Using the Spring Framework for Aspect-Oriented Programming

Ron Bodkin
New Aspects of Software
rbodkin@newaspects.com
This Talk Is About...

- Spring Overview
- Aspect-Oriented Development
  - Concepts
  - Tools and integration
- Applications of AOP
  - Exploration & enforcement
  - Infrastructure
  - Domain-Specific
- Conclusion
  - Adoption
  - Lifecycle Use
The Crosscutting Problem

- Auxiliary concerns are scattered and tangled
  - data security
  - performance monitoring
  - business rules
  - error handling

- 80% of problems come from this 20% of code
  - inflexibility
  - redundancy
  - incomprehensibility
The AOP Solution

- Crosscutting is natural
  - can’t decompose requirements in one-dimension
- The problem is a lack of support
- Aspects provide *modular support* for crosscutting
- Evolutionary step for software development:
  - structured → objects → aspects
Benefits

- Greater product flexibility
- Reduced development costs
- Enforcement of policy...
  Reliable implementation of policy
- Fewer intractable bugs
Spring Refresher

- [http://www.springframework.org](http://www.springframework.org)
- Open Source (Apache License)
- Inversion of Control Container
  - Injects dependencies into POJOs
  - Separates environment from core logic
  - Supports testing
- Bean definitions
  - Usually XML beans file
- Additional services and wrapping layers
  - Servlet MVC, JDBC, EJB, ...
public class EmailValidator {
    private ValidationService remoteService;
    private List validDomains;

    public ValidationService getValidationService() {
        return remoteService;
    }

    public void setValidationService(ValidationService sv) {
        remoteService = sv;
    }

    ...
}
Spring Bean Definition XML

```xml
<beans>
  <bean id="emailValidator" class="EmailValidator">
    <property name="validationService" ref="validation"/>
    <property name="validDomains">
      <list>
        <value>.com</value>
        <value>.net</value>
      </list>
    </property>
  </bean>

  <bean id="validation" class="org.hitcopper.Validator"/>
</beans>
```

- Bean properties can be strings, numbers, collections, or references to other beans.
Spring Client Coding

ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

EmailValidator test = (EmailValidator) factory.getBean("emailValidator");

List domains = test.getDomains();

- ApplicationContexts look up Beans by name
  - Is normally used in a Session or Domain Factory
- Typical implementations read Spring XML configuration file for Bean Definitions
DEMO
Example: Online Music Service

- Online music streaming
- Playlists have Songs
- Both Songs and Playlists can be played by a User

Inspired by the "Implementing Observer in .NET" example at MSDN and Figures from the original AspectJ tutorial
New Requirement: Metering User Activity

- When using titles
  - Individual songs… including lyrics
  - Playing play lists
- Should track usage to allow charging user account
- Billing may vary on a number of factors
  - Monthly subscription
  - Daily pass
  - Per title
  - Promotions…
Join Points

Key points in dynamic call graph

Client

Playlist

Song

Song

Playlist.play()

execution

play()

return

play()

return

play()

return

Song.play()

execution
Pointcuts: Queries over Join Points

- This pointcut captures the method execution join points of `play()` and `showLyrics()`

```java
@Pointcut("execution(void model.Song.play()) || execution(void model.Song.showLyrics())")
void useTitle() {}
```

- Execution of `Song.play()` method
  - Name and Parameters
  - Execution of `Song.showLyrics()` method
Advice

- Code that runs before, after, or instead of a join point

```java
@Pointcut("execution(void m.Song.play()) || execution(void m.Song.showLyrics())")
void useTitle() {}

@AfterReturning("useTitle()")
public void trackTitleUse() {
    // code to run after using a title
}
```
An `Aspect` for Metering

```java
@Aspect
public class MeteringPolicy {
    @Pointcut("execution(void model.Song.play()) ||
              execution(void model.Song.showLyrics())")
    void useTitle() {}

    @AfterReturning("useTitle()")
    public void trackTitleUse() {
        MeteringService.trackUse();
    }
}
```

- An aspect is a special type
  - Like a class that crosscuts other types
  - Can contain constructs like pointcuts and advice
Metering Playables

@Aspect
public class MeteringPolicy {
    @Pointcut("execution(void modelPlayable.play()) || execution(void modelSong.showLyrics())")
    void useTitle() {}

    @AfterReturning("useTitle()")
    public void trackTitleUse() {
        MeteringService.trackUse();
    }
}

- Aspect now applies to Playlist and any other Playables (including Song)
Exposing Context

```java
@Aspect
public class MeteringPolicy {
    @Pointcut("(execution(void model.Playable.play()) || execution(void model.Song.showLyrics())) && this(playable)")
    void useTitle(Playable playable) {}

    @AfterReturning("@Pointcut()")
    public void afterUseTitle(Playable playable) {
        MeteringService.trackUse(playable);
    }
}
```

- This version exposes the currently executing object at each join point (i.e. the Playlist or Song) using this()
What Is Spring 2.0 AOP?

- Proxy-based AOP support for Java
  - Integrated with Spring Bean Container
  - Proxies generated at runtime
  - Avoids time and complexity of weaving many classes
  - Provides instance-based configuration
- Two styles for defining Aspects:
  - @AspectJ style with Java 5 Annotations
  - XML Schema-based in Spring config file
- Supported Pointcuts
  - execution, within, this, target, args,
    @within, @target, @args, @annotation
**Spring AOP Mechanisms**

- Dynamic proxy creation per instance of advised *beans*
Spring AOP Configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans.xsd
                          http://www.springframework.org/schema/aop
                          http://www.springframework.org/schema/aop/spring-aop.xsd">
  <aop:aspectj-autoproxy/> <!-- apply @AspectJ Beans -->
</beans>
```

- An empty Spring Beans config file with AOP support.
**Spring AOP Configuration**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

    <aop:aspectj-autoproxy/>

    <!-- include Aspect -->
    <bean class="model.metering.MeteringPolicy"/>

</beans>
```

- Referencing an `@AspectJ` annotated class
  - Spring will automatically proxy advised Beans
Spring AOP Configuration

```xml
<beans ...>
    <aop:aspectj-autoproxy/>
    <bean class="MeteringPolicy"/>

    <!-- Beans will be auto-proxied with Aspect -->
    <bean name="abbyRoad" class="music.model.Song">
        <property name="name" value="Abby Road"/>
    </bean>

    <bean name="playList" class="music.model.PlayList" scope="prototype">
        <property name="entries">
            <list value-type="music.modelPlayable">
                <ref bean="abbyRoad"/>
                <ref bean="rockLobster"/>
            </list>
        </property>
    </bean>
</beans>
```

Song and PlayList beans will be autoproxied by the MeteringPolicy Aspect: database later
Java Implementation

```java
class PlayList {
    private String name;
    private List<Playable> entries = new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song {
    private String name;

    public void play() {
        // play song
    }

    public void showLyrics() {
        // show lyrics
    }
}
```
Java Implementation

```java
class PlayList {
    private String name;
    private List<Playable> entries =
        new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song{
    private String name;

    public void play() {
        // play song
        MeteringService.trackUse();
    }

    public void showLyrics(){
        // show lyrics
        MeteringService.trackUse();
    }
}
```
```java
class PlayList {
    private String name;
    private List<Playable> entries = new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
        MeteringService.trackUse();
    }
}

class Song{
    private String name;

    public void play() {
        // play song
        MeteringService.trackUse();
    }

    public void showLyrics()
    {
        // show lyrics
        MeteringService.trackUse();
    }
}
```
**Java Implementation**

```java
class PlayList {
    private String name;
    private List<Playable> entries = new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
        MeteringService.trackUse(this);
    }
}

class Song{
    private String name;

    public void play() {
        // play song
        MeteringService.trackUse(this);
    }

    public void showLyrics(){
        // show lyrics
        MeteringService.trackUse(this);
    }
}
```

- Metering code scattered through domain objects
- No module captures intent and implementation of metering policy
- Evolution of metering behavior cumbersome
  - Each caller must be changed
  - Easy to introduce bugs
Spring AOP Implementation

class PlayList {
    private String name;
    private List<Playable> entries =
        new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song{
    private String name;

    public void play() {
        // play song
    }

    public void showLyrics(){
        // show lyrics
    }
}
class Playlist {
    private String name;
    private List<Playable> entries =
            new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song{
    private String name;

    public void play() {
        // play song
    }

    public void showLyrics() {
        // show lyrics
    }
}

@Aspect
class MeteringPolicy {
    @Pointcut("execution(void Song.showLyrics())
            || execution(void Song.play())")
    void useTitle() {}

    @AfterReturning("useTitle()")
    public void trackTitleUse() {
        MeteringService.trackUse();
    }
}

class PlayList {

    private String name;
    private List<Playable> entries =
            new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song{
    private String name;

    public void play() {
        // play song
    }

    public void showLyrics() {
        // show lyrics
    }
}
class PlayList {
    private String name;
    private List<Playable> entries = new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song{
    private String name;

    public void play() {
        // play song
    }

    public void showLyrics(){
        // show lyrics
    }
}
### Spring AOP Implementation

```java
class PlayList {
    private String name;
    private List<Playable> entries = new ArrayList<Playable>();

    public void play() {
        for (Playable entry : entries) {
            entry.play();
        }
    }
}

class Song {
    private String name;

    public void play() {
        // play song
    }

    public void showLyrics() {
        // show lyrics
    }
}

@Aspect
class MeteringPolicy {
    @Pointcut("(execution(void Song.showLyrics()) || execution(void Playable.play())) && this(playable)")
    void useTitle(Playable playable) {};

    @AfterReturning("useTitle(Playable playable)")
    public void trackTitleUse(Playable playable) {
        MeteringService.trackUse(playable);
    }
}
```

**Metering code centralized in MeteringPolicy**

- Intent of metering behavior is clear
- Changes to policy only affect aspect
- **Modular** evolution
Double Billing

- Don’t want to meter twice for songs played within the context of playing a Playlist
- A ThreadLocal can be used to only meter top-level advice
- Can also accomplish using AspectJ control flow pointcuts (AspectJ is discussed later)
  - cflow() and cflowbelow()
Avoiding Double Billing

```java
ThreadLocal<Integer> callDepth = ...;

@Around("useTitle(playable)")
public void aroundUseTitle(ProceedingJoinPoint jp, Playable playable) {
    try {
        callDepth.set(callDepth.get() + 1); // increment counter
        jp.proceed(); // continue joinpoint
        if (callDepth.get() == 1) { // if first call
            MeteringService.trackUse(playable);
        }
    } finally {
        callDepth.set(callDepth.get() - 1); // decrement counter
    }
}
```
Configuring Spring Aspects

- Aspects can reference externally defined services

```java
@Aspect
public class MeteringPolicy {
    private AccountManager accountManager;
    private MeteringService meteringService;

    public void setAccountManager(AccountManager actManager) {
        this.accountManager = actManager;
    }

    public void setMeteringService(MeteringService mtrSvc) {
        this.meteringService = mtrSvc;
    }

    ...
}
```
Configuring Spring Aspects

@AspectJ Aspects can be directly configured in Spring standard <property> declarations

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <aop:aspectj-autoproxy/>

    <bean class="model.metering.MeteringPolicy">
        <property name="accountManager" bean="ldapActManager"/>
        <property name="meteringService" bean="titleMetering"/>
    </bean>

</beans>
```
Schema style Aspects

- Spring Aspects can be defined with XML declarations
  - instead of @AspectJ annotated classes
- Useful when Java 5 Annotations aren't available, e.g., Java 1.4
- Less capable than @AspectJ style
  - Can't combine named pointcuts
  - Only singleton lifecycle supported
The MeteringPolicy class is coded without Annotations

- No Pointcuts are defined
- Advice is implemented, but not declared
The MeteringPolicy Bean is declared with a standard `<bean>` declaration.

An `<aop:aspect>` is defined referring to the MeteringPolicy Bean.
An `<aop:pointcut>` is defined as `useTitle`

- Note: "and", "or", and "not" are substituted in XML syntax to avoid char escaping issues
Spring Configuration

```xml
<aop:config>
  <aop:aspect id="meteringPolicyAspect" ref="meteringPolicy">
    <aop:pointcut id="useTitle" expression="execution.."/>
    <aop:after-returning pointcut-ref="useTitle" method="trackTitleUse"/>
  </aop:aspect>
</aop:config>

<bean id="meteringPolicy" class="metering.MeteringPolicy"/>
```

- `<aop:after-returning>` advice is declared
  - Referencing the `useTitle` pointcut
  - and `MeteringPolicy.afterUseTitle()` method
Creating Playables from DAOs

```java
public class MusicService {
    public void play(String title) {
        Playable playable = dao.find(user, title);
        playable.play();
    }

    public void setDao(PlayableDao dao) {
        this.dao = dao;
    }

    private PlayableDao dao;
    private User user;
}
```
Configuring Database Access

```xml
<beans ...

...<bean id="playableDao" class="music.dao.PlayableDaoImpl">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
   <property name="dataSource" ref="musicDB"/>
   <property name="mappingResources">
      <list>
         <value>music.hbm.xml</value>
      </list>
   </property>
</bean>

..."
@Aspect  public class MeteringPolicy {
... <as before>

    @Pointcut("execution(music.model.Playable
             music.model.PlayableDao.find*(..))")
    void createPlayable() {
    }

    @Around("createPlayable()")
    public Playable proxyPlayable(ProceedingJoinPoint pjp)
    throws Throwable {
        Playable created = (Playable) pjp.proceed();
        return proxyPlayable(created);
    }

    ...

private Playable proxyPlayable(Playable playable) {
    AspectJProxyFactory factory = new AspectJProxyFactory(playable);
    factory.addAspect(this);
    if (playable instanceof PlayList)
        proxyChildren((PlayList)playable);
    return (Playable)factory.getProxy();
}

private void proxyChildren(PlayList playList) {
    for (ListIterator<Playable> it = playList
            .getEntries().listIterator(); it.hasNext()) {
        Playable child = it.next();
        Playable proxy = proxyPlayable(child);
        it.remove();
        it.add(proxy);
    }
}
Part II: Sample Applications

- Exploration & Enforcement
- Transactions
- Error Handling
Enforcing State Transitions

```java
@Aspect
public class StateTracker {
    @Pointcut("execution(* close()) && this(resource)")
    public void closingResource(Resource resource);

    @Before("closingResource(resource)"
    public void callingModelFromDataAccess(Resource resource) {
        if (resource.hasPendingRequests()) {
            throw new InvalidCallException(resource);
        }
    }

    ...
}
```
Spring Transactions

- Spring provides declarative transaction management with
  - XML Declarations, or
  - @Transactional Annotation
- Relies on a PlatformTransactionManager
  - Implementations for JDBC, JTA, Hibernate, ...
- Programmatic Transaction Management can be implemented with AspectJ Aspects
AOP Transactions Setup

- `<tx:advice id="tx-advice" transaction-manager="txManager">` binds to a Transaction Manager
  
  *Legacy concept*

- `<tx:attributes>` configures the Advice

- Here the "txManager" is a Jdbc Manager

```xml
<tx:advice id="tx-advice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>

<bean id="txManager" class=".jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```
### Apply AOP Transactions

```xml
<tx:advice id="tx-advice" transaction-manager="txManager">
    <aop:config>
        <aop:advisor>
            pointcut="execution(public * service.Service+.*(..))"
            advice-ref="tx-advice" />
        </aop:advisor>
    </aop:config>
</tx:advice>
```

- An `<aop:advisor>` binds the Service Layer pointcut to the Transactional Advice
  
  *Legacy concept: aspect with one advice*

- This declares the entire Service Layer to be Transactional
The `@Transactional` annotation marks types as requiring Transaction semantics.

Optional properties for `@Transactional`:

- propagation, isolation, readonly, rollbackFor, rollbackForClassname, noRollbackFor, noRollbackForClassname
@Transactional Config

```xml
<bean name="orderService" class="service.ServiceImpl">
  <property ... />
</bean>

<tx:annotation-driven/>

<bean id="transactionManager"
     class="..jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

- The `<tx:annotation-driven>` element enables the `@Transactional` processing of beans
- A TransactionManager is still needed
  - Default id is "transactionManager"
Annotations and Pointcuts

- Annotations: a little goes a long way
  - Useful to pick out key characteristics (business operation, immutable, etc.)
  - Brittle if scattered and tangled macro invocations: not an improvement over inline code calls

- AOP supports robust, maintainable, testable use
  - Structural pointcuts (the best)
  - Can derive implementation from core domain abstractions
  - Can annotate exceptions
  - Shows the forest and the trees
Error Handling

```java
@Aspect
public class ModelErrorHandling {
    @Pointcut("execution(public * model..*(..))")
    void model() {}

    @AfterThrowing("model()", throwing="e")
    public void modelErrors(JoinPoint jp, Throwable e) {
        if (!(e instanceof ModelException)) {
            ModelException me = new ModelException(e);
            me.setModel(jp.getTarget());
            me.setArgs(jp.getArgs());
            throw me;
        }
    }
}
```
Spring Security

Source: [http://www.acegisecurity.org](http://www.acegisecurity.org)
Part III: AspectJ Integration

- Fine-Grained Configuration
- Other Examples
Some objects are created outside of the bean container, e.g., persistent objects created by Hibernate.

```java
public class Account {
    private TaxCalculator taxCalculator;
    ...

    public void update() {
        taxOwed += taxCalculator.computeTax();
        ...
    }
    ...
}
```
Enter AspectJ …

- The original AOP implementation for Java
  - Language extension, @AspectJ, and XML options
  - Java platform compatible
  - Performance comparable to hand-written equivalent

- Tool support
  - Compiler, linker, classloader-based weaving
  - IDE support: Eclipse, JBuilder, JDeveloper, NetBeans
  - Ant, Maven, ajdoc, Java debugger

- Open source: http://eclipse.org/aspectj
Configure by *Annotation*...

- Enable load-time weaving: add JVM arg
  ```
  -javaagent:lib/aspectjweaver.jar
  ```

```
@Configurable
public class Account { … }
```

- Spring will automatically configure

```
<beans..>
  <bean class="com.example.app.domain.Account"
    scope="prototype">
    <property name="taxCalculator" ref="taxCalculator" />
  </bean>
...</beans>
```
Configure by *Pointcut*

```java
@Aspect
class ConfigureDomain extends AbstractBeanConfigurerAspect {
    @Pointcut("execution(new(..)) && this(instance) && within(com.example.app.domain..*)")
    public void beanCreation(Object instance) {}
}
```

- Also uses AspectJ weaving
  - Load-time (or *build-time*...)
- Annotation vs. pointcut trade-offs
  - Just as with Spring AOP
Configure with Declaration

```java
aspect ConfigureDomain {
    declare @type: com.example.app.domain..*: @Configurable;
}
```

- Uses AspectJ language syntax
- Adds annotations to all types based on a type pattern
- Also useful for other annotation-driven APIs
  - EJB3 persistence, JAX-WS, ...
AspectJ Advising *Playables*

- In our earlier example, we had to manually proxy persistent objects returned from Hibernate.
- If we are using AspectJ, it will advise instances of Playable and Song when created.
  - Since these are our domain types it will work with build-time or load-time weaving.
Architectural Layers

@Aspect
public abstract class Architecture { // abstract aspect for reuse

    @Pointcut("within(junit.framework.TestCase+) ||
              within(*..test..*)")
    public void inTest() {}

    @Pointcut("!inTest()")
    public void scope() {}

    @Pointcut("call(public * com.example.app.model..*(..))")
    public void modelCall() {}

    @Pointcut("within(com.example.app.persistence..*)")
    public void inDataAccess() {}

    @Pointcut("call(public * org.hibernate..*(..))")
    public void persistenceCall() {}
}

Enforcing Layering at Weave Time

```java
@Aspect
class Layering extends Architecture {
    @DeclareError("scope() && modelCall() && inDataAccess()")
    static final String callingModelFromDataAccess =
            "Don't call the model from the data access tier directly";

    @DeclareError("scope() && persistenceCall() && !inDataAccess()")
    static final String callingPersistenceNotFromDataAccess =
            "Don't call the persistence tier from outside data access";
}
```
Glassbox Open Source

- **Non-invasive data capture**
  - Captures data with AOP, Java 5 and server JMX data
  - Exposes detailed data through JMX consoles

- **Automated analysis: Glassbox Troubleshooter**
  - Automatically diagnoses common problems
  - Correlates, compares, analyzes data from data capture & summary
  - Exposed through an AJAX Web client

- **Focus on the 80% of common problems**
  - Database issues (connections, slow query, death by 1000 cuts)
  - Remote service calls (failures, chattiness, slow response)
  - Java contention, failures

- **Open Source LGPL License**
- **Supports Java 1.4 and later**
Glassbox Architecture

- Load-time weaving
- Discovers and tracks high-level operations as they execute
- Efficiently detects common problems, e.g.
  - Slow queries
  - Excessive service calls
  - Connection failures
  - Java bottlenecks
This illustrates AspectJ XML-defined aspects

Defined inside a load-time weaving configuration file: META-INF/aop.xml
@Aspect
public class PlayableMonitor extends AbstractMonitor {
    @Pointcut("music.metering.MeteringPolicy.useTitle(key)")
    public void monitorPoint(Object key) {
    }
}
public class PlayableMonitor {
    private ResponseFactory responseFactory = AbstractMonitor.getResponseFactory();

    public void setResponseFactory(ResponseFactory factory) {
        this.responseFactory = factory;
    }

    public ResponseFactory getResponseFactory() {
        return factory;
    }

    public void beforeUseTitle(Playable playable) {
        Response response = responseFactory.getResponse(playable.getName());
        response.setLayer("streaming");
        response.start();
    }
}
... public void afterExceptionTitle(Playable playable) {
    Response response = responseFactory.getLastResponse();
    FailureDescription =
        fdFactory.getFailureDescription(t);
    response.set(Response.FAILURE_DATA, description);
    response.complete();
}

public void afterUseTitle(Playable playable) {
    responseFactory.getLastResponse().complete();
}
}
Schema AOP with Glassbox

```xml
<aop:config>
  <aop:pointcut id="useTitle" expression="execution(..)"/>
  <aop:aspect id="playableMonitorAspect" ref="playableMonitor">
    <aop:before pointcut-ref="useTitle" method="beforeUseTitle"/>
    <aop:after-throwing pointcut-ref="useTitle" method="afterExceptionUseTitle"/>
    <aop:after-returning pointcut-ref="useTitle" method="afterUseTitle"/>
  </aop:aspect>
  <aop:aspect id="meteringPolicyAspect"/>
</aop:config>

<bean id="playableMonitor" class="music.PlayableMonitor"/>
```
Part III: Conclusion …

• The state of AOP
• Adoption strategy
The State of AOP

- **AspectJ 5:**
  - The pure play
  - Powerful, complete, more to learn
  - Tools support

- **Spring 2.0**
  - Built-in AOP + AspectJ integration

- **JBoss AOP**
  - Integrated aspects:
    - Core to EJB3 implementation, POJ O Cache, …

- **Emerging for .NET, PHP, Ruby …**
  - *e.g.*, Spring.NET aspects
Aspect-Oriented Development

- Good OOD & good AOD work together
- Use UML extension with stereotypes
- Analysis aspects from crosscutting concepts
- **ARC**: Aspect, Responsibilities, Collaborators
- Pointcut design should use stable properties
- Aspects let you unit test & integration test crosscutting requirements
Don't Run before You Can Walk

Risky Space

reward vs. time & confidence
Phases of Adoption

- **exploration**
- **enforcement**
- **auxiliary / infrastructure**
- **core / business**

 Axes:
- **time & confidence**
- **reward**

Legend:
- Green: exploration
- Light Green: enforcement
- Yellow: auxiliary / infrastructure
- Red: core / business
Conclusion

- Spring and AOP are rapidly gaining adoption in the enterprise

- Incremental adoption works best
  - Coarse-grained Spring aspects
  - Fine-grained AspectJ aspects

- Training, consulting, and support available
Thank You

Ron Bodkin  rbodkin@newaspects.com

New Aspects of Software

Thanks to John Heintz for creating this presentation with me. Some slides in this presentation were created by or in collaboration with Gregor Kiczales, IBM, Nicholas Lesiecki, Ramnivas Laddad, and Dion Almaer, and are used with permission. The copyright notice on this presentation refers to the slides created by New Aspects of Software and to this assembly into a whole. Thanks to all!