CHAPTER 24

Developing Software Components with Aspects: Some Issues and Experiences

John Grundy and John Hosking

Engineering software components is a challenging task. Existing approaches to component-based software development are for the most part focused on functional decomposition. All have the weakness of failing to take into account the impact of crosscutting concerns on components. In this chapter, we outline *aspect oriented component engineering*. Our approach uses aspects to help engineer better software components. Motivating our work with a simple example of a distributed system, we describe how speci-

fications and designs can use aspects to provide additional information about components and how aspects can be used to help implement more decoupled 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.

24.1. INTRODUCTION

Component-based systems development is the composition of systems from parts, called software components. Components encapsulate data and functions. They often provide events, are self-describing, and many can be dynamically "plugged and played" into running applications [1,7, 36]. In building systems, we often use a mixture of newly built and existing COTS (Commercial Off-The-Shelf) components. For the later, we usually have no access to source code.

24.1. Introduction

Engineering software components can be quite a challenging task. Components must be identified and their requirements specified. Component interaction is crucial, so both provided and required component behavior needs identification and documentation [32, 18]. Ideally, components are implemented using a 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 and implementation technologies. In our experience, they do not produce components with sufficiently flexible interfaces, run-time adaptability, or good-enough documentation [11, 12]. A major weakness of current methodology is the inability to describe functional and non-functional characteristics and inter-relationships of the components.

In the past, we used *aspects* (crosscutting concerns) at the requirements level to improve the description of our components [10]. When this proved successful, we applied the concept to component design and implementation [11, 12]. This involves using aspects to better describe the impact of cross-

cutting concerns on components at the design level. We have made 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 [12]. 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 the way we do. Aspects are used to identify and codify crosscutting concerns on objects. Similarly, some reflective systems use aspect information to support run-time adaptation [25, 30]. Most aspect programming systems weave code into join-points of programs [20]. Some design approaches use aspects (or "viewpoints" or "hyper-slices") to provide multiple perspectives ont the object designs [8, 16, 18, 37].

In this chapter, we provide a summary of our work applying aspects to the development of software components. We refer readers interested in a more comprehensive discussion of our work to our previously published papers [10, 11, 12, 14].

24.2. MOTIVATION

Consider a collaborative travel planning application that is to be used by customers and travel agents to make travel bookings [12]. Examples of the user interfaces provided by such a system are illustrated in Figure 24-1 (a). Some of the software components composed to form such an application are illustrated in Figure 24-1 (b).

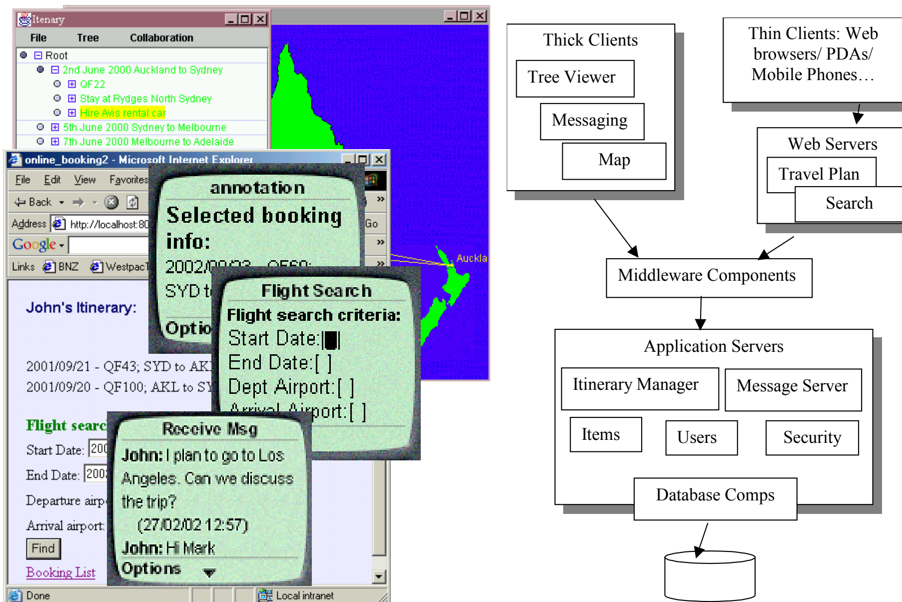


Figure 24-1 Example component-based application.

We built this component-based system by composing a set of software components that provide the necessary facilities. These include travel itinerary management, customer and staff data management, system integration with remote booking systems, and various user interfaces. Some components, such as the map visualization, database, and email server are quite general and highly reusable. 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 many components. These have usually been built using “functional decomposition”: organizing 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 crosscutting many of the different components in the system [10, 20]. For example, user interfaces, data persistency, data distribution, security management, and resource utilization all have pandemic impact. Some components provide such functional-

ities; others require them [3]. We use the term “aspects” to describe these crosscutting, horizontally impacting concerns.

To illustrate how systemic aspects affect components, Figure 24-2 shows three components from the travel planner system: Tree viewer, Travel itinerary and Database. Aspects User Interface, Persistency, Collaborative Work and Transactions crosscut these 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 coordination support but requires transaction coordination. The three components must work together to provide the travel plan viewing, business process and data management required by the system. Note that several aspects affect each of these components in different ways.

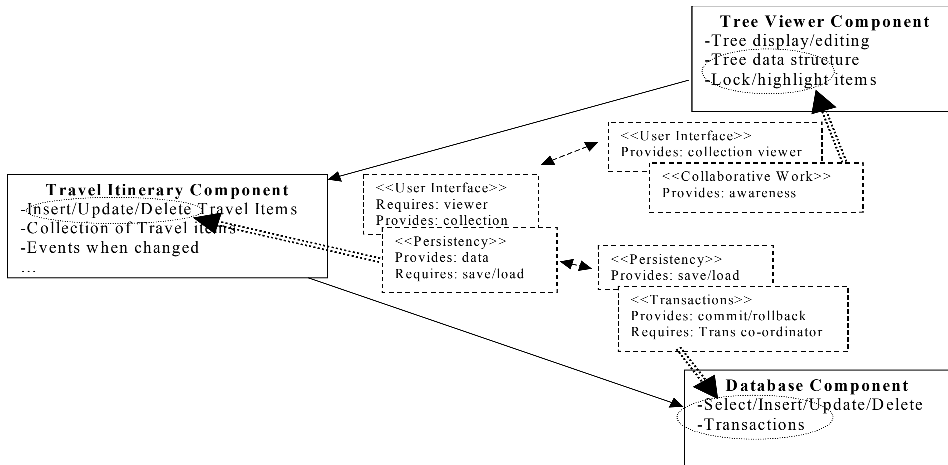


Figure 24-2. The concept of aspects crosscutting inter-related software components.

For each aspect that affects a component, we need to provide additional information, a set of *aspect details*. 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 assert that the Persistency aspect affects the Itinerary manager. 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.

In aspect-oriented programming languages [20], aspects support code injection into methods. For example, in AspectJ point-cuts can specify where to add persistency management, memory utilization, user interface and distributed communication code [6]. In aspect-oriented design [3, 18, 34, 35], aspects are used to describe crosscutting concerns affecting the components. In dynamic aspect-oriented programming [30, 38, 40], components might be modified at run-time using the aspects to change their parameters or their running code.

24.3. OUR APPROACH

We have developed aspect-oriented component engineering, a new method for developing software components with aspects. The use of aspects provides us with "multiple perspectives" on software component designs. Figure 24-1 illustrates our approach. Component specifications and designs, typically UML diagrams, are augmented with aspect information (1). A key activity is determining 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 a more decoupled component interaction and dynamic component configuration. This enables us to maximize 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 reconfiguration 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).

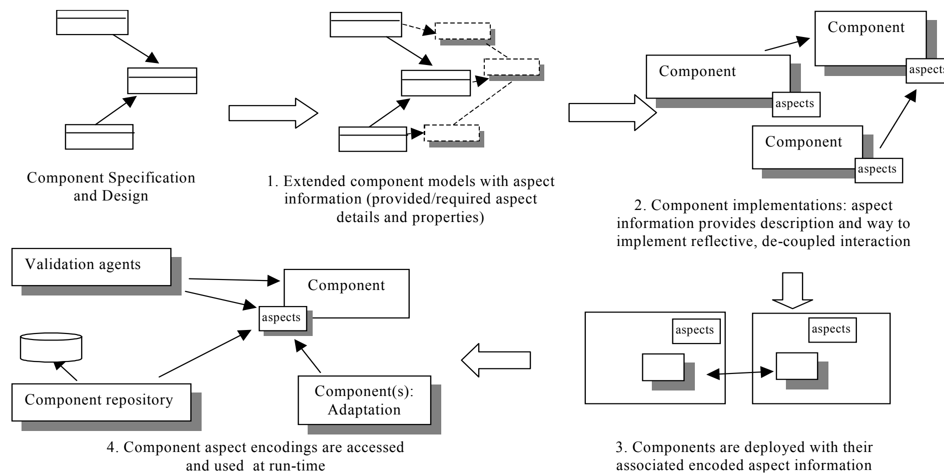


Figure 24-3 An overview of aspect-oriented component engineering.

In the following three sections, we briefly illustrate how we use aspects for component design, decoupled and configurable component implementation, and at run-time.

24.4. COMPONENT SPECIFICATION AND DESIGN WITH ASPECTS

Our approach gives developers a way to capture crosscutting 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 [1, 8, 13]. Using aspects gives developers a way to categorize the impact of these concerns on 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 include User interface, Collaborative work, Component configuration, Security, Transaction processing, Distribution, Persistency and Resource management. Domain-specific aspects can also be used. In the example domain, these include ser-

vices relating to Travel itinerary management, Payment and Order processing.

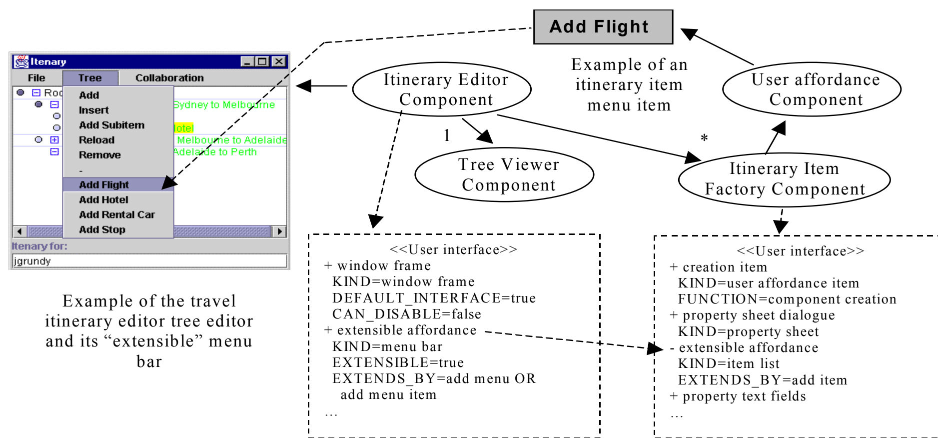


Figure 24-4 A simple component specification example using aspects.

Figure 24-4 illustrates a simple use of aspects when specifying two interrelated software components. In this example, the travel planner's requirements identify several components that must provide extensible user interfaces, where one component provides a user interface that another adapts this at run-time. This adaptation is usually the structured addition of a new user interaction "affordance". 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 24-4 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 provide (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 for creating 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.

During design, developers refine component specifications into detailed designs and then design 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 affecting a group of related components. An example from the travel planner system is shown in Figure 24-5, an annotated Unified Modeling 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 affecting each component. Both method invocations and component objects have been annotated in this way. We have used UML note annotations to further characterize particular method invocations between components. These indicate provided and requires aspect details. Some details are also

characterized 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.

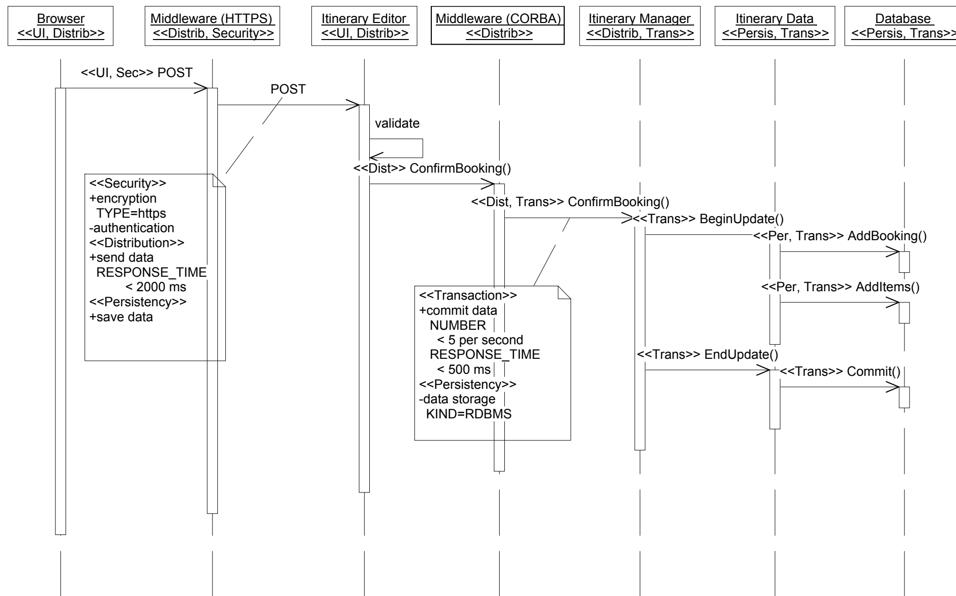


Figure 24-5 Component interaction design example.

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. Our aspects

give developers a way to capture and document both functional and non-functional constraints during design.

24.5. COMPONENT IMPLEMENTATION WITH ASPECTS

When implementing software components some key issues arise:

How can reusability of components be maximized? This is desirable to realize the component-based development philosophy of "building systems from reusable parts."

How can component interaction be decoupled, thereby minimizing the knowledge required of other components, interfaces, and methods? 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? This allows dynamically evolving systems.

In our work, we have used aspects to help achieve these goals. We have developed two approaches that make use of aspect information when implementing software components. Figure 24-6 (a) illustrates how information about the aspects affecting a component can be obtained.

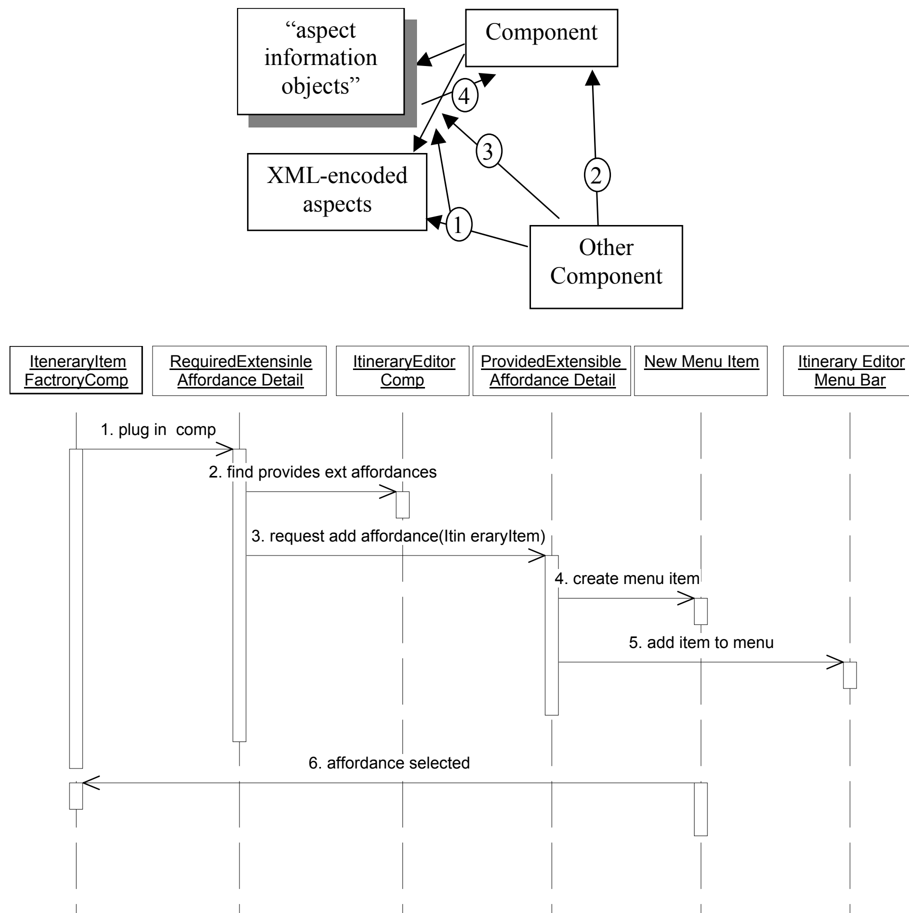


Figure 24-6 Using aspects to de-couple components.

The aspect information associated with a component can be queried by other components (1). We have developed two ways of doing this. One has the

aspect information encoded using a special class hierarchy of "aspect information objects." The other has aspect information 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). Alternatively, it can 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 decoupling many common component interactions by the use of a set of standard aspect-oriented interactions.

Figure 24-6 (b) shows a simple example of using this decoupled approach. 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 realized (menu item, button, etc). First, it tells its own aspect information to initialize after plug in (1). Its required extensible affordance aspect object queries the components related

to the itinerary item factory (e.g., the itinerary editor) for its aspect information (2). It then locates a provided extensible affordance aspect object and requests this object to add an affordance to the itinerary editor's user interface (3). This creates a menu item (4) and adds it to the itinerary editor's menu bar at an appropriate place (5). The menu item notifies the factory when it should create a new travel item of a particular kind (6). 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. 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 decoupled interactions.

24.6. 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 many ways after component deployment. This is illustrated in Figure 24-7. Client components obtain aspect information from a component and use this to understand the component's provided or required services affected 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

whether the deployed component meets its specification in its current deployment context.

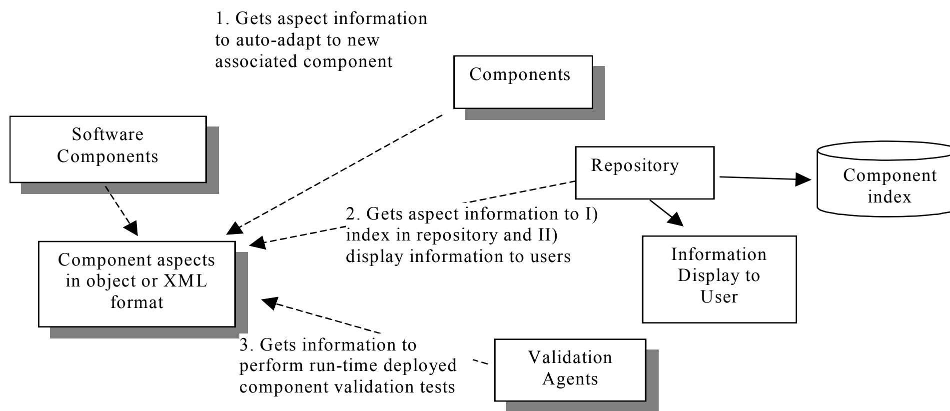
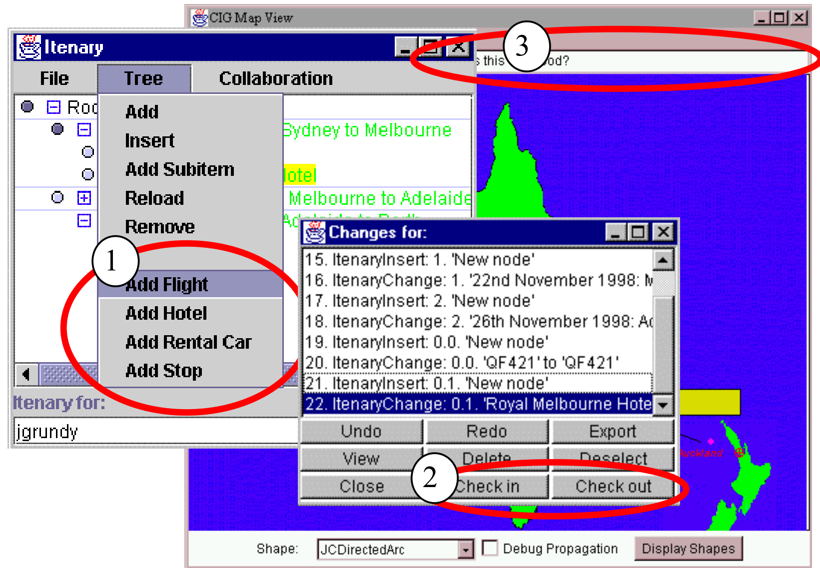


Figure 24-7 Examples of using aspects at run-time.

Figure 24-8 (a) shows how several user interface adaptations in the collaborative travel planner have been realized using dynamic discovery and invocation of component functions. The itinerary editor menu has been extended using the mechanism described in the previous section (1). 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 (2). 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 to allow sharing of versions across users. The same mechanism is used to achieve this but different aspect objects are introspected and invoked. The map viewer has had a collaborative messaging bar added to it dynamically via the same mechanism (3).

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 behavioral constraints. The implementation of decoupled component interaction by the aspect information allows run-time adaptation.



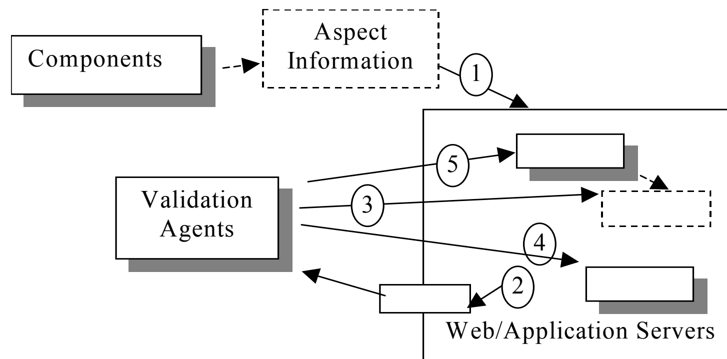


Figure 24-8 Two examples of run-time usage of aspect information.

As a final example of the use of aspects with software components, consider the issue of checking whether deployed components are correctly config-

ured. The operation of most software components is affected by a variety of deployment scenario conditions, particularly the other components they are deployed with. We make use of aspect characterizations of components to enable run-time test construction and validation of component behavior. This approach is useful because many components cannot be adequately validated until they are actually deployed.

Figure 24-8 (b) illustrates how we use 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 (2). 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 that the component is accessible or that its functions work when invoked. Some check performance of components, transaction support or resource utilization (memory, CPU or disk space). Some validation agents run tests (5) by invoking deployed component functionality.

24.7. RELATED WORK

A great deal of work has been done on separation of concerns in software development [16]. Examples of such work include viewpoint-based requirements, designs and tools [8, 13], subject-oriented programming [16], hyperslices [37] and aspect-oriented programming [18, 20, 30]. Viewpoints, or partial views on parts of software artifacts, have been used for purposes such as requirements engineering, specification and design, user interface construction and in various software tools. Aspects are in essence a specialization 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™, Select-Perspective™ and COMO [2, 4, 21]. 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 tan-

gling of systemic, crosscutting concerns in both the component designs and in their implementation code [11, 20, 25]. This is the same kind of problem that aspect-oriented programming tries to address for object-oriented programs. In contrast with these other approaches, we have used characterizations of crosscutting concerns to help design and implement software components in similar way to other UML extensions with aspects [34]. 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 [16, 29, 37]. They attempt to provide developers with alternative views of crosscutting concerns. Our aspect-oriented component engineering views are specialized 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 characterize software components. Our work is but one such approach. Some component development methods have introduced specialized views of component characteristics. These notably include security and distribution issues [15]. Our

aspects adopt a similar approach but provide a uniform modeling approach for components' crosscutting concerns in general. Some approaches use formal specifications of component behavior [28]. Others make use of characterizations of services that components provide [31]. Some approaches focus on both provided and required functional component services [19, 32]. Our aspects provide a design, implementation-encoded and run-time accessible characterization of software components. We have focused using common crosscutting 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.

So far, little attention has been paid to applying aspects to component-based systems. Adaptive plug-and-play components and composite adapters [25, 26] make use of components that implement something similar to the concept of our use of aspect-based decoupled interfaces. These are mixed to help realize the separation of various concerns from component implementations. Component design methods currently provide a very limited ability to identify overlapping concerns between components. However, the isolation

of systemic functions, (for example, communications, database access and security), into reusable components is common in component technologies [27, 32, 39]. 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, though some success has been achieved with middleware-supporting components [3]. This is due to overlaps and the eventual over-decomposition of systems. JAsCo [35] provides a component-based development method incorporating aspects which uses the concept of aspect beans and connectors to extend Java Beans for aspect-oriented composition. This approach is similar to our component aspects, but specialized for a JavaBean-based development platform.

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 [30, 38, 40]. This allows running systems and their components to have aspects imposed on them after 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 crosscutting concerns are encapsulated within 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 decoupled 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 standardized 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 [24]. A major difference with current UDDI registries is the ability to use

categorized 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 [17]. The component interface, comments, or behavior, when executed with same data, is used to index and retrieve components. Run-time adaptation can also be very well-supported by aspects, illustrated in our own work and that of others more recently [5, 38]. 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 [22, 23]. Using aspect-encoded information associated with components allows validation agents to query for expected functional behavior 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.

24.8. EVALUATION

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 decoup-

led components is effective though needs good tool support. Run-time validation of software components with aspects is potentially an important long-term contribution of this work.

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

Crosscutting properties of a system can be explicitly represented in design diagrams. This provides a way for developers to see the impact of these crosscutting concerns on their components, and the components interfaces, operations and relationships.

Adding aspect information allows crosscutting behavior (aspect details) and related non-functional constraints (aspect detail properties) to be expressed together. This allows these to be more easily understood and reasoned about when building component compositions.

Using aspect-oriented designs when implementing software components can result in greater decoupling 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 behavior and constraints.

However, there are also several potential disadvantages to our approach.

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 limited tool support for AOCE. Our 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.

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 decoupled interaction.

The use of different aspect ontologies 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 characterizations would need a mapping of one ontology to another. This is a difficult problem to solve in general.

24.9. Future Research Directions

We are developing 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 transforma-

tions may be defined between different aspect ontologies. This supports translation between different component descriptions. We are investigating the use of an adaptable commercial CASE tool with notations that are more tailorable, and meta-models to enable 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 characterizations using XSLT transformation scripts. This would generate component skeleton code. We are investigating the use of new .NET reflective technologies to enable efficient run-time weaving of aspect-implementing code with .NET components. This would enable supporting third party component run-time extensions.

24.10. CONCLUSIONS

We have been working on using the concept of an “aspect”—a piece of crosscutting systemic functionality—to clearly identify the impact of these concerns on parts of software components, and on providing tools to enable

developers to represent aspects using augmented component specification and design diagrams. These extended component descriptions allow developers to more easily reason about inter-component provided and required functionality and constraints. We have found these aspect characterizations provide a useful way of decoupling implemented component interaction. They also provide a practical component description approach. Using encodings of aspect information and making these available at run-time enables more sophisticated component introspection and dynamic component adaptation. It also enables doing 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. 2000. *Realizing E-Business with Components*. Addison-Wesley, Reading, Massachusetts.

2. ALLEN, P. AND FROST, S. 1998. *Component-based Development for Enterprise Systems: Applying the SELECT Perspective*. Cambridge University Press, Cambridge, UK.

3. COYLER, A. AND CLEMENT, A. Large-scale AOSD for Middleware, In *2004 Int'l Conf. Aspect-Oriented Software Development* (Lancaster, UK). ACM, 56-65.

4. D'SOUZA, D. F. AND WILLS, A. C. 1999. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Massachusetts.

5. DUZAN, G., LOYALL, J. AND SCHANTZ, R. 2004. Building adaptiveadaptive distributed applications with middleware and aspects, In *Int'l Conf. Aspect-Oriented Software Development* (Lancaster UK). ACM, 66-73.

6. ERNST, E. AND LORENZ, D.H. 2003. Aspects and Polymorphism in AspectJ. In *Proc. 2003 Aspect-oriented Software Development Conf.* (Boston, MA). ACM, 150-157.

7. FINGAR, P. 2000. Component-based frameworks for e-commerce. *Comm. ACM 43*, 10, 61–67.

8. FINKELSTEIN, A. C. W., GABBAY, D., HUNTER, A., KRAMER, J., AND NUSEIBEH, B. 1994. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering 20*, 8, 569–578.

9. GREEN, T. AND PETRE, M. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing 7*, 131–174.

10. GRUNDY, J. 1999. Aspect-oriented requirements engineering for component-based software systems. In *4th IEEE Int'l Symp. Requirements Engineering* (Limerick). IEEE, 84–91.

11. GRUNDY, J. 2000. Multi-perspective specification, design and implementation of software components using aspects. *Int'l Journal of Software Engineering and Knowledge Engineering 20*, 6, 713–734.

12. GRUNDY, J. AND HOSKING, J. 2002. Engineering plug-in software components to support collaborative work. *Software Practice and Experience 32*, 10, 983–1013.

13. GRUNDY, J., MUGRIDGE, W., AND HOSKING, J. 2000. Constructing component-based software engineering environments: Issues and experiences. *Journal of Information and Software Technology 42*, 2 (Jan.), 117–128.

14. GRUNDY, J. AND PATEL, R. 2001. Developing software components with the UML, Enterprise Java Beans and aspects. In *13th Australian Conf. Software Engineering*, (Canberra). IEEE, 127–136.

15. KHAN, K.M., HAN, J. 2003. A Security Characterisation Framework for Trustworthy Component Based Software Systems. In *Proc. COMPSAC 2003* (Dallas, TX). IEEE, 164-169.

16. HARRISON, W. AND OSSHER, H. 1993. Subject-oriented programming—a critique of pure objects. In *8th Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (Washington, D. C.). ACM, 411–428.

17. HENNINGER, S. 1996. Supporting the construction and evolution of component repositories. In *18th Int'l Conf. Software Engineering (ICSE)* (Berlin). IEEE, 279–288.

18. HO, W.M., JÉZÉQUEL, J.M., PENNANEAC'H, F., PLOUZEAU, N. 2002. A toolkit for weaving aspect oriented UML designs, In *2002 Int'l Conf. Aspect-Oriented Software Development* (Enschede, The Netherlands), ACM, 99-105.

19. KATARA, M. AND KATZ, S. 2003. Architectural Views of Aspects, In *2003 Int'l Conf. Aspect-Oriented Software Development* (Boston, MA). ACM, 1-10.

20. KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming, 11th European Conf.*, M. Akşit and S. Matsuoka, Eds. LNCS, vol. 1241. Springer-Verlag, Berlin, 220–242.

21. LEE, S., YANG, Y., CHO, F., KIM, S., AND RHEW, S. 1999. COMO: A UML-based component development methodology. In *6th Asia-Pacific Software Engineering Conf. (APSEC),* (Takamatsu, Japan). IEEE, 54–61.

22. MA, Y.-S., OH, S.-U., BAE, D.-H., AND KWON, K.-R. 2001. Framework for third party testing of component software. In *8th Asia-Pacific Software Engineering Conf. (APSEC),* (Macau, China). IEEE, 431–434.

23. MCGREGOR, J. D. 1997. Parallel architecture for component testing. *Journal of Object-Oriented Programming 10*, 2 (May), 10–14.

24. MCKINLAY, M. AND TARI, Z. 2002. DynWES — a dynamic and interoperable protocol for web services. In *3rd Int'l Symp. Electronic Commerce (ISEC)*, (Research Triangle Park, North Carolina). http://ecommerce.ncsu.edu/ISEC/papers/09_tari_dynamic.pdf.

25. MEZINI, M. AND LIEBERHERR, K. 1998. Adaptive plug-and-play components for evolutionary software development. In *13th Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA),* (Vancouver). ACM, 97–116.

26. MEZINI, M., SEITER, L., AND LIEBERHERR, K. 2001. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology*, M. Akşit, Ed. Kluwer Academic Publishers, Boston, 325–356.

27. MONSON-HAEFEL, R. 2001. *Enterprise JavaBeans*. 3rd Edition, O'Reilly, California.

28. MOTTA, E., FENSEL, D., GASPARI, M., AND BENJAMINS, R. 1999. Specifications of knowledge components for reuse. In *11th Int'l Conf. Software Engineering and Knowledge Engineering (SEKE)* (Kaiserslautern, Germany). Knowledge Systems Institute, Skokie, Illinois, 36–43.

29. OSSHER, H. AND TARR, P. 2001. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology*, M. Akşit, Ed. Kluwer Academic Publishers, Boston, 293–323.

30. PRYOR, J. L. AND BASTAN, N. A. 2000. Java meta-level architecture for the dynamic handling of aspects. In *5th Int'l Conf. Parallel and Distributed Processing Techniques and Applications* (Las Vegas). CSREA Press, Bogart, Georgia, 257–262..

31. QIONG, W., JICHUAN, C., HONG, M., AND FUQING, Y. 1997. JBCDL: An object-oriented component description language. In *24th Conf. Technology of Object-Oriented Languages (TOOLS)*, (Beijing). IEEE, 198–205.

32. RAKOTONIRAINY, A. INDULSKA, J. WAI LOKE, S. 2001. ZASLAVSKY, A. Middleware for Reactive Components: An Integrated Use of Context, Roles, and Event Based Coordination, In *Proc*eedings of. *Middleware 2001* (Heidelberg, Germany). LNCS 2218, Springer,77-98.

33. STEARNS, M. AND PICCINELLI, G. 2002. Managing interaction concerns in web-service systems. In *22nd Int'l Conf. Distributed Computing Systems Workshops* (Vienna, Austria). IEEE, 424–429.

34. STEIN, D., HANENBERG, S., AND UNLAND, R. 2002. An UML-based aspect-oriented design notation. In *1st Int'l Conf. Aspect-Oriented Software Developmen),* (Enschede, The Netherlands), G. Kiczales, Ed. ACM, 106–112.

35. SUVIE, D. AND VANDERPERREN, W. 2003. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development, In.*2003 Int'l Conf. on Aspect-Oriented Software Development*, (Boston, MA), ACM, 21-29.

36. SZYPERSKI, C. A. 1997. *Component Software: Beyond OO Programming*. Addison-Wesley, Reading, Massachusetts.

37. TARR, P., OSSHER, H., HARRISON, W., AND SUTTON JR., S. M. 1999. *N* degrees of separation: Multi-dimensional separation of concerns. In *21st Int'l Conf. Software Engineering (ICSE)* (Los Angeles). IEEE, 107 – 119.

38. TRUYEN, E., VANHAUTE, B., JOOSEN, W., VERBAETEN, P., AND JØRGENSEN, B. N. 2001. Dynamic and selective combination of extensions in component-based applications. In *23rd Int'l Conf. Software Engineering (ICSE)* (Toronto). IEEE, 233–242.

39. VOGAL, A. 1998. CORBA and Enterprise Java Beans-based electronic commerce. In *Int'l Workshop on Component-based Electronic Commerce* (Berkeley).

40. WELCH, I. AND STROUD, R. 1999. Load-time application of aspects to Java COTS software. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP)* (Lisbon). http://trese.cs.utwente.nl/aop-ecoop99/papers/welch.pdf.