ENTERPRISE DEVELOPMENT SERVICES DIRECT FROM THE CREATORS OF SPRING FRAMEWORK

INTERFACE21

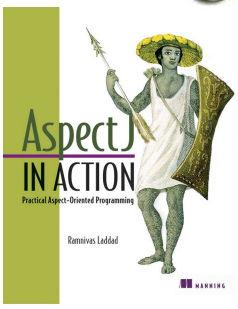
Advances in Aspect-oriented Programming

Ramnivas Laddad
Principal, Interface21
Author, AspectJ in Action
ramnivas.laddad@interface21.com

Spring

INTERFACE21

About Speaker



- Principal at Interface21
  - Specializing in aspect-oriented programming and enterprise Java
- Author of books and articles
  - AspectJ in Action: Practical Aspect-Oriented Programming
  - Several articles on IBM developerWorks, TheServerSide, JavaWorld and other publications
- Speaker at many professional conferences
  - No Fluff Just Stuff, JavaOne, JavaPolis, SpringOne, Software Development, EclipseCon, O'Reilly OSCON etc.
- Active involvement in AspectJ and Spring

Spring

INTERFACE21

The state of AOP

- The power of AOP is clear
  - Writing complex system without AOP in maintainable, understandable manner is impossible
- AOP is still an emerging technology
  - Only 10 years
- AspectJ programming model is the de-facto standard
  - Multiple syntax and implementations
- Adoption is steadily increasing
  - Especially with Spring 2.0

Spring

INTERFACE21

Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

Spring

INTERFACE21

Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

Spring

INTERFACE21

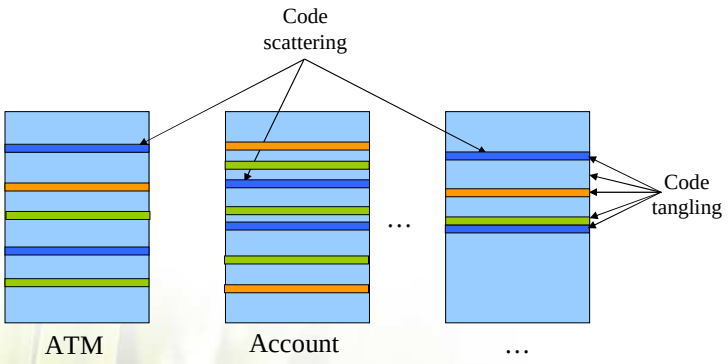
A quick intro to AOP

- Crosscutting concerns
  - Functionality whose implementation spans multiple modules
  - Many examples: Logging and tracing, Transaction management, security, caching, error handling, business rules, performance monitoring…
- AOP
  - A programming methodology to help with crosscutting concerns

Spring

INTERFACE21

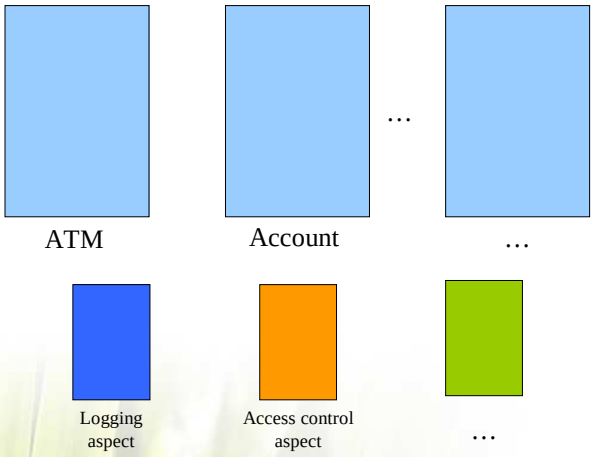
System Evolution: Conventional



Spring

INTERFACE21

System Evolution: AOP based



Spring

INTERFACE21

Core AOP concepts

- Join point
  - An identifiable point in the execution of a program.
  - Central, distinguishing concept in AOP
- Pointcut
  - Program construct that selects join points and collects context at those points.
- Advice
  - Code to be executed at a join point that has been selected by a pointcut
- Inter-type declarations
  - Additional data or method to existing types, implementing new interfaces, attaching new annotations

Spring

INTERFACE21

Pointcut

- Selects join points and collects context at those points
- Example:

```
pointcut release(Connection conn)
  : call(void Connection.close()) && target(conn);

pointcut accountCreation(Account account)
  : execution(Account.new()) && this(account);

pointcut service()
   : execution(* Remote+.*(..) throws RemoteException);

pointcut accountChangeFromATM(ATM atm, Account account)
  : cflow(atmRequest(atm)) && accountChange(account);
```


Spring

INTERFACE21

Dynamic crosscutting: Advice

- Code to be executed at a join point that has been selected by a pointcut
- Three kinds:
  - Before

```
before() : call(* ATM.*(..)) {
    ...
}
```

  - After

```
after(Account account) returning : accountOperation(account) {
    ...
}
```

  - Around

```
Object around() : call(* Remote+.*(..) throws RemoteException) {
    try {
        ... proceed(); ... Retry logic ...
    } catch (...) {
    }
}
```


Spring

INTERFACE21

A Logging Aspect

```
public aspect BankLoggingAspect {
  private static Logger _logger
    = Logger.getLogger("banking");

  public pointcut loggedOperations()
    : execution(* banking..*.*(..))
      && !within(BankLoggingAspect);

  before() : loggedOperations() {
    Signature sig =
        thisJoinPointStaticPart.getSignature();
    _logger.logp(Level.INFO,
        sig.getDeclaringType().getName(),
        sig.getName(), "Entering");
  }
}
```


Spring

Static crosscutting

- Introduce new parent types

```
declare parents: banking..* implements Loggable;
```

- Introduce new members

```
public Logger Loggable._logger;
```

- Soften exceptions

```
declare soft: SQLException
            : within(banking.model..*);
```

- Compile-time errors and warning

```
declare error
    : call(* Thread.*(..))
      && within(javax.ejb.EnterpriseBean+)
    : "Use of thread disallowed from EJBs";
```


Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

INTERFACE21

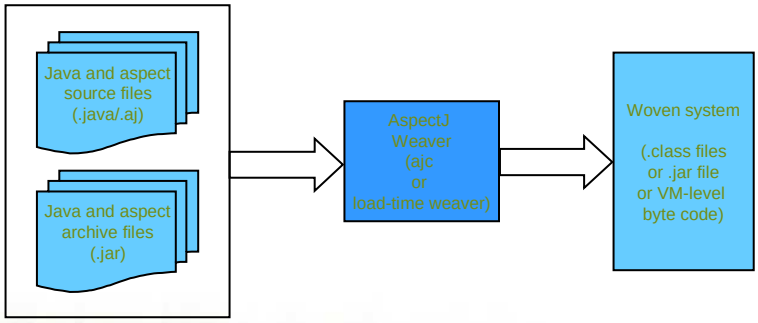
Understanding various AOP systems

- AspectJ AOP
  - Byte-code modification based
  - Provides most powerful model
- Spring AOP is proxy-based
  - Best suited for applying crosscutting concerns to services, DAOs etc.
  - Zero logistical changes to get you started

Spring

INTERFACE21

AspectJ AOP schematic

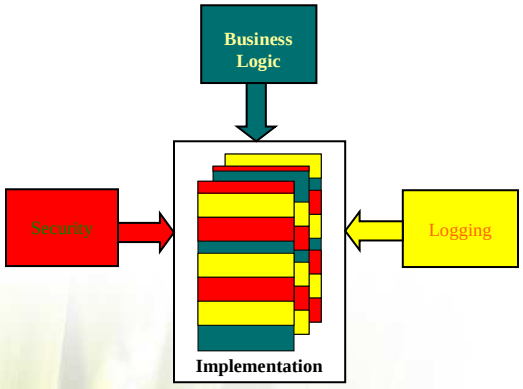


Other possibilities: binary and load-time weaving

Spring

INTERFACE21

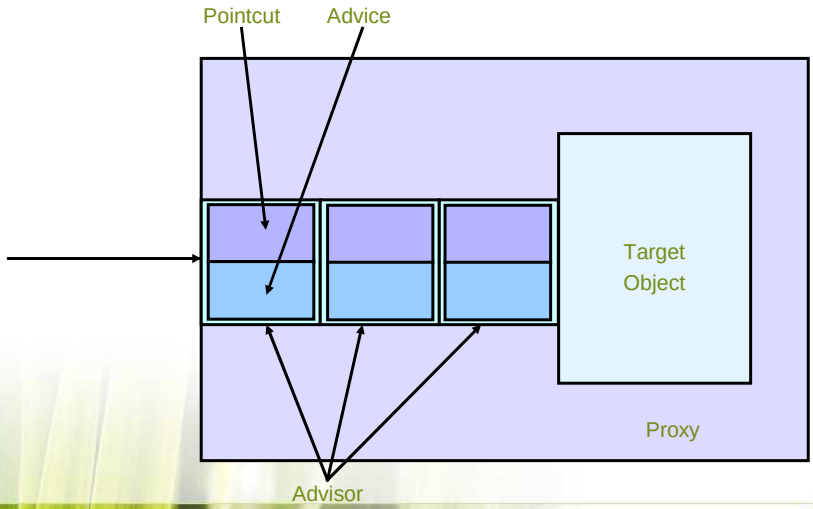
AspectJ implementation model



Spring

INTERFACE21

Spring AOP implementation model



Spring

INTERFACE21

Advantages of proxy-based approach

- Requires no special compiler
- Allow per-object interceptors
  - Not per-class
- Less full-fledged
  - Exposes only method-execution join points
  - Allows easing into AOP
- Easy to modify applicable interceptors dynamically

Spring

INTERFACE21

Limitations of proxy-based approach

- Method-only interception
  - No field-access or object creation
- Explicit creation of proxy required
  - Can't use 'new' to create objects
  - Auto-proxy mechanism helps with this
- Calls to 'self' don't go through interceptors

Spring

INTERFACE21

Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

Spring

INTERFACE21

New @AspectJ syntax

```
@Aspect
public class LoggingAspect {
  private static Logger _logger = Logger.getLogger(“tracer");

  @Pointcut(“execution(* *.* (..))”)
  public void loggedOperations() {}

  @Before(“loggedOperations()”)
  public void log(JoinPoint.StaticPart
   thisJoinPointStaticPart) {
     Signature sig = thisJoinPointStaticPart.getSignature();
     _logger.logp(Level.INFO, sig.getDeclaringType().getName(),
                  sig.getName(), "Entering");
  }
}
```


Spring

XML syntax

- Base aspect expressed in traditional or @AspectJ syntax

```
public abstract aspect MonitoringAspect {
   public abstract pointcut monitoredOps();

   Object around() : monitoredOps() {
      long startTime = System.nanoTime();
      Object ret = proceed();
      long endTime = System.nanoTime();

      monitorAgent.record(thisJoinPointStaticPart,
                          endTime-startTime);
      return ret;
   }
   ...
}
```


XML syntax: Defining concrete aspect

```
<aspectj>
  <aspects>
     <concrete-aspect
          name="com.aspectivity.monitoring.JDBCMonitoringAspect"
          extends="com.aspectivity.monitoring.MonitoringAspect">
        <pointcut name="monitoredOps"
                  expression="call(* java.sql.*.*(..))"/>
     </concrete-aspect>

     ...

  </aspects>

  ...

</aspectj>
```

INTERFACE21

Load-time weaving

- Weave while classes are being loaded
  - No explicit compile-time or binary weaving needed
- Agents supported
  - JVMTI-based
    - Application-server agnostic
    - VMs supported
      - All 5.0+ VMs

      java -javaagent:aspectjweaver.jar …
      - JRockit VM 1.3/1.4

      java -Xmanagement:class=org.aspectj.weaver.loadtime.JRockitAgent …
  - Classloader-based
    - Application-server specific
      - Not available for all app servers

Spring

INTERFACE21

LTW deployment: aop.xml

```
<aspectj>
  <aspects>
    <aspect name="com.aspectivity.security.FineGrainedSecurity"/>
    <aspect name="com.aspectivity.tracing.ExceptionTracer"/>

    <concrete-aspect
          name="com.aspectivity.monitoring.JDBCMonitoringAspect"
          extends="com.aspectivity.monitoring.MonitoringAspect">
        <pointcut name="monitoredOps"
                  expression="call(java.sql.*.*(..))"/>
     </concrete-aspect>
  </aspects>

  <weaver>
    <include within="com.interface21..*"/>
    <exclude within="com.interface21.rss..*/>
  </weaver>
</aspectj>
```


Spring

INTERFACE21

Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

Spring

INTERFACE21

Spring-AspectJ integration

- Provides power of AspectJ in Spring
- Core message
  - Spring AOP is good enough in many cases
  - But, you often need more power
- Multiple level of integration
  - Something for every one

Spring

INTERFACE21

Spring-AspectJ: Levels of integration

- No special compiler
  - AspectJ pointcut expression
    - With proxy mechanism
  - Typed advice
    - XML-based aspects definition with plain Java classes
    - @AspectJ aspects
- Special compiler or weaver
  - Full power AspectJ

Spring

INTERFACE21

AspectJ pointcut expression

```
<bean id="spamPreventor"
      class="o.s.aop.support.DefaultPointcutAdvisor">
   <property name="advice">
      <bean class="example.SpamPreventionInterceptor"/>
   </property>
   <property name="pointcut">
      <bean class="o.s...AspectJExpressionPointcut">
         <property name="expression"
                   value="execution(* send(..))"/>
      </bean>
   </property>
</bean>
```


Spring

Plain Java aspect

```
package example;

public class EmailLogger {
  public void log(JoinPoint.StaticPart tjpsp,
                  String address) {
      System.err.println(
         "Invoking " + tjpsp.getSignature() +
         " for " + address);
   }
}
```

```
void before(Method method, Object[] args,
            Object target) throws Throwable;
```


Typed advice: Defining aspect

```
<beans ...>
<aop:config>
   <aop:aspect id="emailLoggerBean"
               ref="emailerLogger">
      <aop:before
         pointcut="execution(* send(String, String))
                   and args(address, *)"
         method="log"/>
   </aop:aspect>
</aop:config>

<bean id="emailerLogger"
      class="example.EmailLogger"/>
...
</beans>
```

Do away with XML: @AspectJ

```
package example;

@Aspect
public class EmailLogger {
  @Before(execution(* send(String, String))
                    && args(address, *)")
  public void log(JoinPoint.StaticPart tjpsp,
                  String address) {
      System.err.println(
          "Invoking " + tjpsp.getSignature() +
          " for " + address);
   }
}
```


Using @AspectJ aspect

```
<beans ...>

  <aop:aspectj-autoproxy/>

  <bean id="emailerLogger"
        class="example.EmailLogger"/>

  ... Other beans ...

</beans>
```

INTERFACE21

Choosing Spring vs. AspectJ AOP

- Use Spring AOP when
  - Method-only interception is sufficient
    - Full power of AOP overwhelming
  - Don't want to use special compiler
  - Domain object's don't need to be crosscutted
  - Pre-written aspects meet your needs
- Use AspectJ AOP when
  - Otherwise…

Spring

INTERFACE21

Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

Spring

INTERFACE21

Layering violation enforcement

- Problem
  - UI layer often directly use DAO layer contrary to architectural policies
  - No automatic enforcement
- Solution
  - One simple, reusable aspect to enforce the policies
    - Just add to your build system

Spring

INTERFACE21

Layering policy enforcement aspect

```
public aspect LayeringPolicyEnforcementAspect {
  pointcut dataAccess()
    : call(* com.mycompany.dao..*.*(..))
      || call(* org.hibernate..*.*(..))
      || call(* *..sql..*.*(..));

  declare error : dataAccess()
                  && within(com.mycompany.ui..*)
    : "UI layer shouldn't directly access data;
       Use business layer instead";
}
```


Spring

INTERFACE21

First failure data capture (FFDC)

- Problem:
  - Exception at low level trickles through layers; however, original context is lost
- Solution:
  - FFDC aspect logging the exception at the first failure site

Spring

INTERFACE21

FFDC: Base aspect

```
public abstract aspect FFDC {
    public abstract pointcut ffdcOp();

    after() throwing(Exception ex) : ffdcOp() {
        if(_logger.isLoggable(Level.WARNING)) {
            Signature sig = thisJoinPointStaticPart.getSignature();
            logFFDC(sig.getDeclaringType().getName(),
                    sig.getName(), ex,
                    thisJoinPoint.getThis(),
                    thisJoinPoint.getArgs());
        }
    }

    public void logFFDC(String className, String methodName,
                        Exception ex, Object thiz, Object[] args) {
        ...
    }
}
```


Spring

FFDC: Derived aspect

```
public aspect BlogFFDC extends FFDC {
  public pointcut ffdcOp()
    : execution(* com.mycompany..*.*(..)
                                  throws RemoteException)
      || execution(* com.mycompany..dao.*.*(..));
}
```


Mix-in implementation

- Problem:
  - Boilerplate implementation of an implementation
- Solution:
  - Provide the default implementation using static crosscutting

INTERFACE21

Interface needing boilerplate code

```
public interface ServiceCenter {
    public String getId();
    public void setId(String id);

    public String getAddress();
    public void setAddress(String address);
}
```


Spring

INTERFACE21

Interface with mixin implementation

```
public interface ServiceCenterMixin extends ServiceCenter {
    static aspect IMPL {
        private String ServiceCenterMixin.id;
        private String ServiceCenterMixin.address;

        public String ServiceCenterMixin.getId() {
            return id;
        }

        public void ServiceCenterMixin.setId(String id) {
            this.id = id;
        }

        public String ServiceCenterMixin.getAddress() {
            return address;
        }

        public void ServiceCenterMixin.setAddress(String address) {
            this.address = address;
        }
    }
}
```


Spring

Interface implementations

```
public class ATM extends Teller
    implements Ser
    ... No set/get
    ... No id, add
}


public class Brick
    implements Ser
    ... No set/getId(), set/getAddress()
    ... No id, address
}
```

Try it yourself:

Mixin for *Aware interfaces

- BeanNameAware
- ApplicationContextAware
- …

Example: Sanitizing input data

```
@Aspect
public class SanitizeUntrustedData {
  @Pointcut("execution(* service..*(..))
            && args(untrusted, ..)")
  public void untrustedDataUsage(
                      Untrusted untrusted) {
  }

  @Before("untrustedDataUsage(untrusted)")
  public void sanitize(Untrusted untrusted) {
    ... Santitize data
  }
}
```

INTERFACE21

Annotation-driven Crosscutting

- Combines
  - Power of AOP
  - Ease of annotations
- Offers AOP with ease
  - Core developers supply annotations
  - Aspect developers consume annotations
  - ➔ A lot smoother learning curve

Spring

INTERFACE21

Transaction management

- Issue:
  - How to designate methods
  - How to specify transaction attributes
- Solution:
  - Annotations to designate methods
  - Annotation properties to specify transaction attributes

Spring

INTERFACE21

Making a service transactional

```
@Transactional
class DefaultAccountService implements AccountService {

    public void debit(Account account, float amount) {
        //
    }

    @Transactional(readOnly=true)
    public float getBalance(Account account) {
        //
    }

    ...
}
```


Spring

INTERFACE21

Transaction management: XML configuration

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

  <bean id="accountService"
        class="service.DefaultAccountService" />

  <bean id="txManager"
     class="o.s...DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <tx:annotation-driven
                  transaction-manager="txManager"/>

</beans>
```


Spring

INTERFACE21

Data-driven access control

- Utilize pointcut expression language to select methods based on
  - Annotations to defining type
  - Annotations to methods
  - Annotations to return types
  - Annotations to arguments

Spring

INTERFACE21

Data-driven access control

```
@Sensitive
public class MedicalRecord {
  ... details
}
```

Copyright 2004-2007, Interface21 Ltd. Copying, publishing, or distributing without expressed written permission is prohibited.

Spring

Monitoring Sensitive Information Access

```
@Aspect
class SensitiveAccessMonitoring {
    @Pointcut("execution((@Sensitive *) *(..))")
    public void sensitiveAccess() {}

    @AfterReturning(value="sensitiveAccess()",
                    returning="sensitiveObj")
    public void monitor(Object sensitiveObj) {
        ... Access control logic
    }
}
```


Injecting dependencies using Aspect

- Limitations of Spring's "traditional" DI
  - Injection limited to beans created through configuration
  - → Not sufficient for objects created thru other mechanisms: Hibernate, JDO, fine grained objects
  - → Prevents richer domain models prescribed by domain-driven design (DDD)
- Solution: Use aspects to inject dependencies

INTERFACE21

Dependency declaration: Based on template id

```
@Configurable("emailerClient")
public class Order {
   private Emailer emailer;

   public void process() {
      ...
      emailer.send(...)
   }

   public void setEmailer(Emailer emailer) {
      this.emailer = emailer;
   }
}
```


Spring

INTERFACE21

Dependency configuration: Based on template id

```
<beans...>
  <bean id="smtpEmailer" .../>

  <bean id="emailerClient" scope="prototype">
      <property name="emailer" ref="smtpEmailer"/>
  </bean>

  <aop:spring-configured/>

</beans>
```


Spring

Dependency declaration: Based on template class

```
@Configurable
public class Order {
   private Emailer emailer;

   public void process() {
      ...
      emailer.send(...)
   }

   public void setEmailer(Emailer emailer) {
      this.emailer = emailer;
   }
}
```


Dependency configuration: Based on template class

```
<beans...>
  <bean id="smtpEmailer" .../>

  <bean class="example.Order" scope="prototype">
      <property name="emailer" ref="smtpEmailer"/>
  </bean>

  <aop:spring-configured/>

</beans>
```

INTERFACE21

Agenda

- A quick introduction to AOP
- Various AOP implementation models
- Advances in AspectJ
- Advances in Spring/AOP
- Advances in usage model
- The future

Spring

INTERFACE21

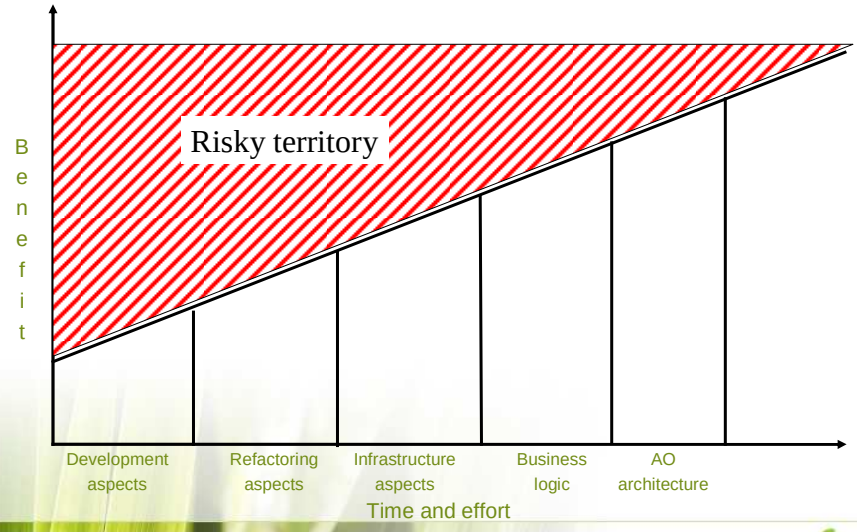
What's on horizon?

- Improvements in load-time weaver
- Spring-native load-time weaver
- Improvements in tools
- AOSGi
  - Early version already available
- VM-level AOP
  - Some prototypes available
  - High-value, high-effort project

Spring

INTERFACE21

AOP adoption curve



Spring

INTERFACE21

Pragmatic adoption

- Use Spring/AOP when pre-written aspects meet your needs
- Use Spring AOP with AspectJ to start tapping into AspectJ's power
- Use AspectJ incrementally
  - Development aspects
  - Refactoring aspects
  - Production aspects

Spring

INTERFACE21

Summary

- AOP is advancing at pragmatic pace
  - Not too fast ➔ Stable core model
  - Not too slow ➔ Responsive implementation model
- AspectJ LTW provides easy integration possibilities
- Spring/AspectJ integration offers full power of AOP in the Spring environment
  - De-facto for new Spring projects

The fun has just began…

Spring

INTERFACE21

Spring and AOP trainings coming to Philadelphia

Core Spring (May 1-4)

Core AOP (May 22-25)

(Discount code: emergingtech)

http://interface21.com/training

Spring

INTERFACE21

Questions?

Ramnivas Laddad
Email: ramnivas.laddad@interface21.com
Blog: http://ramnivas.com/blog

Spring