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

Mik Kersten and Gail C. Murphy
University of British Columbia
2366 Main Mall, Vancouver, BC
V6T 1Z4, Canada
(650) 888-3483

{mkersten,murphy}@cs.ubc.ca

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 obstacles along the way, this combination of technology helped us build a fast, well-structured system in a reasonable amount of time.

Keywords

Aspect-oriented programming, software engineering practices, web-based applications, distributed systems.

1. INTRODUCTION

The Advanced Teaching and Learning Academic Server (Atlas) is a software system that supports web-based learning. Using Atlas, students can register for courses and can navigate through personalized views of course material. In 1998, an initial version of Atlas was developed in C++ [6]. Although functional, this version suffered from typical initial version problems. In particular, on-the-fly design changes had made the code base difficult to change, maintain, and test.

Hindsight suggested that some of the difficulties faced in the initial implementation might be addressed effectively by the emerging aspect-oriented programming [4]. Aspect-oriented programming provides explicit language support for modularizing code belonging to design decisions that cross-cut a program. To investigate whether aspect-oriented programming could help, we undertook a new development of Atlas using Xerox PARC's AspectJ™ [5], which provides an aspect-oriented extension to Java™ [3].

In this paper, we describe what it was like to build a moderate-sized (180 class) system using AspectJ. Section 2 provides background on Atlas and on AspectJ. Section 3 presents the aspects used for both the system and its development. Section 4 discusses the style of aspect-oriented programming we used, considering categories of aspects that arose, trade-offs we found in using aspects from the different categories, and policies we employed to achieve our goals. Throughout the paper, a straightforward notation based on UML [1] is used to express the design of aspects in our system; this notation is described in Section 4.3.

For Atlas, the use of aspect-oriented programming paid off: we were able to develop a system that meets its functional requirements and that shows promise for meeting the non-functional requirements of maintainability and modifiability. In Section 5, we summarize lessons we learned through the Atlas development and describe outstanding issues associated with using aspect-oriented programming.

2. BACKGROUND

2.1 Atlas Requirements

To use an installation of Atlas, a student accesses a web page and logs into the system. Once logged in, Atlas presents the student, via the web browser, with a customized desktop-like interface (Figure 1). This desktop provides a student with access to course tools (interactive user-tailored courses, quick references, and a manual), support tools (such as a calculator and an HTML editor), 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.

An installation of Atlas may be accessed concurrently by a few students or by thousands of students. Some students may be new users of Atlas; others will be accessing courses for which they have previously registered. Some students may have a high-speed local are network connection to the Atlas installation; others may

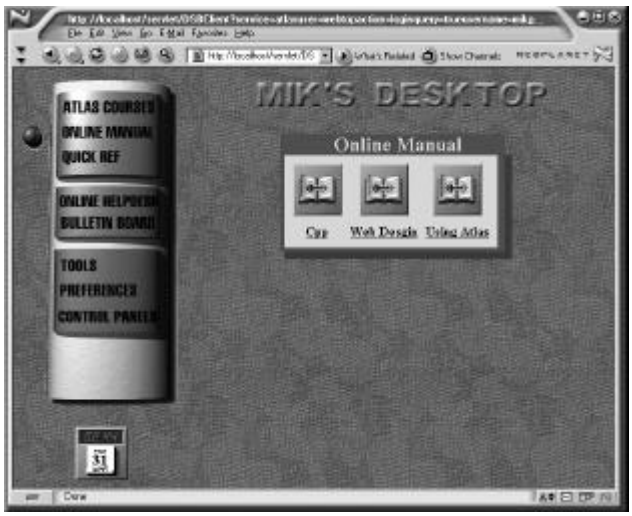

Figure 1 A Student's View of Atlas

be using a lower-bandwidth connection. To economically support a wide variety of environments, Atlas may be configured to run in one of four *network contexts* (Figure 2).

1. *Single server* context (Figure 2a). In this context, there is one server. Requests to Atlas are run in different threads on the server. This context is suitable for a modest number of users.
2. *Application server* context (Figure 2b). In this context, there are two servers: a web server and an application server. The web server forwards application requests to the application server. This configuration helps support higher user loads.
3. *Parallel application server* context (Figure 2c). Atlas is distributed over a number of servers to exploit the inherent parallelism associated with requests through the web. Requests are load balanced across the application servers. This configuration provides a better response rate under heavy load.
4. *Applet* context (Figure 2d). To achieve the performance and responsiveness associated with desktop applications, most of the Atlas functionality in this context runs in the web browser. This context requires a centralized database to ensure that a student with multiple browsers open is provided a consistent view. The database functionality may be run on the web server.

2.2 AspectJ

Aspect-oriented programming (AOP) is intended to help software developers more cleanly separate concerns in their source code. In object-oriented programming, the code for a concern is typically spread across multiple methods in multiple classes; individual methods often contain a tangle of code from different concerns. In aspect-oriented programming, code for a concern—an *aspect*—can be separated from the classes to which it applies. An aspect modularizes the code for a concern and describes how the concern code 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

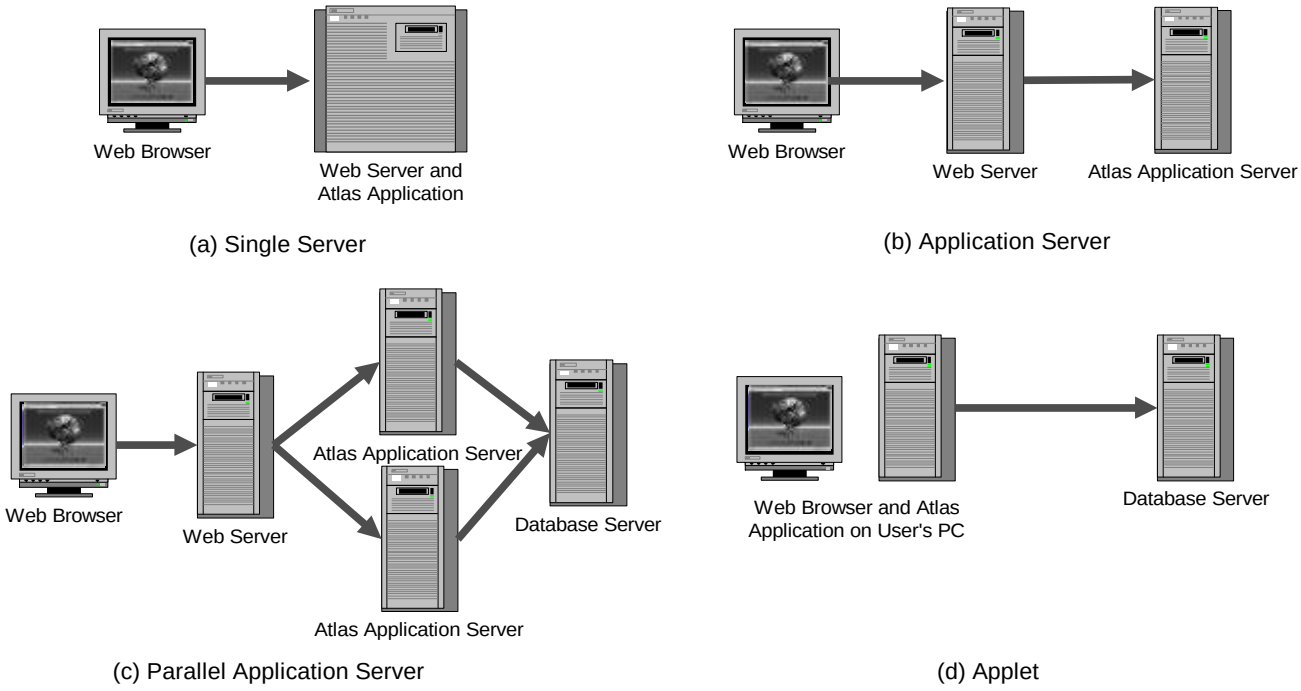

Figure 2 Network Contexts

`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, `StaticCallTracer`, that uses both the introduce and the advise constructs to add tracing code into an existing class. The introduce construct is used to add a variable called `writer` into `AnApplicationClass`. The `writer` variable is initialized to `System.out`. The advise construct is used to add the tracing code to the start of all methods on `AnApplicationClass;` the code will write the name of a method when it is entered. This aspect gathers together code that would otherwise be spread throughout `AnApplicationClass`. If tracing is not needed, we can simply not apply the aspect to the system.

In the `StaticCallTracer` 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, `DynamicCallTracer`. In comparison to the static aspect, this aspect has a local `writer` variable, a constructor, and a non-static weave (i.e., the `before` is not preceded by the `static` modifier). 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. The behaviour described in the non-static weave will only apply to objects that have been registered with the aspect. Dynamic aspects can be used to support runtime configurable behaviour.

To create a system using AspectJ, a developer uses a weaver tool

A - advise and introduce

```
import java.io.*;
aspect StaticCallTracer {
    introduce PrintWriter AnApplicationClass.writer
       = new PrintWriter( System.out );
    // Advise all methods on AnApplicationClass by using wildcards
    advise * AnApplicationClass.*(..) {
        static before {
            writer.println( "ENTERING " + thisJoinPoint.methodName );
        }
    }
}
```

B - dynamic aspect and driver class code

```
import java.io.*;
aspect DynamicCallTracer {
   PrintWriter writer;

    public DynamicCallTracer( ) {
        writer = new PrintWriter( System.out );
    }

    advise * AnApplicationClass.*(..) {
        before {
            writer.println( "ENTERING " + thisJoinPoint.methodName );
        }
    }
}
```

```
public static void main( String[] args ) {
    AnApplicationClass anObj =  new AnApplicationClass(); // create an object
    // calls to anObj here will not be traced
    DynamicCallTracer traceAspect                        // create the aspect
        = new DynamicCallTracer();
    traceAspect.addObject( objA );                       // add the object
    // calls to anObj here will be traced
}
```

Figure 3 Sample AspectJ Code

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 source files that can then be compiled with a standard Java compiler.

Several versions of the AspectJ pre-processor were used in the development of Atlas; specifically, versions 0.2.0beta4 through beta10. We have upgraded the examples shown in this paper to conform to the syntax of AspectJ 0.3.

3. ATLAS AND ASPECTS

Atlas uses aspects for several different purposes: to support different architectural configurations, to implement a design pattern, 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 Sun Microsystem's Java Servlet specification,[1] 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). The Atlas functionality is instantiated into the DSB framework. Aspects are used to support configuration of contexts and to tailor the behaviour of the Atlas functionality in those contexts. First, we describe the DSB; then we describe how aspects are used to support the various configurations.

3.1.1 The Distributed Server Broker Framework

The DSB framework consists of three main components. The `DSBClient` handles HTTP requests forwarded from the Java web server and determines to which application the requests should be forwarded. The `DSBServer` provides access to the applications being served. The `DSBNexus` provides access to any databases and registries that may be used by an application.

Using Atlas with the DSB framework involves setting up the `DSBServer` to access `Atlas`, and configuring the appropriate Atlas databases in the `DSBNexus` component. Atlas functionality is accessed through the `AtlasService` class, which implements the `javax.servlet.Servlet` interface.

Configuring the DSB to run in a particular network context amounts to instantiating and configuring DSB components and the application (i.e., `AtlasService`) in the appropriate numbers on various nodes. Some functionality within Atlas must also be tailored to address performance concerns (see Section 4.1). The complexity of configuring a particular network context is proportional to the level of distribution in the context.

The simplest context is the *single server* context in which all components of the DSB framework and the `AtlasService` run on the web server. In the *applet* context, all components run on the user's machine, except the `DSBNexus` component, which must be centralized to allow for multiple browser access.

In the *application server* context (shown in Figure 4a), all three DSB components and the `AtlasService` components run on a separate server from the web server. This context requires communication between the `DSBClient` and `DSBServer` to cross a server boundary: we use ObjectSpace Voyager[2] to provide transparent access to the distributed Java objects running on different servers.

The *parallel application server context*, which exploits parallelism of HTTP requests, is more complex (Figure 4b). The `DSBNexus` component, which is centralized to ensure consistency, is run on a separate node. In this context, new instantiations of the `AtlasService`, each of which is run on a different server, 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.

Supporting a particular network context using a purely object-oriented approach would be reasonably straightforward. Supporting all of the network contexts using a purely object-oriented approach would be somewhat more complicated. One possible way to proceed would be to introduce classes to represent each context; each class would implement a particular interface. To instantiate and communicate with `DSBServer` objects, the `DSBClient` would delegate through an object representing the appropriate network context. Similarly, the `AtlasService` would delegate through an appropriate context object to communicate with a `DSBNexus` object. This approach would require a means to ensure compatible context objects were being used across the system. For instance, you would not want the `DSBClient` to be acting in an applet context while the `AtlasService` was acting in a parallel server context.

3.1.2 Aspects for Network Contexts

The need in a purely object-oriented approach to make changes across the system design suggested the use of aspects to support configuration of the network contexts. 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.[3] The `NetworkContext` 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 *application server* context, extending the system to handle

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

[2] See http://www.objectspace.com/Products/voyager1.htm.

[3] 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.

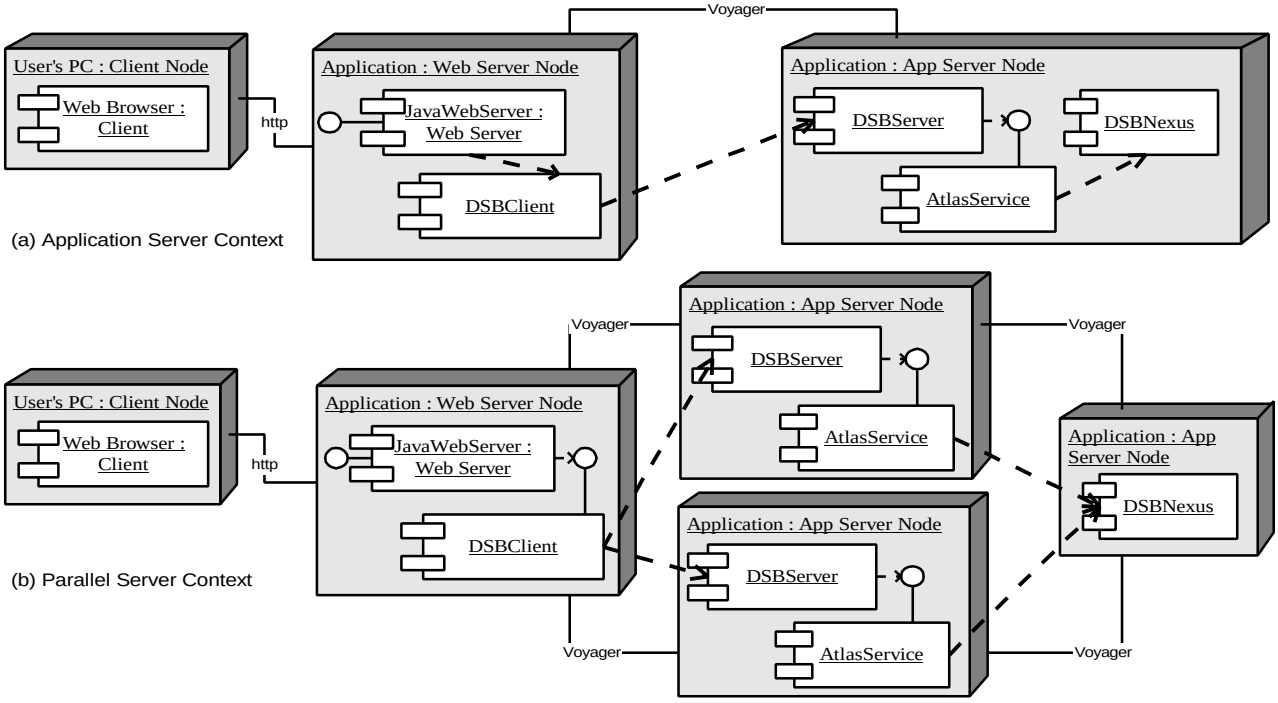

Figure 4 Two Network Contexts

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. Both the `AppServer` and the `ParallelServer` aspects 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.

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 aspects described by `AtlasRemoteContext` (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. 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 while the DSB was running in single server mode, the `applet` aspect could be instantiated and used by that client. Originally a dynamic aspect, called `ConfigurationDriver,` was used to weave the code to create the `NetworkContext` aspects into the system class drivers. This approach permitted run-time reconfigurations of the network contexts. However, since the domain did not require frequent re-configurations a static `ConfigurationDriver` was favored for simplicity.

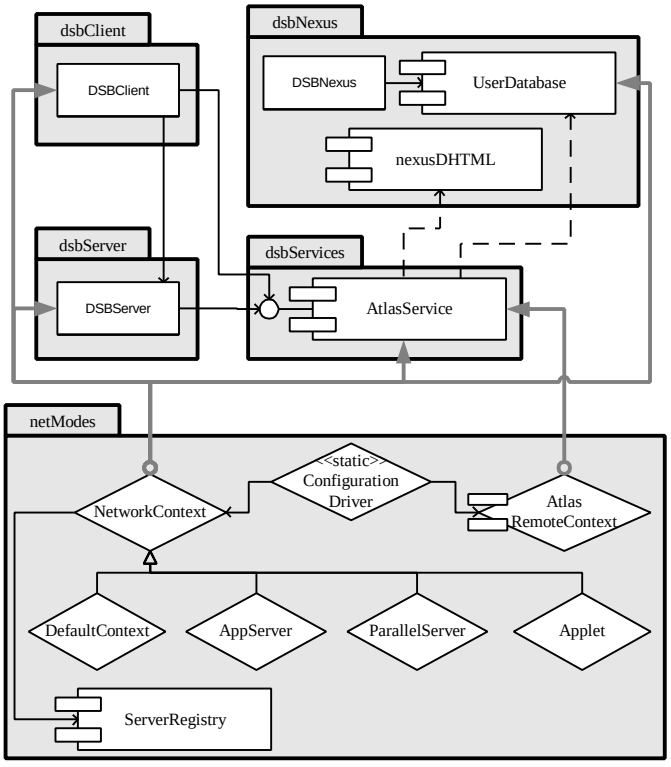

Figure 5 Aspects for Network Contexts

Aspects have made it possible to separate the code of the simple single server Atlas system from the other, more complicated, configurations. The separation has been helpful in debugging: support for a particular context may be added or removed, facilitating the isolation of faults. This separation also makes it easier to read through the code base; a developer can build an understanding of the basic Atlas functionality before tackling the issues of distribution and the various configurations.

3.2 Aspects and Design Patterns

Several design patterns were used in the design of Atlas, including the creational Builder pattern, the structural Composite and Facade patterns, and the behavioural Chain of Responsibility and Strategy patterns [2]. As implementation proceeded, we considered which of these patterns should be expressed as classes and which should be expressed as aspects. Since the patterns we were working with had little or no cross-cutting properties, we found these patterns were more easily expressed using classes.

Later in the implementation, a need arose to allow the student to choose a different look-and-feel when using Atlas. In particular, the web pages served to a student from Atlas needed to correspond to the look-and-feel preferences set by a student. One way to add this support was to apply the Decorator pattern. This pattern would interact with the Builder pattern that was being used to structure the construction of an HTML page.

The Builder pattern was implemented using the classes shown in Figure 6. The abstract `PageBuilderCommon` class contains the common functionality to build a page; the subclasses specialize the building process for the particular kind of page being built, such as introductory pages, course pages, "webtop" interface pages, and error pages.

Implementing a Decorator given this structure would have required substantial changes to the classes in place. In particular, changes would have been required to expose a representation of the page being built to allow decoration to happen at different parts of the building processes. These changes would have affected all of the concrete Builder subclasses.

The cross-cutting nature of these changes suggests the use of an aspect. As shown in Figure 6, 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 representation with new information, such as font faces, text and table colors, and button images.

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 representations of web pages. To provide a consistent implementation of this concern, we created a `WebLookAndFeel` aspect to hold the common data and functionality. The `PageBuildDecorator` aspect inherits from `WebLookAndFeel`; the `HTMLDecorator` aspect is introduced to apply the look-and-feel concern to `webObjects` (Figure 7).

The use of an aspect allowed us to layer Decorator-like functionality on top of the Builder pattern. This implementation approach limited the changes needed to the existing software structure and extended easily to other parts of Atlas in which the same concern arose. More experience is needed to determine when the implementation of such layered functionality is better expressed as an aspect and when it is better to pay the higher price of restructuring the code base.

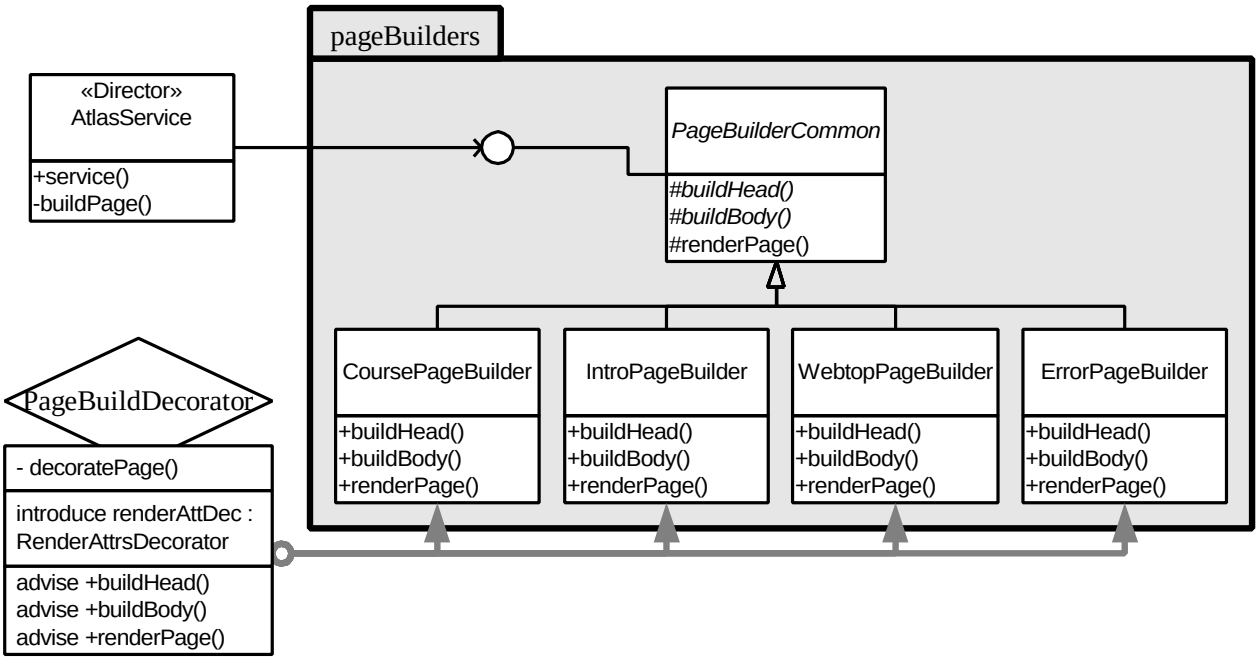

Figure 6 Decorating a Builder Pattern using an Aspect

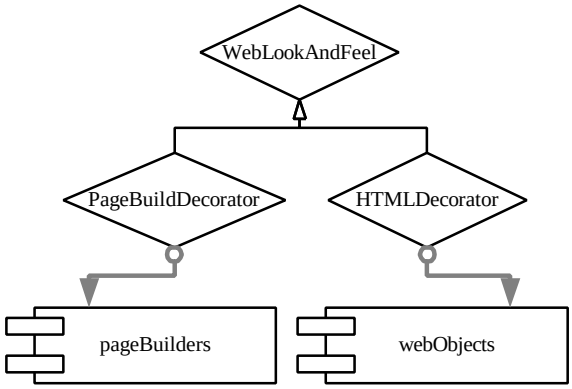

Figure 7 Aspects for Look-and-Feel

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 by means of console print statements or by means of a typical debugger is not effective. Debugging becomes even harder when Atlas is run across multiple nodes. We found the most convenient 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 applied. 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 monitoring 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 advises constructors to monitor the number of objects instantiated.

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 the `CallTracer` aspect has not been used extensively. 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.

4. ASPECTS IN PRACTICE

The process of constructing an aspect-oriented system with AspectJ is similar to that of an object-oriented development: classes must be defined, choices must be made about which interactions between the classes to allow, etc. With aspect-oriented programming, the developer is presented with new choices, such as determining whether new functionality should be added to a class or introduced into the class via an aspect.

The extra flexibility provided by aspects is not always an advantage. If too much functionality is introduced from an aspect, it may be difficult for the next developer—or the same developer a few months later—to read through and understand the code base. Or, it may be harder to restructure or extend the

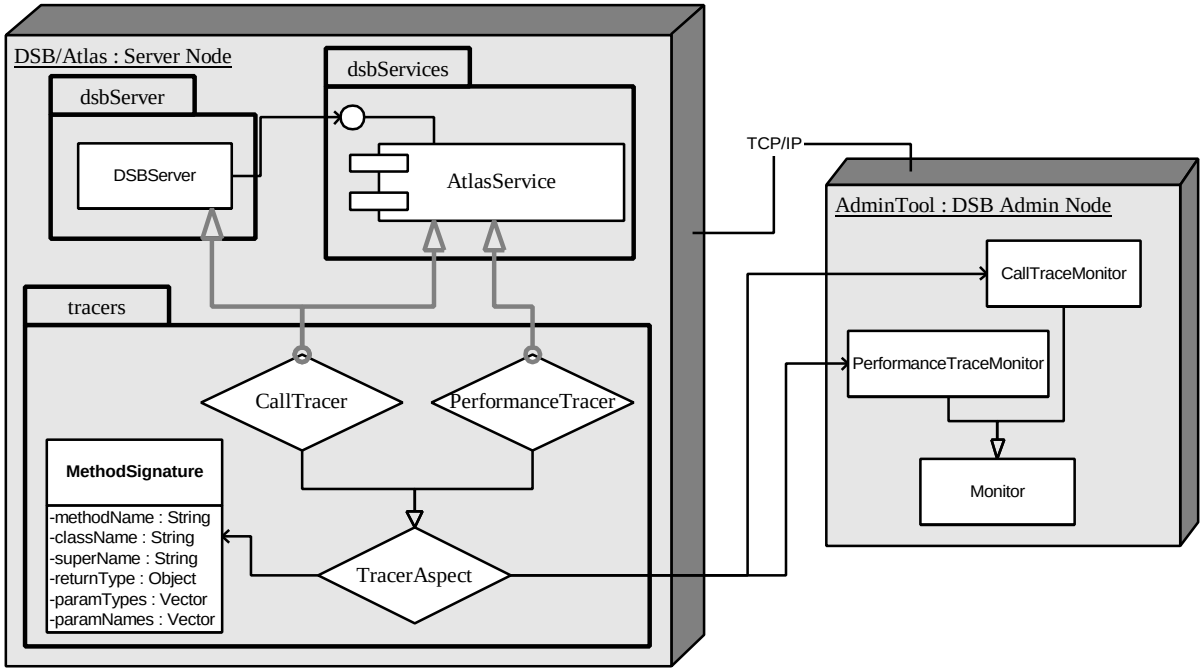

Figure 8 Tracing and Performance Aspects

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

Table 1 Aspect Associations Based on Knows-about Relation

functionality of a system if it impacts complicated weaves. 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).[4]

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 would require support to wildcard package names; the version of AspectJ we used limited the use of wildcards to class and method names. Aspects formed with this kind of association would have the advantage of being easy to understand and easy to reuse.

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 `PageBuildHandler` class, which is responsible for handling requests to generate pages. `PageBuildHandler` contains 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. Frequent operations on a distributed object would have a negative performance impact on Atlas. Our initial approach to address this performance concern was to introduce a variable within the `PageBuildHandler` class called `currentContext`, which indicated the active network context, and an aspect, `remotePrintWriter`, which added support for local buffering of the results of `printContents` to reduce the number of distributed operation invocations (Figure 9b). This approach required modification of the `printPage` method to test whether the current context involved distribution (shown by the arrow in Figure 9b) and if so, to use the newly introduced method. This solution is not satisfactory for a number of reasons: `printPage` contains knowledge of network contexts; and the `PageBuildHandler` class can no longer be understood, compiled, or tested without the `remotePrintWriter` aspect. Separation of concerns is not achieved.[5]

To achieve a cleaner, more modular structure, we evolved the association between the `remotePrintWriter` aspect and the `PageBuildHandler` 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, `PageBuildHandler` is no longer aware of the aspect that acts upon it. (Note the removal of context information in the method in Figure 9c.) The `PageBuildHandler` class can be developed and tested independently. The aspect serves to extend the functionality of the class. The class code remains simple and easy to understand. 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

[4] 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.

[5] We could have achieved the desired effect without modifying the `printPage` method by using an advise weave that performed the test of context and if the context was distributed, called the appropriate version and returned. We discuss in Section 4.2 why we wanted to avoid before weaves that return without running the body of a method.

A - Initial Method

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

B - First Version of remotePrintWriter Aspect

```
aspect remotePrintWriter {
    // Introduce a PrintWriter that prints into a string
    introduce java.io.PrintWriter PageBuildHandler.remoteWriter
        = new PrintWriter( new BufferedWriter( new StringWriter() ) );

    // Introduce a method that uses the remoteWriter
    introduce private PageBuildHandler.printRemotePage( Page responsePage ) {
        // 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( responsePage );
}
```

C - Second Version of Aspect and Method

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

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

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

Figure 9 Various Forms of Aspect-Class Associations

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 lists the aspects by category, providing the number of lines of code in each aspect, a description of the structure of the aspect, and a list of the classes needed to support the aspect. The table also identifies which aspects are static and which are dynamic.

Aspect	Lines of Code	Description of the Aspect Structure	Supporting Classes
Drivers ConfigurationDriver	 104	A static aspect that advises code into system class drivers.	None
Network Context NetworkContext DefaultContext AppServer ParallelServer Applet	 38 24 164 185 155	These dynamic aspects use both the advise and the introduce constructs.	Server Registry Package CircularQueue RemoteServiceContainer RemoteServiceRegistry ServiceInfoContainer
AtlasRemoteContext AtlasUserDatabase CourseRegistry FileReader PageFileReader PageBuildHandler	 37 33 34 36 35	These dynamic aspects use reassociation advises to affect the behaviour of objects in the `AtlasService` component.	None
Look and Feel WebLookAndFeel PageBuildDecorator HTMLDecorator	 32 105 76	These dynamic aspects use both the advise and the introduce constructs.	None
Tracing LogWriter CallTracer PerformanceTracer	 76 124 49	These dynamic aspects use both the advise and the introduce constructs.	Format Package MethodSignature Admin Interface Package AtlasAdminHost PerformanceMonitor TraceMonitor

Table 2 Aspects in Atlas

4.2 Aspect Style

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. When we began development of Atlas, we did not pay much attention to the style in which we used these constructs. However, as Atlas grew to be over 50 classes and aspects, it became more and more difficult to understand and test classes because the way in which we were using aspects made it hard to reason about how all of the code fit together. To help manage this complexity, we began to constrain and stylize our aspect code.

As an example, consider the kind of aspect code we presented for `RemotePrintWriter` in the previous section (Figure 9b). This code handles a performance issue specific to one of the network contexts by altering a method in which the issue arises. Often, handling such an issue was not restricted to one method. Instead, a number of methods in a class had to be advised by the aspect. We found that this approach did not scale well: changes in how the issue was to be handled affected a large number of code points, and additions of new functionality to classes required numerous code additions to the aspects.

Instead, we moved to aspect code that used an approach of *reassociation*. With reassociation, the aspect overrides a member of a class with a more specialized instance to provide desired behaviour. 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 as shown in Figure 10. This advise weave rebinds the `userDbase` variable 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.

Reassociation is not always a possible or desirable approach: the behaviour that is to be tailored may not be encapsulated by an object, or it may not be desirable for aspect code to modify private members of a class. However, for Atlas, this approach simplified and reduced the size of the aspect code, and made it easier to understand the effect of aspects.

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 three aspect style rules.

Rule #1: Exceptions introduced by a weave must be handled in the code comprising the weave.

This rule means that if the code being introduced into a method could raise an exception, it was wrapped in a `try` block that handled the exception.

Rule #2: Advise weaves must maintain the pre- and post-conditions of a method.

```
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> );
        }
     }
  }
```

Figure 10 Reassociation Advise

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.

Rule #3: Before advise weaves must not include a return statement.

This rule means that the code defined in the main body of the method is always run. We instituted this rule to make it easier to reason about the combination of a system and an aspect. One can read through the aspect code looking for how it augments and alters the basic system interactions, rather than having to reason about which basic system interactions might not occur because of a premature return.

Although these style guidelines do not apply to all situations, we found their use made it easier to understand, debug, and test Atlas.

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 can consist 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 the aspects relate to the class structure. Figure 11 summarizes this notation. This notation allows a developer to depict aspects and their interrelationships, as well as associations between aspects and classes. 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 aspect component 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.

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. 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. 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 two packages of aspects: `networkConfiguration` and `general`. The first package contains the aspects and packages associated with the DSB. The second package contains the debugging and performance tracking aspects. Batch files are used 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 independence of this functionality from the aspects was of significant benefit since 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

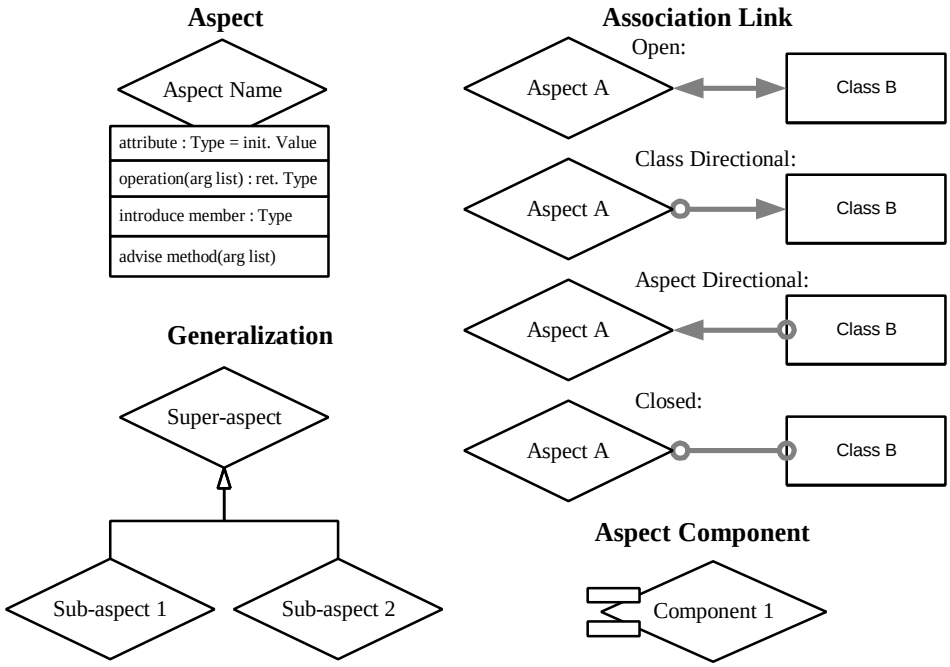

Figure 11 Notation for Aspects

the weave with result placed in `woven`. 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 entirely achieve this goal. The snag we hit was that various forms of class 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. The easiest way to achieve this is through separate class drivers. 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 obstacle could have been overcome by building a more sophisticated driver infrastructure.

5. DISCUSSION

The description of our use of aspects in the previous section illustrates 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 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

5.2.1 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 class-aspect associations and aspect style discussed earlier. More of these kinds of guidelines are needed to help developers make appropriate choices.

5.2.2 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 often hard to reason about the effect without considering a substantial amount of code.

5.2.3 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.

7. 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. The comments of the anonymous reviewers were valuable in revising and improving the paper.

8. REFERENCES

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

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

[3] Gosling,, J., Joy, B., and Steele, G. The Java Language Specification. Addison-Wesley, 1996.

[4] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In Proceedings of ECOOP'97 (Jyväskylä Finland, June 1997), Springer Verlag, 220-242.

[5] Lopes, C. and Kiczales, G. Recent Developments in AspectJ™. In ECOOP '98 Workshop Reader, Springer Verlag, 1998, 398-401.

[6] Stroustrup, B. C++ Programming Language. Addison-Wesley, 1986.