

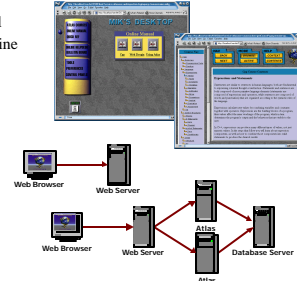
Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming

Mik Kersten and Gail C. Murphy
University of British Columbia

1

Web-Based Learning Environment

- Must support concurrent access by thousands of students
 - Register for courses
 - Browse course material
 - Interact with others online
- Performance is critical
 - Multi-tier architecture
 - Supports different tier configurations
- Size
 - Over 10K loc
 - 180 classes
 - 17 aspects



2

Gaining Experience with AOP

- AspectJ™ is an AOP extension to Java™
- Little experience is available building AOP systems
- We try to fill this gap by
 - Re-implementing a multi-tier application using AspectJ
 - Examining the aspects we used in building Atlas
 - Describing how they improved the design
 - Describing some policies we employed in Atlas to achieve our goals of maintainability and modifiability

3

Overview

Part I: Design concerns handled with aspects

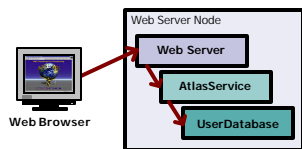
1. Tier configuration
2. Distribution support
3. Tracing and performance monitoring
4. Look-and-feel

Part II: Aspect Style

4

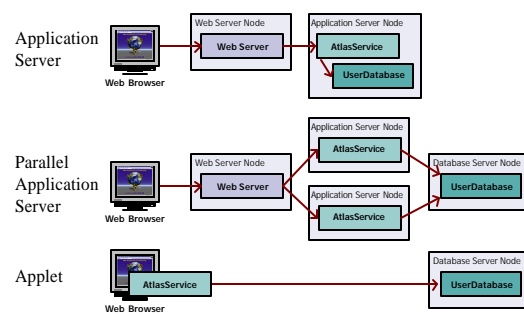
Concern 1: Tier Configuration

- Default configuration
 - User accesses Atlas from a web browser
 - Web server delegates request to the AtlasService Servlet
 - Atlas service handles the request using the UserDatabase



5

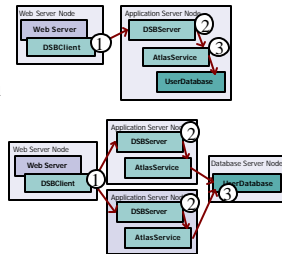
Tier Configuration (continued)



6

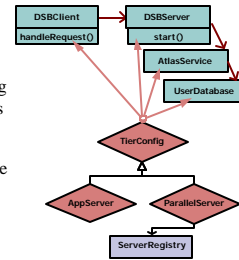
Changing Configurations

- The DSB is a generic utility, separate from *AtlasService*
- Change configurations by modifying connections between
 - DSBClient and DSBServer(s)
 - DSBServer and AtlasService
 - AtlasService and UserDatabase
- A traditional OO design would disperse the tier configuration code among many classes



Architecture with Aspects

- To change between configurations have to change code in numerous classes
- Configuration aspects localize this code
 - configure connections among remote and local components
- To do this we needed to
 - hook into all the places where communication occurs
 - add tier configuration code



ParallelServer Context Code

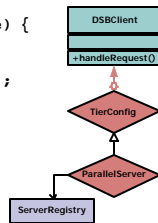
```

aspect ParallelServer extends TierConfig {

    ServerRegistry registry = new ServerRegistry();

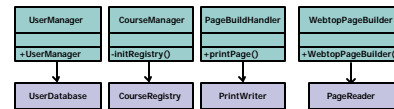
    advise * DSBClient.handleRequest(
        Servlet requestingService) {

        before {
            registry.initForParallelMode();
            handle(requestingService);
            ...
        }
        ...
    }
}
    
```



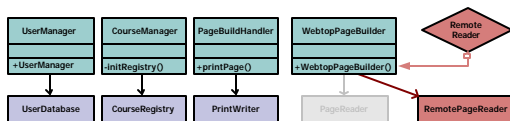
Concern 2: Distribution Support

- Some tier configurations resulted in *AtlasService* running in a distributed context
- Parts of *AtlasService* were not distribution-aware:
 - User Database: local resource
 - Course Registry: local resource
 - Accessing static web page content: read from file system
 - Printing web pages to browser: need to buffer print stream



Distribution by Reassociation

- Could have refactored *AtlasService* to use the Factory pattern to choose the appropriate distribution-aware utility
- Instead, *reassociated* the distribution-sensitive utilities by advising the corresponding methods to use distribution-aware utilities

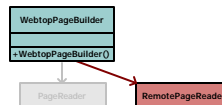


RemoteReader Aspect

- Aspect overrides member of a class with a specialized instance

```

aspect RemoteReader {
    advise WebtopPageBuilder() {
        after {
            pageReader = (RemotePageReader)
                Namespace.lookup(DSB_SERVER_IP +
                    "RemotePageReader");
        }
    }
}
    
```



Concern 3: Tracing and Debugging

- For debugging we wanted to track user requests and the construction of responses
- Challenging to implement because
 - building responses is a complex process
 - distributed system runs on numerous consoles
- Aspect can monitor every method invocation

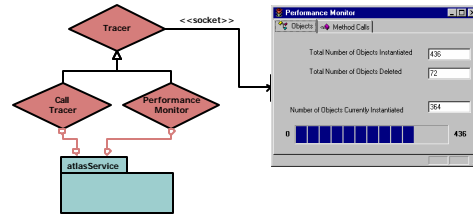
```

advise atlasService.* (*) {
  before {
    socketStream.write( /* signature + values */ );
  }
}
    
```

13

Tracing and Monitoring

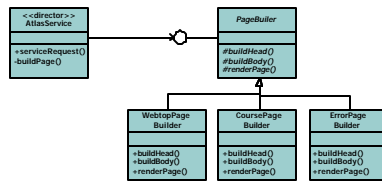
- Advising every method let us implement system-wide tracing and monitoring in as aspects



14

Concern 4: Look-and-Feel

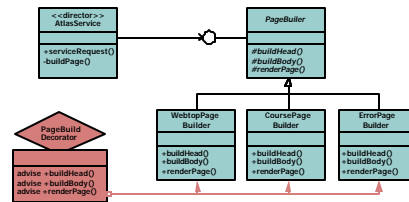
- We used the Builder pattern for constructing different kinds of web pages
- Wanted to modularize web page look-and-feel
- Did not want to refactor or spread look-and-feel code throughout build process



15

Layering in Look-and-Feel

- Could not decorate the page after it had been constructed
 - the build process did not return a data structure
- Aspects enabled layering Decorator-like functionality on top of the Builder pattern



16

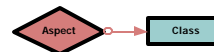
Part II: Aspect Style

- When Atlas grew to over 50 aspects and classes, it became difficult to reason about and test the code
- The system grew to 200 aspects and classes, so to preserve understandability and modifiability we constrained and stylized our aspect code by
 - Restricting the aspect-class association
 - Adopting design policies
 - Adopting implementation policies

17

Aspect-class Associations

- Restricted association link to class-directional
 - Reassociation uses class-directional association
 - Not always possible because the behavior to be tailored may not be encapsulated by an object
 - In Atlas it made it easier to understand the effects of aspects on the architecture
- We avoided having the class rely on the aspect that was providing functionality
 - This simplified the architecture by limiting coupling
 - Made it possible to test classes independently of aspects



18

Design Policies

- Limited the scope of some aspects
 - Look-and-feel concern cut across the entire system
 - It was easier not to couple web page look-and-feel to administrative interface look-and-feel
- Maintained a stand-alone object model
 - Could build an executable system from our object model
 - This enabled a faster edit-compile-debug cycle and facilitated debugging

19

Aspects and Design Patterns

- Used numerous Design Patterns in Atlas such as
 - Builder, Composite, Chain of Responsibility, Strategy
- Considered which to express as classes and which as aspects
 - Most were already well modularized without aspects
- Aspects affected our use of patterns in 2 ways
 - Reassociation avoided refactoring to use Factory pattern for adopting distributed `AtlasService` objects
 - We were able to augment the Builder pattern to accommodate an evolutionary change

20

Implementation Policies

- To ease the understandability of class code we adopted these policies
 - Advice maintains pre- post-conditions of a method
 - Before advice can not include a return statement
 - Exceptions thrown by an advice must be handled by the advice

21

Summary

- Aspects facilitated the development of Atlas
 - Modularized the tier configuration concern
 - Distributed the `AtlasService` without refactoring
 - Implemented a general tracing module
 - Layered in look-and-feel to the Builder pattern
- It was crucial to understand the aspect-class association link
 - Restricting it simplified the architecture
- Design and implementation policies improved understandability and modifiability

22