# Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming♣

Mik A. Kersten and Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver BC Canada V6T 1Z4
{mkersten, murphy}@cs.ubc.ca

Technical Report Number TR-99-04

April 7, 1999

**Abstract**. The Advanced Teaching and Learning Academic Server (Atlas) is a software system that supports web-based learning. Students can register for courses, and can navigate through personalized views of course material. Atlas has been built according to Sun Microsystem's Java™ Servlet specification using Xerox PARC's aspect-oriented programming support called AspectJ™. Since aspect-oriented programming is still in its infancy, little experience with employing this paradigm is currently available. In this paper, we start filling this gap by describing the aspects we used in Atlas and by discussing the effect of aspects on our object-oriented development practices. We describe some rules and policies that we employed to achieve our goals of maintainability and modifiability, and introduce a straightforward notation to express the design of aspects. Although we faced some small hurdles along the way, this combination of technology helped us build a fast, well-structured system in a reasonable amount of time.

♣ This work is funded in part by a Natural Sciences and Engineering Research Council of Canada (NSERC) research grant.

# 1 Introduction

The Advanced Teaching and Learning Academic Server (Atlas) supports web-based learning: students can register for courses, and can navigate through personalized views of course material.

Atlas has been built using an aggressive combination of new technologies: Sun Microsystem's Servlet technology[1] and aspect-oriented programming [KLM+97]. Building the system to the Servlet specification supported development based in Java™ [GJS96], which in turn permitted the use of Xerox PARC's AspectJ™ [LK98] aspect-oriented extension to Java.

In this paper, we focus on what it was like to build a moderate-sized (180 class) system using the new aspect-oriented programming approach, describing:

- how we used aspects for both the system and its development, and
- how the use of aspects affected our object-oriented development practices.

Synthesizing from our experiences, we describe basic categories of aspects that arose, trade-offs we have found in using aspects from these different categories, and policies we employed to achieve our goals. We also introduce a straightforward notation to express the design of aspects in our system. For Atlas, the use of aspect-oriented programming paid off: we have been able to develop a system that meets its functional requirements and that shows promise for meeting the non-functional requirements of maintainability and modifiability.

# 2 Background

## 2.1 Atlas Base Requirements

Students logging into Atlas are presented with a customized desktop-like interface (Figure 1). The desktop offers Atlas course tools (interactive user-tailored courses, quick references, and a manual), support tools (such as a calculator, HTML editor, and color picker), and interaction tools (bulletin board and chat). The content and style of the desktop is generated based on user settings, the characteristics of the browser, and the connection bandwidth.

Atlas needs to handle concurrent access by thousands of students, and to scale gracefully to anticipated increases in demand.

## 2.2 Extended Atlas Requirements

Educational computing environments vary greatly. Some educational institutions may serve courses to thousands of students with all of the students accessing the courses through a high-speed intranet. Other institutions may serve large numbers of students some of whom are located on a high-speed link while others are located on lower-speed links. Still others may desire to serve just a small number of students.

Most of today's web-based course servers meet this need using a hardware-based approach**.** Supporting more students means an investment in newer, higher-performance hardware. This approach is an expensive alternative to software-based scaling which can make effective use of previous generation commodity machines [Bre98].

[1] See http://java.sun.com/products/servlet/index.html. We used Version 1.2 of the Servlet API.


**Figure 1 The User Interface to Atlas**

We wanted to build Atlas to be economically configurable for the different kinds of educational computing environments in which it might run. As a result, we set the following two requirements. First, we required Atlas to run in different configurations based on performance requirements, hardware availability, and network conditions. Second, we required the system to be scalable at runtime in order to minimize downtime.

Four run-time configurations, *network contexts*, are of interest (Figure 2):

1. *Single server* context (Figure 2a). In this context, there is one server. As user requests come in, the web server spawns threads to run the application.
2. *Web server + application* context (Figure 2b). In this context, the execution of the application functionality can be moved to a separate node in order to reduce the load on the web server node.
3. *Parallel application server* context (Figure 2c). The web application is distributed over a number of nodes to exploit the inherent parallelism associated with HTTP requests. Requests are load balanced across the application nodes.
4. *Applet* context (Figure 2d). To achieve the performance and responsiveness associated with desktop applications, the application functionality in this context is moved into the web browser. This context requires a centralized database as their may be an arbitrary number of browser clients running. We place this database on the same node as the web server.

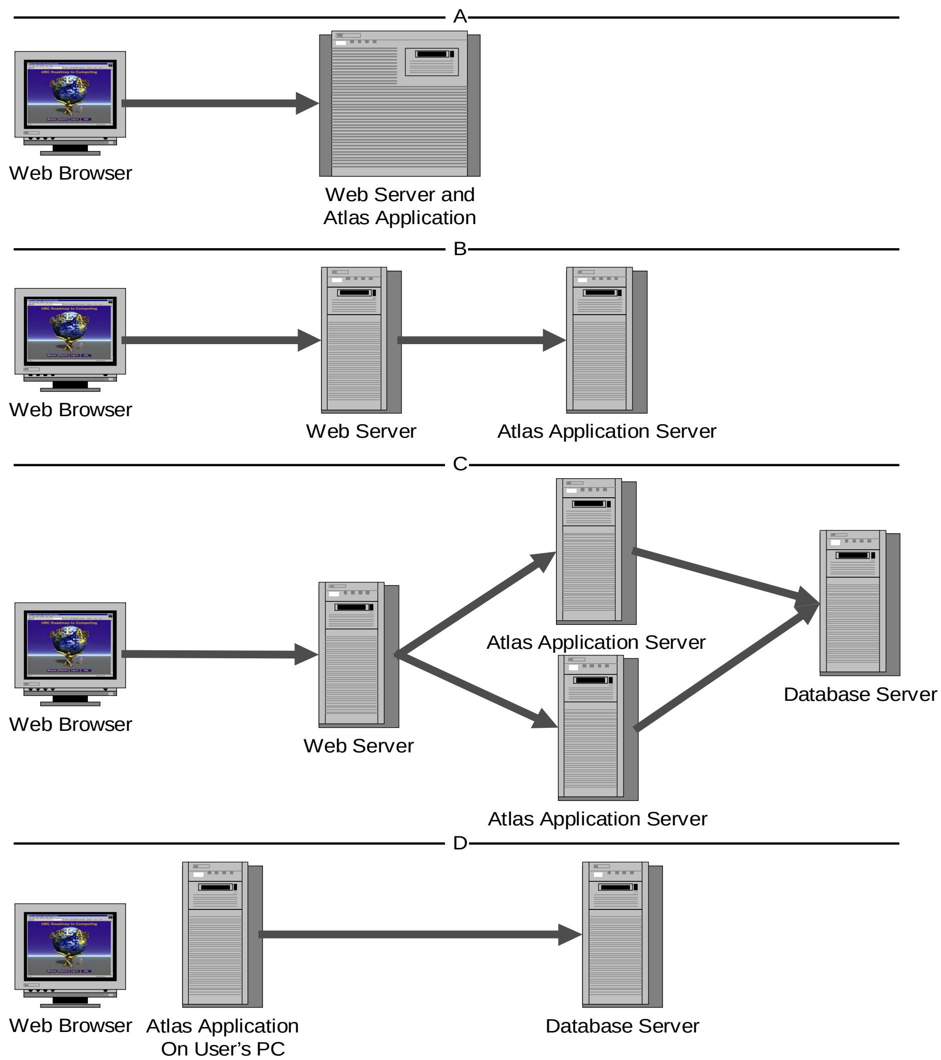


**Figure 2 Network Contexts**

## 2.3 Aspect-Oriented Programming with AspectJ

Aspect-oriented programming (AOP) is intended to help software developers more cleanly separate concerns in their source code. An *aspect* describes how code for a concern should be integrated, or *woven*, into code for the system. AspectJ provides aspect-oriented programming support for Java.

Using AspectJ, an aspect is defined in a similar manner to a Java class: an aspect has a name and may have its own data members and methods. Two other constructs are available. The `advise` construct permits a software developer to add code `before` or `after` an existing method or methods in a system. The `introduce` construct permits a software developer to introduce new state or functionality into an existing class or classes.

Figure 3a shows an aspect, `RemoteCallTracer`, that uses both the introduce and the advise constructs. The introduce construct is used to add a socket into the `PageBuilder` class. The advise construct is used to add code before all methods on the `PageBuilder` class that will write strings representing the entry of a method to a socket that is introduced into each class.

In the `RemoteCallTracer` aspect, the `before` keyword is preceded by the `static` modifier. This modifier means that the advise acts on every instance of the specified class or classes. In this paper, we refer to aspects with these kinds of weaves as *static* aspects.

AspectJ also supports *dynamic* aspects, which allow a developer to advise specific instances of a class. The first part of the code in Figure 3b shows code for a dynamic aspect, `CallTraceMonitor`. In comparison to the static aspect, this aspect has a constructor that creates the socket to be used for tracing and uses non-static weaves. For the dynamic aspect to have an affect at run-time, it must be created (similar to an object). Specific objects to be traced can then be registered with (added to) the aspect. The main code shown at the bottom of Figure 3b performs the creation and registration operations.

To create a system using AspectJ, a developer uses a weaver tool to integrate the code in aspect files into the classes of the system. Since AspectJ works as a pre-processor, the output of the weaver is a set of Java class files that can then be compiled with a standard Java compiler.

We used several versions of the AspectJ pre-processor during the development of Atlas; specifically, AspectJ 0.2.0beta4 through beta10.

A - advise and introduce

```
import java.io.*; import java.net.*;
aspect RemoteCallTracer {
    introduce PrintWriter PageBuilder.socketWriter
       = new PrintWriter( new Socket( "hostname", 4444 ).getOutputStream() );

    advise * PageBuilder.*( * ) {
        static before {
            socketWriter.println( "ENTERING " + thisMethodName );
        }
    }
}
```

B - dynamic aspec and driver class code

```
aspect CallTraceMonitor {
    PrintWriter socketWriter = null;
    introduce PrintWriter PageBuilder.socketWriter;

    public CallTraceMonitor( String hostname, int socket ) {
        socketWriter = new PrintWriter(
            new Socket( "hostname", 4444 ).getOutputStream() );
    }

    advise * PageBuilder.*( * ) {
        before {
            socketWriter.println( "ENTERING " + thisMethodName );
        }
    }
}
```

```
public static void main( String[] args ) {
    PageBuilder pageBuilder = new PageBuilder();        // create the object
    CallTraceMonitor builderMonitor                     // create the aspect
        = new RemoteCallTraceMonitor( "localhost", 4444 );
    builderMonitor.addObject( pageBuilder );            // add the object
    // . . .
}
```

**Figure 3   Sample AspectJ Code**

# 3 Atlas and Aspects

Atlas uses aspects in several different ways: to support different architectural configurations; to extend the implementation of design patterns; and to support the development of the system. Atlas comprises 48 packages, 180 classes, 17 aspects (including sub-aspects), and approximately 11000 lines of commented source code.

## 3.1 Aspects in the Atlas Architecture

The main architectural challenge in Atlas was determining how to support the four different network contexts without blowing up the complexity of the system structure. Since Atlas is built to the Servlet specification, one constraint was clear: requests from a student's browser would arrive at a Java web server and would then be delegated by the server to Atlas. The Atlas code would then be responsible for providing application functionality back to the student.

To facilitate support for the network contexts given this basic constraint, we separated out the basic infrastructure components for the contexts as an object-oriented framework: the Distributed Servlet Broker (DSB). Aspects are then used to support run-time configuration of contexts and to tailor the behaviour of the Atlas functionality, which is instantiated into the DSB framework. First, we describe the DSB; then we describe how aspects are used to support the various configurations.

### 3.1.1 The Distributed Server Broker (DSB) Framework

The DSB framework consists of three main components. The `DSBClient` handles the HTTP requests forwarded from the Java web server and determines to which application the requests should be forwarded. The `DSBServer` provides an interface to the web application or applications being served, such as Atlas. The `DSBNexus` provides an interface to the databases and registries used by an application and offers common functionality, such as a dynamic HTML generating library.

Instantiating Atlas within this framework first involves encapsulating the functionality of Atlas into a component called `AtlasService`. This component must meet an interface set by the `DSBServer` component; this interface is identical to the Servlet interface. The `DSBServer` is then configured to be able to instantiate an `AtlasService`. Finally, the `DSBNexus` component must be configured to hook to the appropriate databases.

With the DSB, running Atlas in different network contexts amounts to instantiating and configuring DSB components and the `AtlasService` in the appropriate numbers on various nodes.

Configuring two of the contexts is straightforward. In the *single-server* context, all components of the DSB framework and the `AtlasService` run on a server node with connections made from client machines. In the *applet* context, all components run on the client machine, except the `DSBNexus` component that must be centralized to allow for multiple browser access.

In the *web server + application* context (shown using UML [BJR98] notation in Figure 4a), the Java web server runs on one node with a `DSBClient`; the `DSBServer`, `DSBNexus`, and `AtlasService` components run on a second node. This context requires communication between the `DSBClient` and `DSBServer` to cross a node boundary: we use ObjectSpace Voyager [Obj] to provide transparent access to the distributed Java objects running on different nodes.

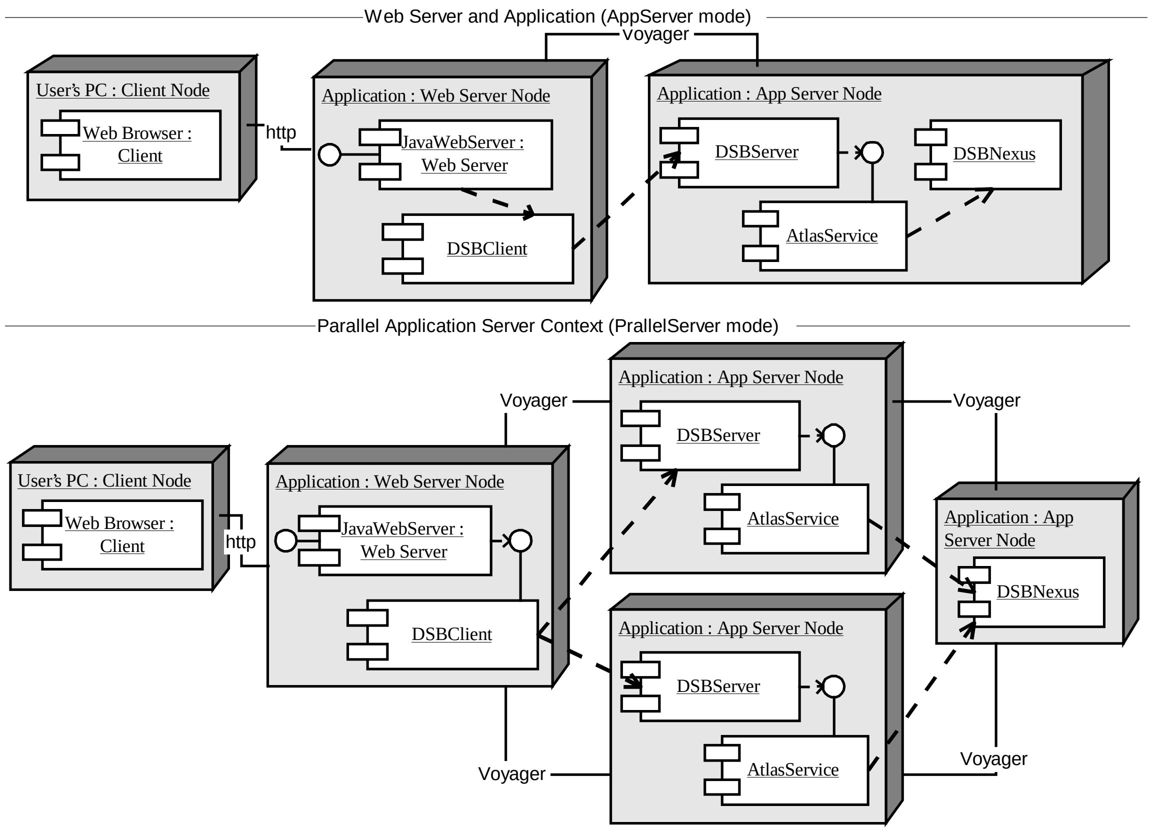


**Figure 4 Two Network Contexts**

The *parallel application server context*, which exploits parallelisms of HTTP requests, is somewhat more complex (Figure 4b). Any node running the `AtlasService` must also be running a `DSBServer` component. The `DSBNexus` component, which is centralized to avoid concurrency problems, is run on a separate node. In this context, new instantiations of the `AtlasService` running on a new node may be added at runtime; the `DSBClient` is responsible for load balancing between the available application nodes. ObjectSpace Voyager is again used to provide transparent access to distributed Java objects.

### 3.1.2 Aspects for Network Contexts

Supporting the different network contexts requires DSB components to interact in a number of different ways. In the *parallel application server* context, for instance, the `DSBClient` must be capable of interacting with more than one distributed `DSBServer`. Implementing any particular context requires changes to be made across the DSB components. Given the cross-cutting nature of these contexts, we chose to modularize the network contexts as aspects.

Figure 5 depicts the aspects (shown as diamonds) involved in providing the network contexts, and shows how these aspects interact with the source code for the system.[2] The `NetworkContext`

[2] An arrow from an aspect to a package indicates that the aspect acts on the source code comprising the package. More detail about the graphical notation used is provided in Section 4.3.

aspect generalizes four sub-aspects, each of which is responsible for providing the appropriate behaviour for a context.

Each of the specific network context sub-aspects contains the code particular to the given context. The `DefaultContext` supports the *single-server* context. This context is the default mode for the application. As a result, this aspect alters no behaviour in the application. The `AppServer` aspect supports the *web server + application* context, extending the system to handle remote requests between the `DSBClient` and the `DSBServer`. The `ParallelServer` aspect does the same, and also extends the system to communicate with a remote, centralized database. The `AppServer` and `ParallelServer` aspect use the `ServerRegistry` class to keep track of servers being used; for the latter, the `ServerRegistry` also performs load-balancing. The `Applet` aspect generates a page that contains an applet that acts as a wrapper for a `DSBClient`. This aspect is also responsible for the additional protocol issues of communicating from an applet rather than a regular node. (The `Applet` aspect has been designed and is currently being implemented.)

As Figure 5 shows, the aspects interact with both DSB components and the `AtlasService`. The `AtlasService` was not written with distribution in mind. As a result, a large amount of the code is dependent on the local execution context. Many classes rely on utilities, such as file streams, which cannot execute correctly in a distributed context. Issues, such as concurrency, must also be addressed in order for a distributed form of the application to run correctly. The `AtlasRemoteContext` aspects (which, as we describe in Section 4, have a different glyph to represent the multiplicity) comprise the behaviour needed to make the application distribution-safe.

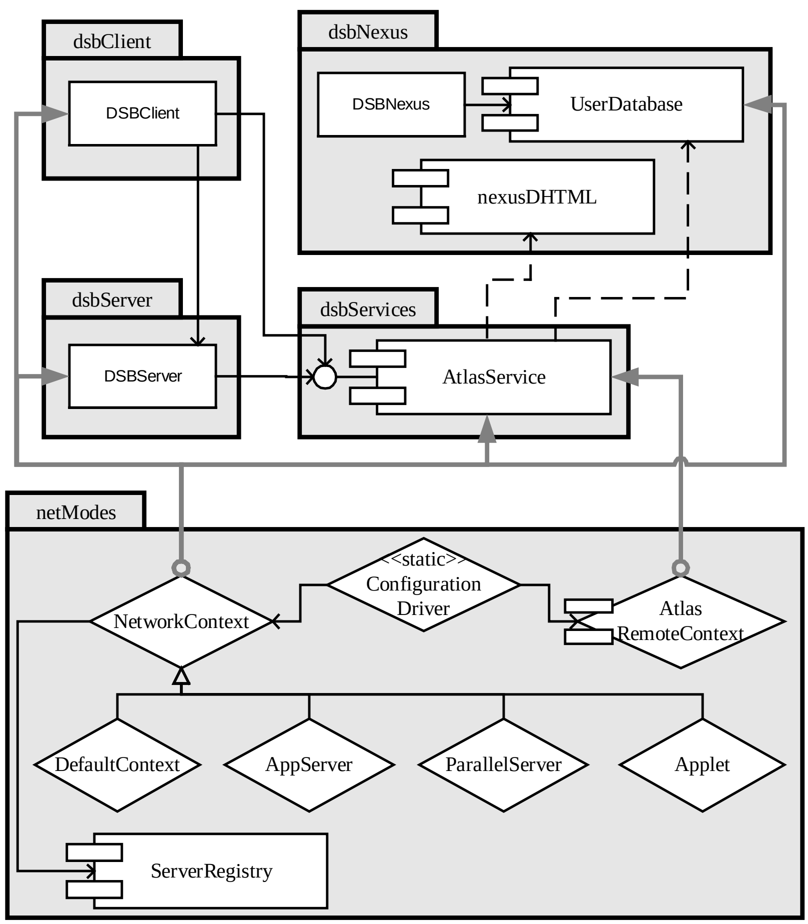


**Figure 5 Aspects for Network Contexts**

These aspects are responsible for overriding calls to context-sensitive objects, such as file streams, to enable them to run in a distributed context. Section 4.2 provides more detail on how this was accomplished.

The `NetworkContext` and its sub-aspects are dynamic aspects; this permits a context to be created and destroyed at run-time, making possible dynamic reconfigurations. For example, if an unusually high bandwidth was recognized in a browser client and the DSB was running in single server mode, the `applet` aspect could be instantiated and used by that client. The static `ConfigurationDriver` aspect weaves code into the system driver to create the dynamic aspects necessary for the system to run in the desired configuration. The `ConfigurationDriver` also adds code to create the `AtlasRemoteContext` aspects.

Without aspects, providing support for the multiple contexts would have required a complex inheritance structure with multiple classes interacting to provide different contexts. The support provided by the aspects makes it easy to build the system to work in a default context; support for other contexts can then be layered in as desired. As we discuss later (Section 4.4), separating this support into aspects also helps debug these contexts because it is possible to add in and remove support, facilitating the isolation of faults. This use of aspects also means that the base Atlas code has no knowledge of configurations or distributions, greatly improving the readability, and hopefully the maintainability, of the code.

### 3.2 Aspects to Extend Design Patterns

Several design patterns were used in the design of Atlas, including the creational Builder and Factory patterns, the structural Composite and Façade patterns, and the behavioural Chain of Responsibility and Strategy patterns [GHJV96]. We found it helpful to use aspects to extend the functionality of some of these design patterns.

As one example, consider our use of the Builder pattern (Figure 6). The `PageBuilderCommon` abstract class contains the common functionality associated with building an HTML page, such as building a header and building a body for the page. Subclasses are used to build different pages that may be served back to the student, such as a course page or a page reporting an error.

We wanted to allow a student to be able to choose a different look-and-feel when using Atlas as part of setting their user preferences. One way to add this support might have been to layer a Decorator pattern on top of the Builder pattern. This approach would have required substantial changes to the classes in place. Instead, we used an aspect to extend the Builder pattern. The `PageBuildDecorator` aspect hooks into the method calls responsible for constructing and printing the web page to the browser. Once a web page is constructed, the aspect decorates the resulting tree representation with new information, such as font faces, text and table colors, and button images.

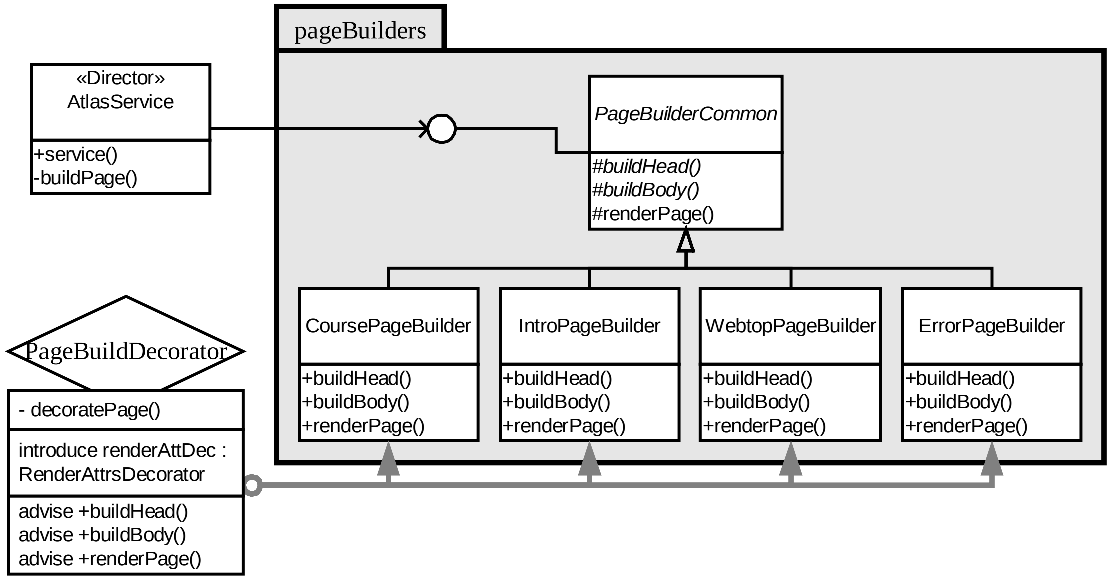


**Figure 6 Decorating a Builder Pattern with an Aspect**

The look-and-feel concern cross-cuts more than the `pageBuilders` package. The `webObjects` component is a library that can be used to create object representation of web pages. The Composite pattern that is used in the `webObjects` library also contributes to the application's look-and-feel. To provide a consistent implementation of this concern, we created a `WebLookAndFeel` aspect to hold the common data and functionality. `PageBuildDecorator` became a sub-aspect of `WebLookAndFeel`; `HTMLDecorator` was introduced to apply the look-and-feel concern to `webObjects`.

When we began development, we thought we might find it useful to code more of the design patterns as aspects. However, we found that the implementations of the patterns used in Atlas were sufficiently clean and sufficiently local that they would not benefit from encoding as aspects. It was when we wanted to layer two patterns together, as is the case with Builder and Decorator, that cross-cutting issues arose and use of an aspect was beneficial.

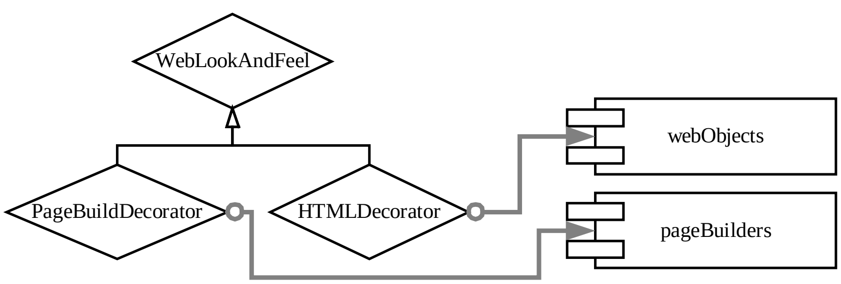


**Figure 7 Modularizing Look-and-Feel**

### 3.3 Aspects in Development

The influence of aspects in the development of Atlas reaches beyond the system structure. Aspects were also used to address two issues that we perceived might be problematic during the development of the system: debugging and tracing.

Since Atlas runs as a Servlet, debugging within a programming environment or by means of console print statements is not effective. Debugging becomes even harder when Atlas is run as a distributed system. We found the easiest way to debug Atlas was to sprinkle debugging code through the system that wrote to a specific output window or file. We used an aspect, `CallTracer`, to modularize this debugging code (Figure 8). This dynamic aspect advises all methods in a package to which it is attached, such as `dsbServices` or `dsbServer`. It uses common services provided by a super-aspect, `TracerAspect`, to write information about method entries and exits to another process. Making this aspect dynamic means that different kinds of tracing parameters can be set for different kinds of objects in the system.

We were also concerned about tracing the performance of Atlas. In Atlas, web pages are represented as objects; a relatively large number of objects—typically over 50—are used to represent a single page. The `PerformanceMonitor` dynamic aspect monitors the number of objects instantiated by advising constructors and maintaining the appropriate statistics.

It has not turned out that these aspects are as useful as we anticipated. The performance of the system has met the requirements, so tracing of object creations has not yet been important. The use of the network context aspects to encode configurations has kept the base Atlas code simple enough that we have not typically found it necessary to use the `CallTracer` aspect. Nonetheless, there are two basic benefits of encoding this support as aspects. First, when needed, the support is there and modularized without having polluted the base Atlas code. Second, as we discuss in Section 4.1, these aspects hold promise for being reused in other system developments.

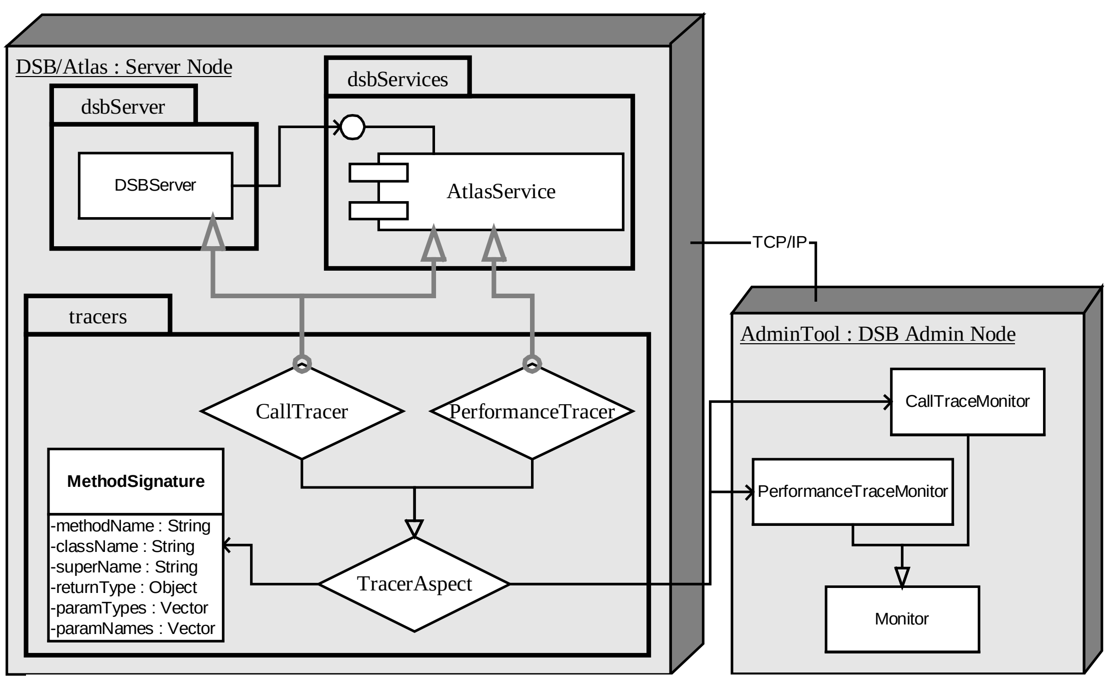


**Figure 8 Tracing and Performance Aspects**

# 4 Aspects in Practice

The process of constructing an aspect-oriented system with AspectJ is similar to that of an object-oriented development. However, the ability to represent cross-cutting concerns as aspects changes the ways in which pieces of the software structure being formed interact. This extra flexibility is not always an advantage. We found it helpful to design and implement the aspects to have certain forms of interaction with the basic class structure of the system if software qualities such as understandability and modifiability are to be achieved. In this section, we describe some guidelines and techniques we used to make it easier to work with aspects.

## 4.1 Aspect/Class Associations

When designing and implementing with aspects, we have found it useful to think about the *knows-about* relation between aspects and classes. An aspect knows about a class when the aspect names the class. A class knows about an aspect if it relies on the aspect to provide it state or functionality before the class can be compiled. Based on this knows-about relation, four different associations may arise between aspects and classes (Table 1).[3]

In a *closed* association, neither the aspect, nor the class, knows about the other. A closed association would be useful for a tracing aspect, such as the `CallTracer`, discussed in the last section. Creating such a general aspect that could be applied to multiple packages in a system at once would require support, not present in the current AspectJ, to wildcard package names.[4] Aspects formed with this kind of association would have the advantage of being easy to understand; the class and aspect could also be considered and manipulated independently. Closed associations would also support the definition of reusable aspects.

**Table 1 Aspect Associations Based on Knows-about Relation**

| Association Link | Flow of "knows-about" information | Benefits/Problems |
|---|---|---|
| **Open** | Arbitrary | – Compromised understandability, reusability |
| **Class-directional** | Aspect knows about the class but not vice-versa | + Easier to understand both classes and aspects<br>+ Classes are more reusable |
| **Aspect-directional** | Class knows about the aspect but not vice-versa | + Aspects are likely more reusable |
| **Closed** | Neither the aspect nor the class knows about the other. | + Easier to understand both classes and aspects<br>+ Aspects are reusable |

[3] In this discussion, we focus on the interaction between a single aspect and a single class. The concepts we discuss generalize to a single aspect acting on multiple classes.
[4] The current AspectJ limits the use of wildcards to class and method names; aspects using a closed association are thus limited to one package.

At the other end of the spectrum is an *open* association in which both the aspect and the class know about each other. As an example, consider the `PageBuilder` class, which is responsible for building a web page to be served to a student. `PageBuilder` contained a method responsible for printing the contents of the generated web page, called `printPage`, to the web browser (Figure 9a). When Atlas is running in a distributed network context, `ResponsePage` is a distributed object. Writing directly and often to a distributed object would have had a negative performance impact on Atlas. As a result, an aspect was introduced to allow writes to a local `ResponsePage` to be buffered; batch operations are then used to affect the actual distributed object, as shown along with the modified `PageBuilder` code in Figure 9b. This solution is not satisfactory for a number of reasons: `printPage` contains knowledge of context (shown by a solid arrow in Figure 9b); and the `PageBuilder` can no longer be understood, compiled, or tested without the `remotePrintWriter` aspect. Separation of concerns is not achieved.

To achieve a cleaner, modular structure, we evolved the association between the `remotePrintWriter` aspect and the `PageBuilder` class to be a *class-directional* association (Figure 9c). This category captures the case when the aspect knows about the class, but the class does not know about the aspect. In this case, `PageBuilder` is no longer aware of the aspect that acts upon it. (Note the removal of context information in the method in Figure 9c.) The `PageBuilder` class can be developed and tested independently with some default functionality. The aspect serves to extend the functionality of the class. This category permits reuse of the class.

The final category is the *aspect-directional* association in which the class knows about the aspect but the aspect does not know about the class. This association is not possible in the current version of AspectJ. Such an association might arise if a class or object requested a service from an aspect. We did not require such functionality in Atlas.

In the early development of Atlas, open associations arose often. However, as the system evolved in complexity, we began to set a policy of using only class-directional associations. (Closed associations would have been helpful in the tracing case but were not possible to express.) This policy was set to improve the understandability, modifiability, and testability of the classes.

Table 2 summarizes the aspects used in Atlas. The table provides a brief description of the code in each aspect, lists the classes supporting each aspect, and identifies whether the aspect is static or dynamic.

A - Initial Method

```
public void printPage( Page responsePage, PrintWriter browserWriter ) {
       ResponsePage.printContents( browserWriter );
}
```

B - First Version of remotePrintWriter Aspect

```
aspect remotePrintWriter {
    // Create a PrintWriter that prints into a string
    java.io.StringWriter stringWriter = new StringWriter();
    public java.io.PrintWriter remoteWriter
        = new PrintWriter( new BufferedWriter( stringWriter ) );
    // Introduce it into the given class
    introduce java.io.PrintWriter PageBuilder.remoteWriter;

    introduce private PageBuilder.printRemotePage( * ) {
        // Print the web page into a string
        responsePage.printContents( remoteWriter );

        // Make the remote call
        dsbClient.printToBrowser( remoteWriter.getBuffer() );
    }
}
```

```
public void printPage( Page responsePage, PrintWriter browserWriter ) {
    // Print the page to the web browser according to context
    if ( currentContext.equals( "local" )
        responsePage.printContents( browserWriter );
    else
        printRemotePage( servlet, responsePage );
}
```

C - Second Version of Aspect and Method

```
aspect remotePrintWriter {
    // Create a PrintWriter that prints into a string
    java.io.StringWriter stringWriter = new StringWriter();
    public java.io.PrintWriter remoteWriter
        = new PrintWriter( new BufferedWriter( stringWriter ) );

    advise void PageBuilder.PrintPage( * ) {
        static before {
            thisObject.browserWriter = thisAspect.remoteWriter;
        }
        static after {
            dsbClient.printToBrowser( remoteWriter.getBuffer() );
        }
    }
}
```

```
// The unaltered printPage method from PageBuilder
public void printPage( Page responsePage, PrintWriter browserWriter ) {
       responsePage.printContents( browserWriter );
}
```

**Figure 9  Various Forms of Aspect-Class Associations**

**Table 2 Aspects in Atlas**

| Aspect (lines of code) | Description of the Aspect Structure | Supporting Classes |
|---|---|---|
| **Driver**<br>• **ConfigurationDriver (76)** | A static aspect that advises code into system drivers. | |
| **Network Context**<br>• **Configuration**<br>• **NetworkContext (38)**<br>• **DefaultContext (24)**<br>• **SingleServer (164)**<br>• **MultipleServer (185)**<br>• **Applet (TBD)** | These dynamic aspects use both the advise and introduce constructs. | `ServerRegistry`<br>• 6 classes |
| **AtlasRemoteContext**<br>• **AtlasUserDatabase (37)**<br>• **CourseRegistry (33)**<br>• **FileReader (34)**<br>• **PageFileReader (36)**<br>• **PageBuildHandler (35)** | These dynamic aspects use the reassociation policy via advise constructs (described in Section 4.2) to affect the behaviour of objects in the `AtlasService` component. | |
| **Look and Feel**<br>• **WebLookAndFeel (32)**<br>• **PageBuildDecorator (105)**<br>• **HTMLDecorator (76)** | These dynamic aspects also use both the advise and introduce constructs. | |
| **Tracing**<br>• **LogWriter (76)**<br>• **CallTracer (124)**<br>• **PerformanceTracer (96)** | These dynamic aspects also use both the advise and introduce constructs. | `MethodSignature`<br>• 1 class<br>Admin GUI Component<br>• 4 classes |

## 4.2 The Aspect-Class Interface

As introduced in Section 2.3, AspectJ provides two basic constructs to describe how an aspect affects a class. The `introduce` construct introduces new state or new functionality into a class. The `advise` construct introduces new functionality before or after particular methods of classes. The use of these constructs forms an interface between an aspect and a class.[5]

Similar to the categories of associations discussed above, in the early development of Atlas, we did not pay much attention to this interface. However, as Atlas grew to be over 50 classes and aspects, it became harder and harder to understand and test classes because it was simply difficult to reason about how all of the code fit together. To help manage this complexity, we began to restrict the form of the aspect-class interface we used.

Returning to the issue of making Atlas run in distributed configurations, we found the kind of aspect code we presented for `RemotePrintWriter` in the previous section (Figure 9b) did not scale well. This approach essentially focuses on making classes behave properly in a distributed context by considering each method of the class. Changes to policies in distribution could thus still

[5] As before, although an aspect can interact with more than one class at a time, we analyze the interface with respect to one aspect and one class.

affect a large number of code points. Additions of new functionality to classes required code additions to be made to the aspects. Even though these code points were isolated within aspects, it was still somewhat onerous to affect the necessary changes.

Instead, we moved to an interface policy of *reassociation*. In this policy, an aspect overrides members of classes to alter the behaviour of objects of the class. For example, in the `UserManager` class of Atlas, which is responsible for manipulating user data, the database is represented by a member variable, `userDbase`. By default, this member accesses a local database. To make an object of this class access the database remotely, we introduced an advise weave on the constructor of the class:

```
package netModes.remoteContextAspects;

public aspect AtlasUserDatabaseAspect {
advise * dsbServices.atlas.aUserManager.UserManager.UserManager() {
      after {
         // Reassociate the "userDbase"
         userDbase = (AtlastUserDatabaseI )
            com.objectspace.voyager.Namespace.lookup (
               <Global name of database> );
      }
}
```

This advise weave rebinds the `userDbase` from the local to the remote context. In essence, the weave acts as a factory for creating the member. We used this reassociation mechanism for the code in Atlas that performed actions such as file I/O and database lookups. This policy simplified both the class code and the aspect code. All aspect code for the `AtlasService` part of the code base uses the reassociation interface policy.

Although reassociation was a sufficient policy for the `AtlasService`, the DSB code required the use of more general aspect code that uses both introduces and advises. To help manage the complexity of this code, we employed two aspect style rules. The first rule was to not allow exceptions introduced by a weave to percolate out from the weave; that is, if the code being introduced into a method could raise an exception, it was wrapped in a `try` block that handled the exception. The second rule was to ensure that before and after advise weaves did not alter the pre- and post-conditions of a method. Together, these two rules ensure that an aspect does not change the default interface and functionality of a class. Because the interface is unchanged, the application of an aspect to a class will not affect how the class fits into the existing class structure. Because the functionality of a class is not changed, the aspect does not modify contracts between client and supplier methods in the existing class structure.

The combination of these rules and our use of class-directional aspects make it easier to understand, debug, and test Atlas. By default, the base Atlas code functions in the *single server* context. If the system does not function correctly with another context aspect applied, the fault can be more easily isolated because the kind of actions performed by aspects are constrained.

## 4.3 Aspect Models and Notation

One important decision that a developer makes when building an object-oriented system is the structure of the system classes. The expression of this structure is typically referred to as an object model. When building an aspect-oriented system, the object model is still of central importance. In addition, a developer must choose an aspect model and describe how this model interacts with the object model.

An aspect model consists of both aspects and classes. The classes serve to support the implementation of the aspects. For example, in Figure 5, the `AppServer` aspect and the `ParallelServer` aspect both rely on a `ServerRegistry` class to provide registration services. A developer defining an aspect model is faced with a similar set of choices as a developer defining an object model. Similar to objects, aspects may also be related through inheritance, aggregation, and association links. We do not yet have enough experience to offer much advice on how to choose between these different options when defining an aspect model.

Choosing the design of the aspect model is one challenge; communicating the design is another. To date, we have been focusing on capturing the static structure of aspects: how the aspects relate to each other and how they relate to the class structure. Figure 10 summarizes this notation. This notation allows a developer to depict aspects, the kinds of associations they have with classes, and the structure of classes on which they rely. Within an aspect diamond, a developer can describe the structure of an aspect (its members), as well as the advise and introduce constructs in the aspect. The `aspectComponent` glyph is used to represent a collection of aspects, or an aspect with supporting classes.

This simple notation has been effective in supporting communication of the basic ideas and in facilitating discussion of choices in the design of aspect models.

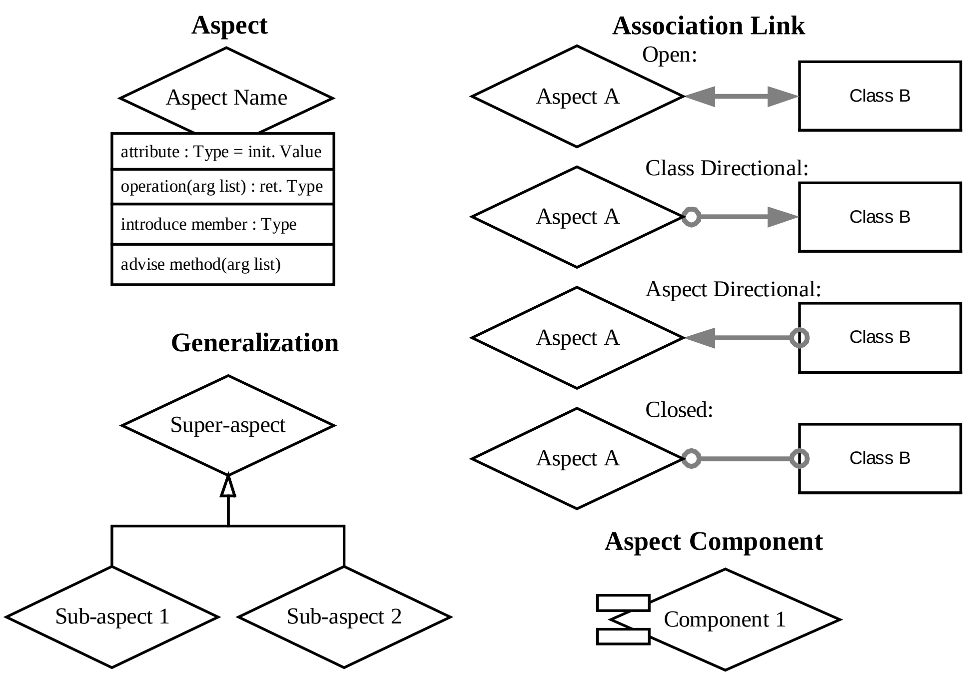


**Figure 10  Notation for Aspects**

## 4.4 Implementing with Aspects

The previous sections have discussed some of the choices that a developer faces when designing and implementing with aspects. In this section, we try to give a sense of some of the nitty-gritty details of working with AspectJ, in the context of the Microsoft Visual J++™ 6.0 interactive development environment.

We faced two main challenges in implementing with AspectJ within this environment. First, as the system grew larger, the cost of weaving became too large to be performed as an iterative edit-weave-compile-debug cycle. Second, we bumped up against limitations on the number of files that could be weaved at any one time in earlier versions of AspectJ. The second problem has been solved in newer releases of AspectJ.

To allow for more independent edit-compile-debug and edit-weave-compile-debug cycles, we separated the aspect code, the class code, and resulting woven code into three different project solutions in Visual J++: `aspect`, `system`, and `woven`.

The `aspect` solution comprises the `networkConfiguration` and `general` packages. The `networkConfiguration` package contains the aspects and packages associated with the DSB. The `general` package contains the debugging and performance tracking aspects. Batch files were created to perform the weaves for these aspects. Separate batch files were used to allow independent application of the different aspects.

The `system` solution comprises the packages for the `AtlasService`. These classes can be compiled and run separately from the application of any aspect. Maintaining this independent functionality was of significant benefit; requiring weaves to test the core Atlas functionality would have been immobilizing. Separating the woven code out as a Visual J++ solution meant that the solution could be compiled independently within the environment after the weave. This set-up also made compile problems resulting from a weave easier to debug.

In setting up this configuration, one of our hopes had been that we would be able to "physically" separate the core functionality of Atlas, the `AtlasService`, from all aspect code. We did not quite achieve this goal. The snag we hit was that various forms of drivers are needed to start-up different network contexts in the DSB. In particular, the `DSBServer` and `DSBNexus` components must sometimes execute as independent processes. As mentioned earlier, we weave the registrations needed for the dynamic aspects into the drivers. The simplest approach was to place these drivers into the `system` solution, in essence, causing us to leak information about DSB into the object model. This snag could have been overcome by building a more sophisticated driver infrastructure. It is not clear that a more complicated infrastructure is desired because the presence of sub-system boundaries in the object model facilitated execution.

# 5 Discussion

The descriptions of our use of aspects in the previous section illustrate some of the changes that occurred as we gained more experience with the technology. In this section, we provide some higher-level perspective, discussing some of the lessons we learned over the course of the project, and discussing some of the more difficult challenges facing others who might decide to use the technology.

## 5.1 Lessons Learned

If we were to start building another system with aspect-oriented technology, here are some guidelines we would apply.

- *Try to limit the knows-about information in the aspect-class association link (Section 4.1).* We found it easier to manage the evolution of our system when classes were not coupled to aspects. Class-directional aspects facilitated the readability, modifiability, and reusability of class and aspect code in Atlas.

- *A reassociation policy, where an aspect acts as a factory, can simplify the extension of an object's behaviour (Section 4.2).* This policy kept the aspect code simple and clear; the aspect code had a well-defined scope of effect on the class code, making it easier to reason about and test.

- *Using dynamic aspects provides runtime configurability, but may complicate system set-up code (Section 4.4)*. We ended up using dynamic aspects much more often than we had first envisioned. Dynamic aspects provide more long-term flexibility and support more sophisticated runtime behaviour, but require the addition, or weaving in, of registration code. The registration code does not always fit easily into the existing system structure.

- *Try to maintain a stand-alone object model, which aspects extend (Section 4.4).* From our object model, we could build an executable system. This configuration enabled a workable edit-compile-debug cycle since weaving was optional. This configuration also helped in debugging the system: if the default configuration worked and there was a problem when an aspect was woven in, it was easier to isolate the fault.

## 5.2 Outstanding Issues

*Aspect-oriented Design.* By far the hardest decision facing a developer working with aspect-oriented technology is determining what should be an aspect and what should be a class. In the beginning of our development, we thought we would have many more aspects. In many cases, we started implementing an aspect and then found that some straightforward changes to our object model could accomplish the same goal more effectively. Our current approach to aspect-oriented design has been to start with an initial object model, and then to incrementally consider cross-cutting additions as aspects, using the concepts of aspect associations and aspect-class interfaces discussed earlier. More policy and style rules along these lines are needed to help developers make appropriate choices.

*Aspect Notations*. In this paper, we have introduced a straightforward approach to diagram some "aspects" of aspects. This notation captures only the static structure of an aspect. We have not yet determined a tractable way of illustrating dynamic aspects and the scope of their effect on the object model. Illustrating the scope of effect of a dynamic aspect seems useful: it is hard to reason about the effect without considering a substantial amount of code.

*Aspect Scope*. Aspects are helpful because they allow a developer to modularize cross-cutting concerns. Once a concern is modularized as an aspect, it can be tempting to apply that aspect across more parts of a system. For example, modularizing look-and-feel as an aspect in Atlas and applying it to both the building and representation of web pages was a benefit. It was tempting to extend this aspect to handle look-and-feel for the administrative GUI for Atlas as well. But, we decided that there was no reason to couple, however loosely, the look-and-feel for of the pages and the GUI. The problem lies not necessarily in the original development, but in later interpretations of the use of the aspect by other developers and maintainers. The trade-offs of scoping aspects to affect more or less of a system are not clear.

## 6 Summary

This paper has described the development of a web-based learning environment called Atlas that was built using aspect-oriented programming as provided by AspectJ. In describing the system, we have focused on our experiences with aspect-oriented programming, synthesizing some lessons we have learned in applying this new technology. Although we faced some small hurdles along the way, this combination of technology helped us build a fast, well-structured system in a reasonable amount of time.

## Acknowledgements

George Tsiknis and Ian Cavers helped design Atlas from the perspective of a web-based course tool. We thank Rob Walker, Gregor Kiczales, and Martin Robillard for helpful comments on an earlier draft of this paper. Liz Kendall provided insight on the aspect model notation and on the intersection of aspects and patterns.

## References

[BJR98] G. Booch, I. Jacobson and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[Bre98] E.A. Brewer. Delivering High Availability for Inktomi Search Engines. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, p. 538, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company. 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Maschusetts, 1996.

[KLM+97] G. Kiczales, J.Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J.Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-oriented Programming* (ECOOP'97), pp. 220-242, 1997.

[LK95] C. Lopes and G. Kiczales. Recent Developments in AspectJ™. In *ECOOP '98 Workshop Reader*, 1998.

[Obj] ObjectSpace web page: http://www.objectspace.com/Products/voyager1.htm