Copyright 2000 IEEE. Published in the Proceedings of 2001 Australian Software Engineering Conference, Canberra, Australia. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

Developing Software Components with the UML, Enterprise Java Beans and Aspects

John Grundy[1] and Rakesh Patel[2]

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

[2]Hewlet-Packard Ltd
Auckland, New Zealand

Abstract

Component-based systems have become increasingly popular approaches to developing complex systems, offering well-formed abstractions, strong potential for reuse, dynamic plug-and-play and sometimes end-user application enhancement. Unfortunately the design, implementation and deployment of components is very challenging, particularly achieving appropriate division of responsibility among components, designing components and implementing components. We have developed the Aspect-Oriented Component Engineering method to help improve component development by the use of aspects during component specification, design, implementation and deployment. We describe our recent work extending the UML to facilitate aspect-oriented component design and the use of Enterprise Java Beans to implement these designs.

Keywords: aspect-oriented design, component-based systems, UML, Enterprise Java Beans, XML

1. Introduction

Developing systems with software components involves identifying reusable "building block" (i.e. component) abstractions, combining (i.e. composing) multiple components in appropriate ways, configuring components for different reuse situations, and, sometimes, allowing end-users of applications to further plug-and-play components at run-time [6, 23]. It turns out that engineering component-based systems is non-trivial: developers must identify appropriate component abstractions, allocate responsibility to components carefully so minimum duplication or inconsistency occurs, and must take care when combining and configuring components [1, 2, 5, 25]. As with other design and programming approaches, tangling of concerns, particularly in regard to management of non-functional characteristics, usually occurs [14, 18, 9].

Many component technologies and development methods have been developed to try and aid component developers. Examples of component technologies include Enterprise Java Beans, COM+, CORBA's C-IDLs and JViews [4, 8, 21, 22]. While the Unified Modelling Language (UML) has become a de-facto standard for most object-oriented development, the standard UML model and process lacks support for software component development [5, 14]. This has led to the development of component-specific engineering methods, many of which extend the UML to incorporate component representation. Examples of component development methods include Select Perspective™, Catalysis™ and COMO [1, 5, 17]. Most component development methods provide limited guidance and notational support for capturing non-functional, cross-cutting properties of components, however, leading (typically) to continuing tangling of concerns at design and implementation time [9, 3, 14].

Aspect-Oriented Component Engineering (AOCE) addresses the identification and use of cross-cutting aspects of software components [9]. We use the concept of component aspects (broad categories of cross-cutting systemic properties, e.g. persistency, distribution, user interfaces and security), to help component developers identify and reason about provided and required aspect details and their property value constraints during component design and implementation. We describe our recent research investigating the addition of component aspects to the Unified Modelling Language (UML). We also describe recent work investigating the implementation of these aspect-oriented UML component designs using Enterprise Java Beans, including partial generation of components from the XML-encoded Perceval aspect-oriented language.

We first present a motivating example for AOCE, a simple E-commerce system and then explain the key ideas of AOCE in relation to characterising components in this E-commerce application. We describe aspect-based extensions to the UML to facilitate aspect-oriented component design, and describe how these designs can be mapped onto several component implementation technologies, including our own JViews, the Enterprise

Java Beans (EJB) architecture, and the Perceval XML aspect language. We discuss some prototype tool support for aspect-oriented component design and implementation, including generation of EJBs from Perceval specificatiopns. We compare and contrast this work to related work in component development, multiple separation of concerns and aspect-oriented programming.

2. Motivation

Component-based systems (or "componentware") focus on building applications by composing discrete, reusable components. Components provide well-defined interfaces, embody data and processing, often provide run-time reflection mechanisms, utilise various interconnection mechanisms, including subscribe-notify, are often dynamically deployable and configurable, and ideally are highly reusable via parameterisation [1, 5, 23].

As an example, consider a video store library built from components. This system is to provide customers on-line search, review and reserve functionality and staff data maintenance, reporting and rent/return functions. Customers and staff can communicate via messages. Figure 1 shows some of the user interfaces for such a system. Traditional object-oriented analysis and design identifies a set of usually domain-specific object abstractions embodying the data and functions this system will provide. Typically these have limited reusability, can not be dynamically deployed and reconfigured, and communicate via fixed method calling protocols [13, 25].

A component-oriented design tries to reuse existing components and new components are designed for maximal reuse (via configuration facilities, dynamic plug-and-play and de-coupled interaction). A component-based architecture for a prototype of this system we developed is outlined in Figure 1. The customer user interface includes fine-grained GUI components (buttons, text fields etc) and a coarser-grained tree viewer, showing search results, and reusable search panel, composed to form the video search interface. Various middleware components (database, communications, security, transaction processing support) are reused. Server-side components include generalised data management e.g. customer and staff, "product" (configured to represent "videos" for this application by dynamically specifying video-specific field names and types) and "rental" (usable in any system where a "customer" rents a "product"). A search engine component provides retrieval of "products", and "rental processing" is business logic encapsulation. Groupware support includes reviews and messages.

While component-based solutions have become popular for such E-commerce and other systems development projects [2, 4, 6, 26], many challenging issues arise when engineering such applications. When designing components developers must determine appropriate divisions of responsibility, must identify component functional non-functional characteristics, and must design general component inter-communication interfaces [1, 9, 5]. When implementing components, developers need to realise component function encapsulation, dynamic configuration support and de-coupled component interaction mechanisms. At run-time, components are composed and compositions validated.

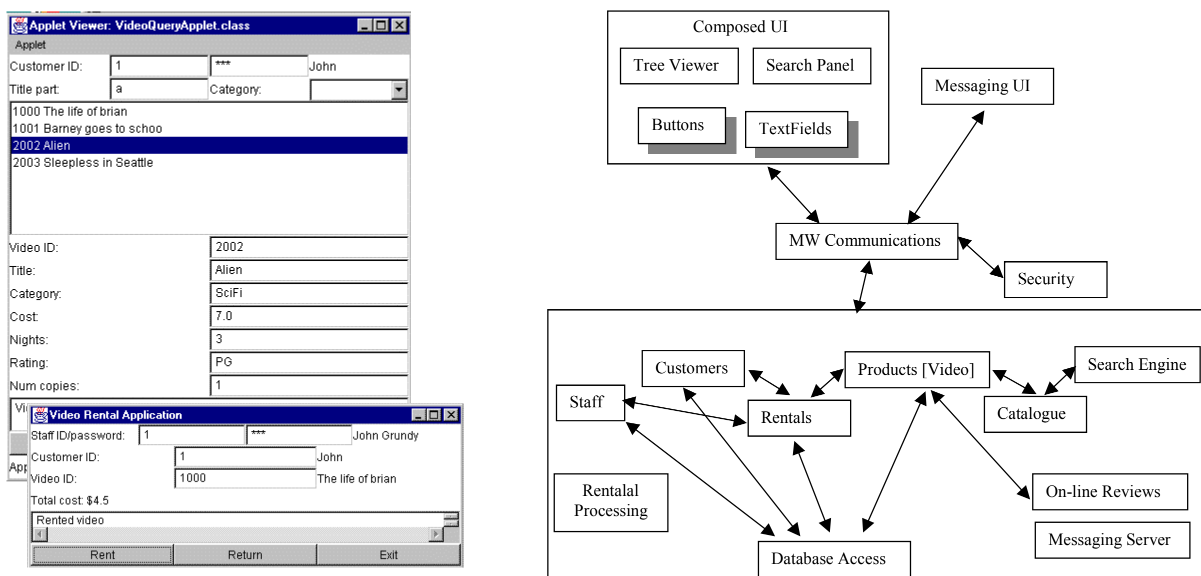


Figure 1. Example E-commerce system and a possible component-based architecture.

3. Aspect-Oriented Component Engineering

We developed Aspect-Oriented Component Engineering (AOCE) to help developers engineer better software components [9]. *Component aspects* are broad categories of annotations we use to describe systemic system properties that components provide functions for or require functions from other components. Examples of component aspects (which we refer to just as "aspects" from here) include user interface, distribution, transaction processing, security, persistency, configuration and collaborative work support facilities.

Aspect details describe various systemic properties under each aspect category that some components *provide* and that others *require*. For example, one component may provide a button panel (user interface aspect detail) another component may require to extend (e.g. to add its own buttons). One component may provide event broadcasting support, which another requires to do distributed communications. Each aspect detail has one or more *aspect detail properties* which further characterise it e.g. event transfer rate, memory usage size, kind of user interface affordance, synchronous vs. asynchronous group editing, and so on. Aspect detail properties may be single-valued or specify an acceptable value-range constraint. Component aspect details may overlap e.g. marshalling for persistency and distribution, feedback for user interface and collaborative work. Several component functions may be impacted by the same aspect detail and a single function may be impacted by multiple details.

Figure 2 shows some example aspects for video system components. Each component provides some services to the overall component-based application e.g. the Tree Viewer and Reviews provide UI parts of its user interface; the middleware component provides distribution services; the database connectivity component provides persistency management and distributed access; the on-line review components provide collaborative work support; and the customers and products components provide server-side data management and processing. Most components also, in a typical system, require various services from other components to operate. For example, the Tree Viewer requires distribution support and local persistency support (for caching); the Reviews UI a collaboration server; the middleware communications component (optionally) requires security services; and the Products component requires persistency and distribution services (to allow client access). Many other aspects could be identified in this system: threading is required by server-side search engine and rental processing components to support multiple, simultaneous client search requests and rentals; in many systems memory management services are required; caching in both client and server components could be provided or required; communications may require security support (encryption, access rights management); and so on.

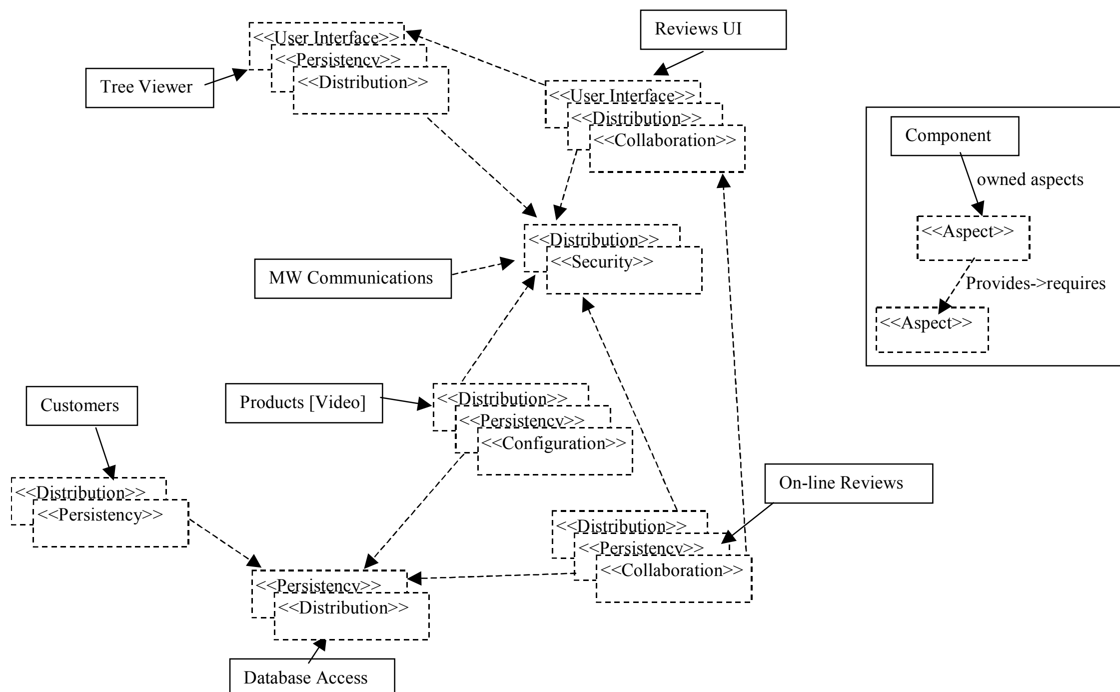


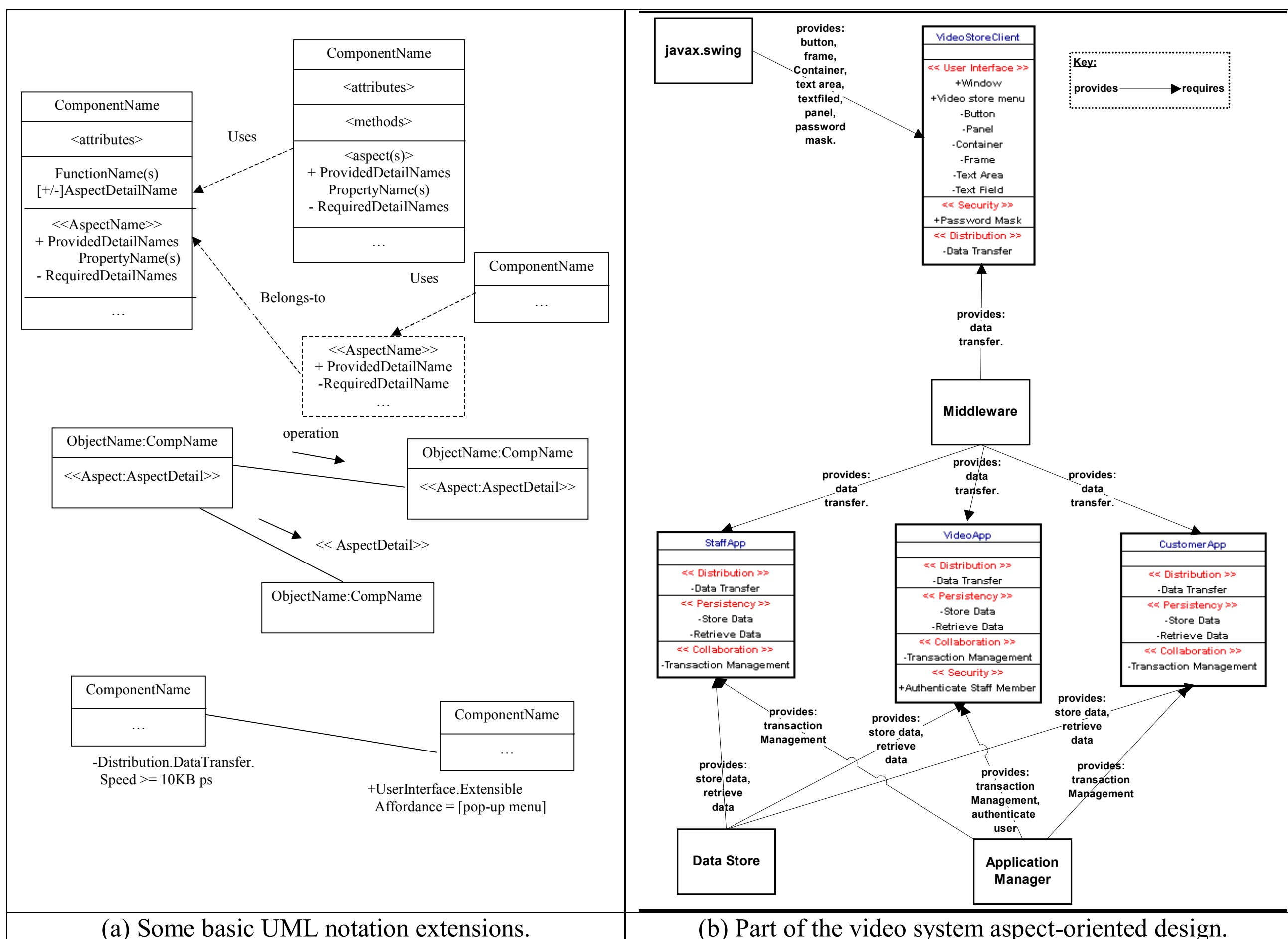
Figure 2. The video system components and some of their aspects.

4. Aspect-Oriented Design with the UML

We originally extended a custom component specification and design notation to incorporate simple representations of aspects and developed a simple textual aspect specification language formalism [9]. While these proved to be effective for use in developing aspect-oriented component designs, they are non-standard and hence problematic for a wider audience to use. Standard design notations like the Unified Modelling Language (UML) are widely used, but lack both component and aspect characterisations. Component development approaches using the UML [1, 5] generally lack adequate component functional and non-functional characterisation along cross-cutting lines [3, 14, 9]. We wanted to see how the notion of component aspects would integrate with the UML and to explore ways of implementing aspect-oriented UML designs using various technologies.

Unlike some approaches to extending the UML for component development and aspect-based design [5, 14], we extend both the UML meta-model and its visual notation. We also introduced additional steps into the Unified Process to include the identification and use of component aspects. We focused our extensions on providing concrete representations of aspect "cross-cuts" on component functional/non-functional characteristics. Our extensions make explicit provided and required component aspect details, in class (software component), sequence and collaboration diagrams.

Figure 3 (a) shows some of the notational extensions of our extended UML. Class diagram icons representing components are extended by adding additional compartments underneath the standard attribute and method ones, one for each kind of aspect. Each compartment contains a label identifying the aspect and aspect details, indicating provided (+) and required (-) functional and non-functional characteristics. Each aspect detail has one or more properties, which can be shown or hidden. Properties have value constraints, typically viewed in dialogues in a CASE tool.



(a) Some basic UML notation extensions. (b) Part of the video system aspect-oriented design.

Figure 3. Part of an aspect-oriented component design.

Component functions may be annotated with the aspect details they provide and/or require. Collaboration and sequence diagram objects may be annotated to indicate event flows relate to particular aspects details provided and required by the objects. Operations may be annotated to indicate the provided aspect detail(s) being used. Designers can specify various properties and property constraints of provided and required details, using UML's Object Constraint Language (OCL) annotations (though these are usually specified in dialogues in our CASE tool). These provide a facility to reason about AOCE designs. For example, if the VideoStoreClient requires data transfer facilities that can handle > 10,000 bytes per second data exchange, then connecting a middleware component providing a modem connection can be invalidated as not sufficient.

The example extended class diagram in Figure 3 (b) shows the VideoStoreClient component, which provides the basic customer user interface (composed from several other components, not shown here). This provides several aspect-encoded facilities (a window and password mask) and requires others (user interface components and data transfer support). Three application server components (CustomerApp, StaffApp and VideoApp) require various middleware facilities, encapsulated in Data Store, Middleware and Application Manager components.

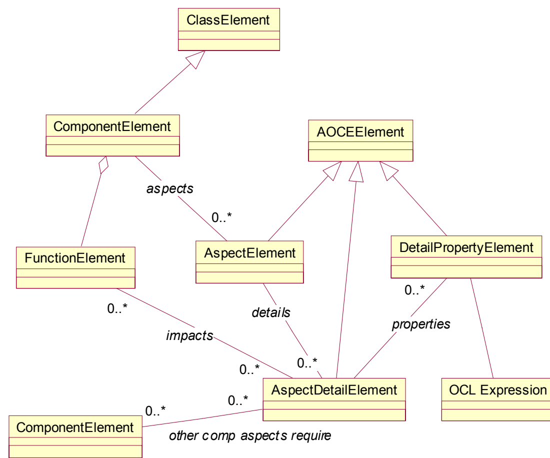


Figure 4. Some of our UML meta-model extensions.

Rather than use ad-hoc UML stereotypes on classes and functions or use unstructured note annotations, as some researchers have done [3], we introduce a concrete set of UML meta-model extensions, outlined in Figure 4 These include Components, Aspects, AspectDetails and DetailProperties, plus some additional inter-element links. Aspect details may be shown as either function annotations or in aspect compartments within class icons, or within their own aspect icons. Aspect detail properties include a type and optional OCL expression, used to specify valid values and valid inter-component aspect detail property values. Aspect extensions can be usefully applied to other UML diagrams and meta-model entities, such as collaboration and sequence diagrams, state diagrams, and object and function call elements.

We have added stages to the Unified Process to incorporate aspect-oriented component design usage. Components are identified and then their aspects characterised (including, where possible aspect details and properties). Linked and composed components have their provides/required relationships analysed to ensure all components with required aspect details are associated with components with one (or more) matching provided details. Aspect detail property constraints are checked for each provided/required detail match to ensure compatibility. Component groupings can also have "aggregated" aspects. These are useful for enforcing group-wide aspect detail properties and for reasoning about group-to-group aggregate aspect relationships.

5. Implementing Aspect-Oriented Designs

We have investigated three mechanisms for realising aspect-oriented component designs: extensions to JViews, a Java Beans-based component model; using Enterprise Java Beans; and using the Perceval intermediate aspect-oriented language.

5.1. JViews Components

We extended a component-based framework we developed, called JViews, to incorporate a set of classes which act as codifications of component aspects. JViews components are built from component designs and sets of aspect-implementing classes are reused or specialised to provide a run-time description of component aspects and various run-time aspect-based facilities. For example, a JViews-implemented tree viewer component advertises that it provides an extensible menu bar (that other components can add items to) and it requires a remote, data-providing component to source tree information from. A messaging server-side component requires database connectivity (to store messages in), a communications component (to send/receive message events to/from), optionally requires a security encoding aspect detail (if a component providing this is linked to it, messages are encrypted, if not they aren't), and provides an implementation of a user registration collaborative work aspect detail.

Figure 5 shows some of the JViews components we built to realise a prototype of the video library aspect-oriented design, along with an indication of some of their aspect details. Aspect detail objects are advertised by each

owning component, with other components accessing these to discover and make use of other component services. For example, the tree model component discovers the search component has a menu bar that can be extended and it uses functions associated with this provided aspect detail to extend the menu in a very de-coupled way [8]. The server-side TCP/IP communications component discovers the security component (if present) has encoding/decoding functions which it envokes to encrypt and decrypt transferred data. Each aspect detail class has several aspect detail properties and constraints that are checked at run-time to validate a component configuration e.g. check all required aspects details for components are met and are consistent with matching provided aspect details.

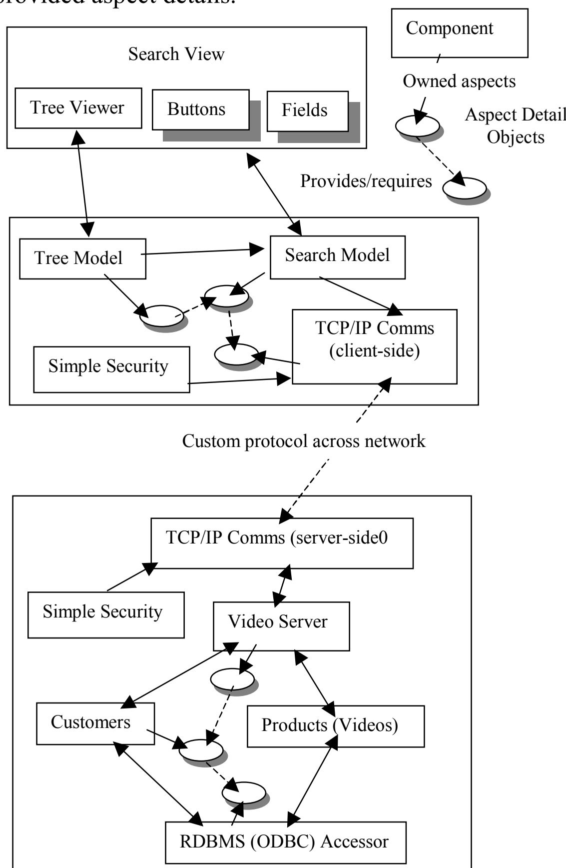


Figure 5. Some of the JViews components implementing the video system.

5.2. Enterprise Java Beans

While our JViews framework extended with aspects is a powerful implementation mechanism for AOCE it suffers from being non-standard and only JViews components can (easily) be used within the framework. We have recently been investigating the use of Enterprise Java Beans (EJBs) to implement aspect-oriented component designs. The basic EJB architecture is outlined in Figure 6 [21]. Server-side *components* ("Beans") provide a "home" interface through which they are located and accessed. Beans are managed by Bean *containers*, which are in turn managed by Enterprise Java Beans *servers*.

Beans typically do not manage their own distributed communications, transaction processing, persistency, security or threading, but these systemic services are provided by containers. Some limited load-balancing and distributed Bean management is provided by EJB server implementations.

EJBs would seem a reasonably amenable component model for AOCE: container services provide many systemic, aspect-related services to components, providing an architectural isolation mechanism for these aspects in component implementations. Figure 6 shows some of the EJB components we used to realise a prototype of the video system aspect-oriented design. Entity Beans (EB) manage data while Session Beans (SB) manage business processing logic. The container manages Entity bean persistency, transactions, threads and client-server communications. OCE-designed components translate reasonably well into EJB-realised component implementations, particularly if the basic EJB architecture is sufficient for the application being developed. Often our AOCE designs include "middleware" components providing facilities that EJB containers provide and thus these components "disappeared" in our EJB implementations (whereas they were mostly realised by multiple JViews reusable components in our previous example). Aspects aid in dividing responsibility between EJB components and we used them to help identify functions that need to be expressed in an EJB's interface to support component configuration and de-coupled interaction (i.e. not tying EJBs to other specific EJB types). This enhances reusability and flexibility of the EJB components.

Unfortunately the EJB model, while suitable for some applications and reasonably flexible, makes some architectural decisions difficult to achieve e.g. decentralised processing, object caching and client-side distribution, caching and persistency functionality. The EJB model is also solely a server-side component model and thus component-based clients must use an alternative implementation model, resulting in a heterogeneous implementation. EJBs provide quite limited run-time introspection and dynamic discovery mechanisms, compared to aspect codifications in JViews. Consequently we have found it much harder to develop truely dynamically deployable, reusable and de-coupled components in EJBs than in JViews.

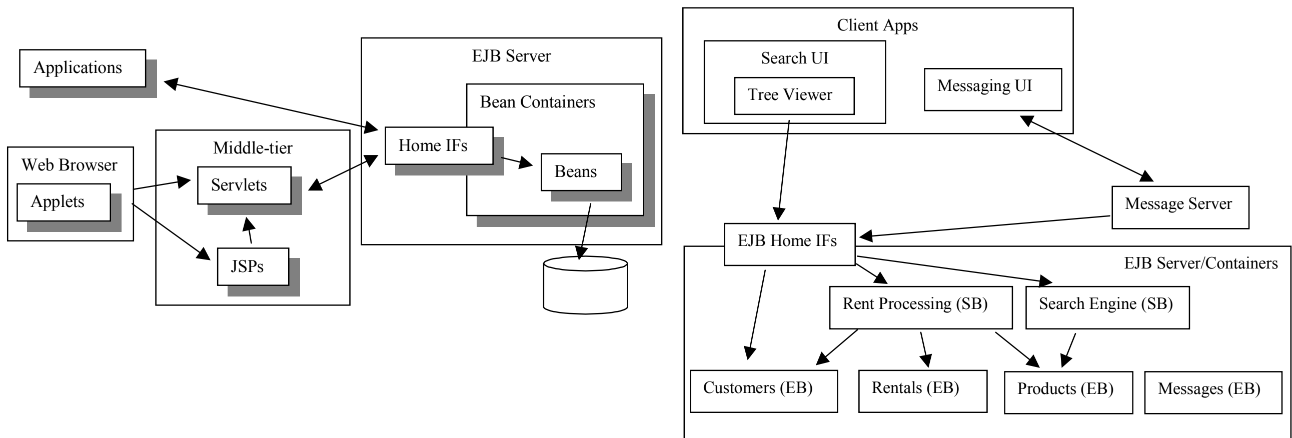


Figure 6. EJB model architecture and some of the EJBs implementing the video system.

The EJB model doesn't defer all aspect-related services to containers, resulting in some components or being developed using custom models e.g. for collaborative work, component configuration and synchronised processing, meaning 3rd party EJBs using different solutions may not be readily combined.

5.3. Perceval

We have briefly investigated using and extending Perceval to encode aspect-oriented component designs. Perceval is an XML-based encoding of aspect-oriented designs which can be generated by CASE tools and which can be translated by other tools to generate various aspect-oriented implementations [3]. Figure 7 shows parts of our extended Perceval specifications of two video system components and three aspect details. These can be translated into various (partial) implementations.

```
<perceval2:component id="VideoStoreClient">
  <perceval2:functionDefn id="constructUI">
  …
  <perceval2:functionDefn id="addMenuItem">
  …
  <perceval2:functionDefn id="rentVideo">
  …
</perceval2:component>

<perceval2:component id="TCPSocketClient">
  <perceval2:functionDefn id="write">
  …
  <perceval2:functionDefn id="read">
  …
</perceval2:component>

<perceval2:aspect id="extensible video menu"
  kind="user interface"
  detail="extensible affordance"
  provided="true">
  <perceval2:class>
    <perceval2:classref ulink="VideoStoreClient">
  </perceval2:class>
  <perceval2:method>
    <perceval2:and>
      <perceval2:methodref ulink="constructUI">
      <perceval2:methodref ulink="addMenuItem">
    </perceval2:and>
  </perceval2:method>
  <perceval2:property name="kind" value="pulldown menu">
  </perceval2:property>
  <perceval2:property name="allowable">
    <perceval2:or>
      <perceval2:value>insert item</perceval2:value>
      <perceval2:value>append item</perceval2:value>
    </pereval2:or>
  </perceval2:property>
  <perceval2:action>
  …
</perceval2:aspect>
```

```
<perceval2:aspect id="video data transfer"
  kind="distribution"
  detail="data transfer"
  required="true">
  <perceval2:class>
    <perceval2:classref ulink="VideoStoreClient">
  </perceval2:class>
  <perceval2:method>
      <perceval2:methodref ulink="rentVideo">
  </perceval2:method>
  <perceval2:property name="kind" value="synchronous">
  </perceval2:property>
  <perceval2:property name="protocol" value="custom">
  </perceval2:property>
  <perceval2:action>
  …
</perceval2:aspect>

<perceval2:aspect id="socket transfer"
  kind="distribution"
  detail="data transfer"
  provided="true">
  <perceval2:class>
    <perceval2:classref ulink="TCPSocketClient">
  </perceval2:class>
  <perceval2:method>
    <perceval2:and>
      <perceval2:methodref ulink="write">
      <perceval2:methodref ulink="read">
    </perceval2:and>
  </perceval2:method>
  <perceval2:property name="kind">
    <perceval2:or>
      <perceval2:value>synchronous</perceval2:value>
      <perceval2:value>asynchronous </perceval2:value>
    </perceval2:or>
  </perceval2:property>
  <perceval2:action>
  …
</perceval2:aspect>
```

Figure 7. Extended Perceval descriptions of some video system components and aspects.

Perceval offers the potential advantages of describing aspect-oriented component designs in a "common exchange format", allowing developers to generate this format from a wide variety of tools (or simply by hand), and generating a variety of component implementations direct from the specification. Perceval XML encodings could form the basis of run-time component introspection and interaction mechanisms, a little like the JViews ones. However, a set of run-time functions in the target Perceval implementation language(s) would be required to use these encodings. Unfortunately Perceval doesn't directly encode the concept of a component, nor provide any explicit UML aspect-based extensions.

6. Tool Support

We originally extended a CASE tool, JComposer, designed to generate JViews component implementations with aspect description facilities [8], adding code generation support to generate aspect codifications specifically for JViews components. More recently we have extended JComposer to provide UML-compliant representations of component-based designs, including adding UML meta-model extensions to represent aspects. OCL expressions are used to specify aspect detail properties and basic inter-aspect detail constraints. We modified JComposer to generate XML-encoded extended perceval documents, describing aspect-oriented component designs. Code generators take the Perceval encodings and respectively generate parts of JViews or EJB component implementations. Developers use programming environments like Jbuilder or Visual Age to complete these implementations and deploy components. Figure 8 shows the basic support for aspect-oriented component design and implementation provided by our toolset.

We have developed a component repository for JViews components, which queries them for their aspect encodings for indexing. We are extending this to provide support for storing and retrieving EJB components, encoding aspects in EJB deployment descriptors for run-time access. Some run-time support for deploying components includes user-browsable aspect descriptions and simple validation functions. These dynamically check all required component aspect details have matching provided details in related components. Some basic aspect detail property constraint checking is also supported.

We implemented our extended JComposer using JViews itself. JComposer component repository data is extracted to generate the Perceval XML encoding. XSLT (XML Stylesheet Language Transformations) scripts are used to implement extensible code generation facilities.

7. Discussion

A great deal of work has been done in recent times to address the problem of separation of concerns in software development [12]. Examples of such work include viewpoint-based requirements, designs and tools [7, 8], subject-oriented programming [11], hyper-slices [24] and aspect-oriented programming systems [15, 16, 14]

Viewpoints have been used for various purposes, including requirements engineering, specification and design, user interface construction and in various software tools [7, 12]. Aspects are a specialisation of the general notion of a viewpoint i.e. a certain perspective on a software system.

Viewpoints of one form or another are used in all development methods, including component development methods like Catalysis™ and SelectPerspective™ [1, 5]. Unfortunately the viewpoints used by almost all current development methods are oriented towards functional decomposition, not addressing the cross-cutting concerns inherent in most system designs and implementations [24]. Current component design methods and implementation technologies adopt this function decomposition-centric approach, resulting in tangling of systemic, cross-cutting concerns in both component designs and implementations [9].

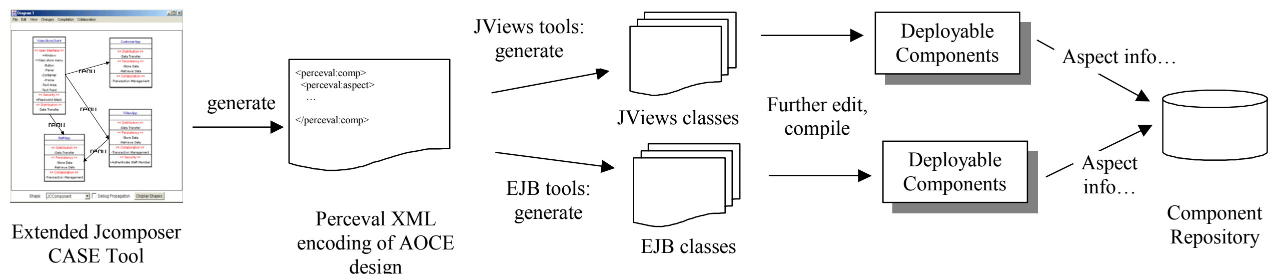


Figure 8. An outline of our prototype tool support for AOCE, UML and EJB implementation.

Hyper-slices and subject-oriented programming are similar to aspect-oriented design and programming in that they attempt to provide developers with alternative views of cross-cutting concerns [11, 24]. In fact, our aspect-oriented component engineering views are specialised kinds of hyper-slices deployed to assist component development. Some component development methods have introduced specialised views of component characteristics, notably security and distribution issues [10, 1].

Little attention has so far been paid to applying aspects to component-based systems development. Adaptive plug-and-play components utilise components that implement something similar to the concept of an aspect, being mixed to realise the separation of various concerns from component implementations [18]. While component design methods provide very limited ability to identify overlapping concerns between components the isolation of systemic functions e.g. communications, database access and security, into reusable components (or component containers, as in EJBs) is common in component technologies [2, 21, 25]. This partially addresses the problems of components encapsulating these systemic services, and enables isolation of these services and access via well-defined, component-based interfaces. However, not all component aspects can be suitably abstracted into individual components, due to overlaps and the eventual over-decomposition of systems (every component function being very small and every function having its own component, producing massive numbers of tiny components).

We have found aspect-oriented component engineering offers the ability to identify, codify and reason about aspects during specification and design of software components. This greatly increases the developer's ability to document and reason about the cross-cutting concerns impacting on components, but also allows reused component's overlapping concerns to be quickly identified and analysed. AOCE designs can be realised in various ways: using dedicated aspect-extended component framework (e.g. our JViews), using a standard component technology (e.g. EJBs or COM+), using an aspect-oriented programming language like Perceval or AspectJ, or even without using component technologies at all. The advantage of using JViews is its codification of aspects in component implementations and the ability of components to discover these at run-time and interact with other components via standardised, highly de-coupled aspect object functions, similar to the composite adaptors concept [19]. The disadvantage is the need to use a non-standard component architecture that is difficult to combine with 3rd party components. The advantage of implementation with EJBs is the production of more generally sharable components, at a cost of losing out on JViews-style run-time access to component aspects and highly de-coupled component interaction support. Using languages like AspectJ, Perceval (to generate AspectJ or other component implementations) or a standard programming language, means an aspect-oriented design is not actually realised by software component technology. In some situations this is acceptable, but dynamic system configuration and end user enhancement are usually not possible. Aspect-oriented designs cost developers more effort in terms of identifying, documenting and using component aspects. However, we have found this effort is well justified on the component development projects we have undertaken as the extra richness in the designs and ability to use aspects to guide more effective component implementation results in much better component implementations.

Various future research directions exist in extending our AOCE-based extensions to the UML. This includes further aspect-based constraint representation, other notational models for aspects, adding aspects to dynamic design diagrams, and further process enhancement. We are interested in incorporating aspect information in EJB and COM+-based component implementations, in ways that integrate seamlessly with their standard introspection and interface management mechanisms, to provide JViews-style run-time configuration and interaction mechanisms using aspect ojects. Many possibilities exist for improved tool support. These include further enhancements to our JComposer UML diagramming, improved constraint checking using aspect property values, the extension of our use of aspect-enhanced data exchange formats between tools, improved code generation from aspect-based designs, and run-time usage of aspects in deployment and component configuration tools.

8. Summary

We have added some basic extensions to the Unified Modelling Language to incorporate aspects, aspect details and aspect detail properties for software components. These allow designers to codify systemic, cross-cutting concerns between components using these additional design viewpoints. Inter-component provision and requiring of services and service non-functional compatibility can be specified and checked using these aspect encodings. We have explored implementing aspect-oriented UML component designs using our JViews component framework, Enterprise Java Beans and the Perceval XML aspect design encoding. JViews provides framework support for component aspects, including run-time support for accessing and using aspects to support component configuration and interaction. EJBs provide a more standard implementation for aspect-oriented component designs. Aspects assist in

identifying EJB component divisions of responsibility and interface definition which help improve component reuse and run-time configuration support. Perceval provides an intermediate, tool data exchange format for aspect-oriented designs. We have built prototype code generation tools using Perceval component design encodings that produce JViews or EJB component interface and class skeletons. These can be further extended using 3rd party component implementation tools. We are investigating further aspect-based extensions to the UML and encoding mechanisms for aspects in EJB implementations.

Acknowledgements

Support for this research from the University of Auckland Research Committee and the Public Good Science fund are gratefully acknowledged.

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. 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..
9. 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.
10. 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, pp.61-6. Los Alamitos, CA, USA.
11. 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 (OOPSLA),* September 1993.
12. 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.
13. Herzum, P. and Sims, O. *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise*, Wiley, December 1999.
14. 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.
15. 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.
16. Kiczales, G. and Lopes, C. Recent developments in AspectJ, In *Proceedings of the ECOOP'98 Workshop on Aspect-oriented Programming*, Brussels, Belgium, July 1998.
17. 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.
18. 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.
19. 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*, Mehmet Aksit, editor, Kluwer Academic Publishers, 2000.
20. Mahmoud, Q. Securing Component-Based E-Commerce Applications, *Component Advisor Magazine*, March 2000.
21. Monson-Haefel, R., Enterprise JavaBeans, O'Reilly, 1999.
22. Sessions, R., *COM and DCOM: Microsoft's vision for distributed objects*, Wiley, 1998.
23. Szyperski, C.A., *Component Software: Beyond OO Programming*, Addison-Wesley, 1997.
24. 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.
25. Vogal, A. CORBA and Enterprise Java Beans-based Electronic Commerce, In *International Workshop on Component-based Electronic Commerce*, Fisher Center for Management & Information Technology, UC Berkeley, 25th July, 1998.
26. Zhao, J.L. and Misic, V.B. Toward Component-Based Electronic Commerce: A Workflow Perspective, In *International Workshop on Component-based Electronic Commerce*, Fisher Center for Management & Information Technology, UC Berkeley, 25th July, 1998.