
8. Green, T.R.G. and Petre, M., Usability analysis of visual programming environments: a 'cognitive dimensions' framework, *Journal of Visual Languages and Computing*, vol 7, 1996, pp. 131-174.
9. 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*, Vol. 42, No. 2, January 2000, pp. 117-128.
10. Grundy, J.C. Aspect-oriented requirements engineering for component-based software systems, in Proceedings of the 4th IEEE International Symposium on Requirements Engineering, Limmerick, Ireland, June 7-11 1999, IEEE CS Press.
11. Grundy, J.C. and Patel, R. Developing Software Components with the UML, Enterprise Java Beans and Aspects, In Proceedings of the 2001 Australian Software Engineering Conference, Canberra, Australia 26-29 August 2001, IEEE CS Press.
12. Grundy, J.C. Multi-perspective specification, design and implementation of software components using aspects, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 10, No. 6, December 2000.
13. Grundy, J.C. and Hosking, J.G. Engineering plug-in software components to support collaborative work, Software – Practice and Experience, vol. 32, Wiley, pp. 983-1013.
14. Han, J. Zheng, Y. Security characterisation and integrity assurance for component-based software. In *Proceedings of the 2000 International Conference on Software Methods and Tools*, IEEE CS Press.
15. Harrison, W. and Ossher, H. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications,* September 1993.
16. Harrison, W., Ossher, H. and Tarr, P. Software Engineering Tools and Environments: A Roadmap, *The Future of Software Engineering*, Finkelstein, A. Ed., ACM Press, 2000.
17. Henninger, S. Supporting the Construction and Evolution of Component Repositories, In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996, IEEE CS Press, pp. 279-288.
18. Herzum, P. and Sims, O. *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise*, Wiley, December 1999.

19. Ho, W.M., Pennaneach, F., Jezequel, J.M., and Plouzeau, N. Aspect-Oriented Design with the UML, In *Proceedings of the ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland, June 6 2000.
20. Hoffman, D., Strooper, P. Tools and techniques for Java API testing. In *Proceedings of the 2000 Australian Software Engineering Conference,*IEEE CS Press, pp.235-245.
21. Iseminger, D. Com+ Developer's Reference Library, Microsoft Press, September 2000.
22. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J. Aspect-oriented Programming, In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, Finland, June 1997, Springer-Verlag, LNCS 124.
23. Kiczales, G. and Lopes, C. Recent developments in AspectJ, In *Proceedings of the ECOOP'98 Workshop on Aspect-oriented Programming*, Brussels, July 1998.
24. Lee, S.D., Yang, Y.J., Cho, F.S., Kim, S.D., and Rhew, S.Y., COMO: a UML-based component development methodology, In *Proceedings of the Sixth Asia-Pacific Software Engineering Conference*, Takamatsu, Japan, 7-10 Dec. 1999, IEEE CS Press, pp. 54-61.
25. Ma, Y-S. Oh, S-U. Bae, D-H., Kwon, K-R. Framework for third party testing of component software. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference*, IEEE CS Press, 2001, pp.431-434.
26. McGregor, J.D. Parallel Architecture for Component Testing. Journal of Object-Oriented Programming, vol. 10, no. 2 (May 1997), SIGS Publications, pp.10-14.
27. McKinlay, M., Tari, Z.. DynWES - a dynamic and interoperable protocol for Web services. In *Proceedings of the Third International Symposium on Electronic Commerce*, IEEE CS Press, 2002, pp.74-83.
28. Mezini, M. and Lieberherr, K. Adaptive Plug-and-Play Components for Evolutionary Software Development, In *Proceedings of OOPSLA'98*, Vancouver, WA, October 1998, ACM Press, pp. 97-116.
29. Mezini, M., Seiter, L. and Lieberherr, K. Component Integration with Pluggable Composite Adapters. *Software Architectures and Component Technology: The State of the Art in Research and Practice*, Kluwer, 2000.
30. Mahmoud, Q. Securing Component-Based E-Commerce Applications, *Component Advisor Magazine*, March 2000.
31. Monson-Haefel, R., Enterprise JavaBeans, O'Reilly, 1999.
32. Motta, E., Fensel, D., Gaspari, M., Benjamins, R. Specifications of Knowledge Components for Reuse, In *Proceedings of 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 36-43.
33. Ossher, H. and Tarr, P. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2000.
34. Perronel,P. Chaganti, K. *Building Java Enterprise Systems with J2EE*, Sams, June 2000.
35. Pryor, J.L. and Bastan, N.A. 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.
36. Qiong, W., Jichuan, C., Hong, M., and Fuqing, Y. 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.
37. 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.
38. Richter, J. *Applied Microsoft .NET Framework Programming*, Microsoft Press, January 2002.
39. Rittri, M. Using types as search keys in function libraries, *Journal of Functional Programming,* 1 (1), 1991, 71-89.
40. Rollins, E., Wing, J. Specifications as search keys for software libraries, In *Proceedings of the International Conference on Logic Programming*, 1991, MIT Press, pp. 173-187.
41. Sessions, R., *COM and DCOM: Microsoft's vision for distributed objects*, Wiley, 1998.
42. Stearns, M. and Piccinelli, G. Managing interaction concerns in web-service systems, In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, Vienna, Austria, July 2002, IEEE CS Press.
43. Stein, D., Hanenberg, S., Unland, R. An UML-based Aspect-Oriented Design Notation, In Proceedings of 2002 Conference on Aspect-oriented Software Development.
44. Szyperski, C.A., *Component Software: Beyond OO Programming*, Addison-Wesley, 1997.
45. Tarr, P., Ossher, H., Harrison, W. and Sutton, S.M. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE 21),* May 1999.
46. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Joergensen, B. Dynamic and Selective Combination of Extensions in Component-based Applications, In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001, Toronto, Canada, IEEE CS Press.
47. Vogal, A. CORBA and Enterprise Java Beans-based Electronic Commerce, In *International Workshop on Component-based Electronic Commerce*, UC Berkeley, 25th July, 1998.
48. Welch, I. and Stoud, R. Load-time application of aspects to COTS, in Proc. of the ECOOP'99 Workshop on Aspect-oriented Programming, Lisbon, Portugal, June 1999.
49. Ye, Y. and Fischer, G. Context-Aware Browsing of Large Component Repositories, In *Proceedings of the 2001 Automated Software Engineering Conference*, San Diego, CA, Nov 26-28 2001, IEEE CS Press.
50. Zhao, J.L. and Misic, V.B. Toward Component-Based Electronic Commerce: A Workflow Perspective, In *International Workshop on Component-based Electronic Commerce*, UC Berkeley, 25th July, 1998.