

INTRODUCTION TO ASPECT ORIENTED PROGRAMMING

Life is found in layers

Single Responsibility Principle

- Every class should have responsibility over a single piece of functionality provided.
- That responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.
- ***A class should have only one reason to change.***

The General AOP Use Case: Scattering & Tangling

- A functional implementation is **scattered** if its code is spread out over multiple modules. Its implementation is not modular. A small change in functionality needs to be made in multiple modules.

Logging is “scattered” throughout an application

- **Example:**

Adding/removing/modifying a logging message that spans ALL service methods

See AOP Demo

- A functional implementation is **tangled** if its code is intermixed with code that implements other functionality. The module in which tangling occurs is not cohesive.

[Programmatic] Transaction Management is “tangled” within a method

See Lesson 2 HibernateSolo Demo

Aspect-oriented approach identifies code scattering and tangling as the indicators of crosscutting concerns.

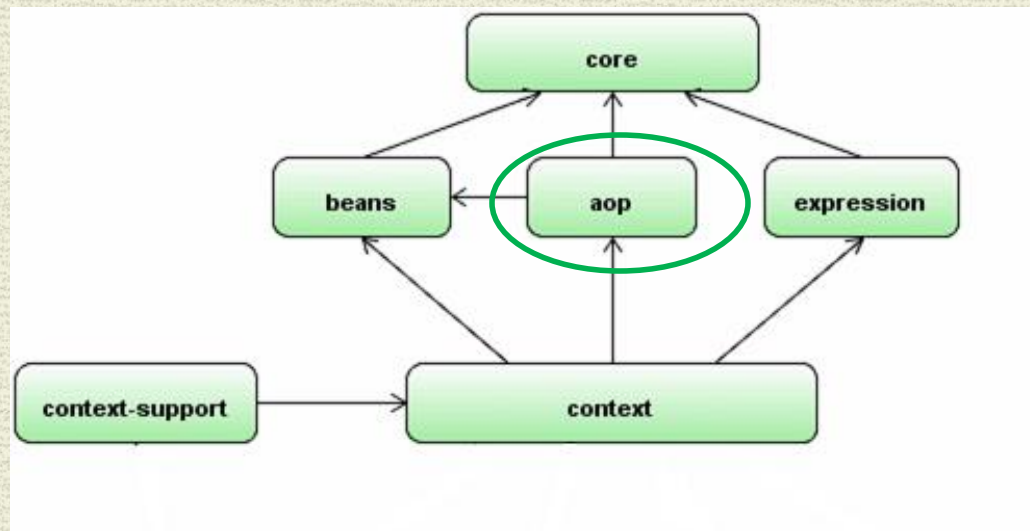
AOP Value Added

- Separation of Concerns
- Increased Modularity
- Reduces “spaghetti” code
- Code reduction
- Removes “hard” dependencies
- **Some USE CASES:**
 - Boilerplate/repetitive code** – unable to be refactored using normal OO techniques
 - Transaction**
 - Security**
 - Logging**
 - Authorization**
 - Validation**

AOP - a Spring Core Technology

- One of the key components of Spring is the *AOP framework*. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.
- The Spring Framework uses Spring AOP internally for transaction management, security, remote access, and Cache Abstraction.

Spring Core & Dependencies



AOP Definitions

Cross-cutting Concern

Another name for an **Aspect**.

An Aspect “crosscuts” core functionality

Aspect

Functionality fundamental to application

BUT not the primary business function.

Aspect is to AOP as Class is to OOP.

AOP Definitions [Cont.]

- **Aspect** - implemented by applying **Advice** (additional behavior) at various **Join points** (methods in Spring application) specified by a **Pointcut** (criteria to match **Join points** to **Advice**).
- **Advice**
 - Implementation code of the aspect.
 - [executed Around, Before or After **Join point**]
 - [Associated with **Join Point** through a **Pointcut**]
- **Join point**
 - Where **Advice** code is applied in application
 - [Always class methods in Spring AOP]
- **Pointcut**
 - An expression that defines a set of **Join points**

AOP Terminology Relationships

- Aspect = Advice + Pointcut

Defines Class as Aspect

- @Aspect

```
public class LoggingAspect {
```

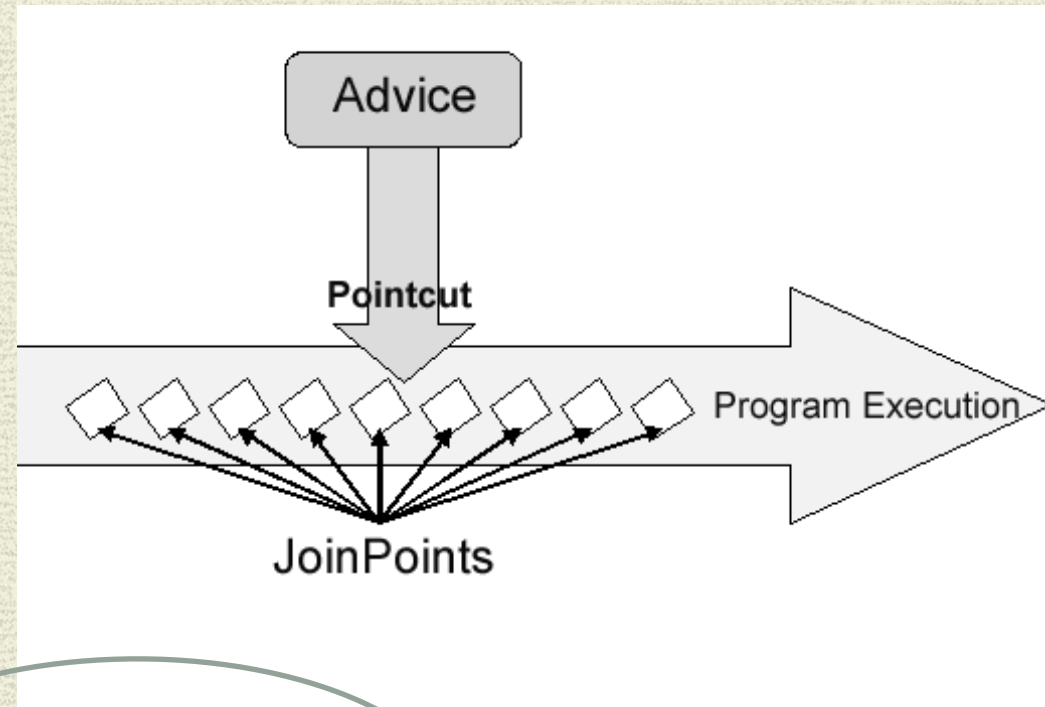
Advice – ‘When’ to
apply crosscutting
function

- @Before ("execution(* edu.mum.service..*(..))")

- public void logIt(JoinPoint joinPoint) {

Advice Method
The Crosscutting Function

Pointcut – Defines the JoinPoints
[Methods in the application]
to apply crosscutting function to.



Main Point

- Aspect Oriented Programming allows us to consolidate in one place the repetitive code that is scattered throughout an application, increasing the maintainability and clarity of the logic.
- ***Science of Consciousness: Refining the function of the nervous system leads to improvement in physical health and mental clarity***

Static & Dynamic AOP in Spring

- **Static [AspectJ]**

Cross-cutting logic is applied at ***compile time***

Byte code modification [better performance]

Aspect can be applied to fields [more join points]

Any Java code

Dynamic [Spring AOP]

Cross-cutting logic applied at ***run time***

Proxy based approach [simple to use]

Aspects applied to methods only

Spring Managed beans

NOTE: @AspectJ is a subset of AspectJ that declares AOP annotations. We will use this subset with Spring AOP without FULL AspectJ implementation

Spring AOP - PROXY

- **JDK Dynamic Proxies**

Interface-based proxy

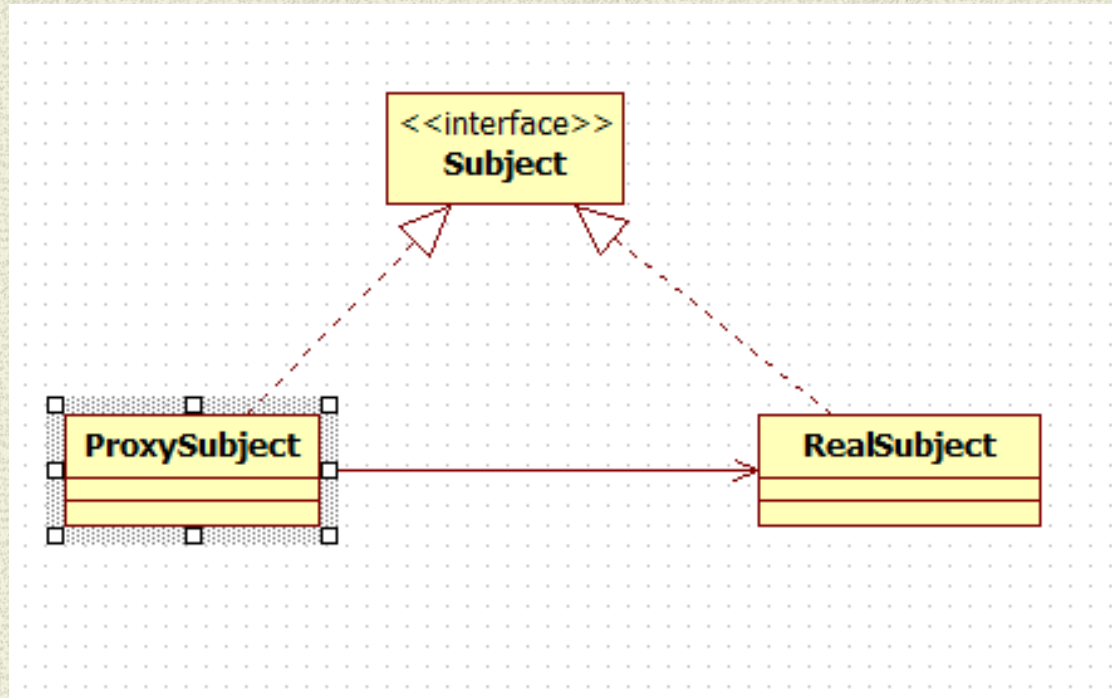
Recommended choice [Spring **IS** interface based]

CGLib [(Byte) Code Generation Library]

Use if Concrete Classes need proxy

Generated subclass used as proxy

Spring AOP - Proxy Pattern



- **Subject** - Interface implemented by the RealSubject
- **Proxy** - Controls access to the RealSubject
- **RealSubject** - the real object that the proxy represents.

Spring AOP Method Level Proxy “gotcha”

- Local or internal method calls within an advised class don't get intercepted by proxy, so advice method of the aspect does not get fired/invoked
- **Public class** Order
- @Logging
- **public** Order update(Order order) {
- **this**.updateAdjunct();
- **return** orderDao.update(order);
- }
- // This Never gets logged when called from update -
- // BECAUSE of Method level proxy "gotcha"
- @Logging
- **public void** updateAdjunct() {
- **return** ;

ADVICE TYPES

ADVICE	DESCRIPTION
@Before	executes before a join point
@AfterReturning	Executes if a join point completes normally
AfterThrowing	executes if a join point throws an exception
@After	executes if a join point executes normally OR throws an exception
@Around	Before AND after the join point. Also can end execution or throw exception

See [AOP Advice Types](#)

Point Cut Designators [PCD]

POINTCUT	DESCRIPTION	SYNTAX
<i>execution</i>	Matches methods Including visibility, return & parameters	<code>("execution(public * *.*.*(..))")</code>
<i>within</i>	Matches join points within "range" of types[Class]	<code>("within(*.*.*.*)")</code>
<i>target</i>	Matches where the target object is an instance of the given specific type [Class]	<code>("target(pkg.pkg.pkg.class)")</code>
<i>args</i>	<ul style="list-style-type: none"> Matches where the arguments are instances of the given types [Plus binding function] 	<code>("args(..)")</code>
@annotation	Matches methods where the given annotation exists	<code>@annotation(annotationName)</code> Supported Pointcuts

PCD Examples

Implicit - PCD expression inside the Advice annotation

The .* matches zero or more characters terminates match with NEXT '.'

Match all Classes; all methods in package *impl* that are public
AND have a void return value

```
@Before("execution( public void edu.mum.service.impl.*(..))")
```

Spring AOP recognizes PUBLIC only

Match Classes[and methods]in package service PLUS ALL subpackages

```
@Before("within(edu.mum.service..*)")
```

The (..) pattern matches any number of parameters (zero or more).

Match all instances of OrderService

Must specify "specific" Class

```
@Before("target(edu.mum.service.OrderService)")
```

Match all methods with a signature of Integer,Product followed by zero or more args

```
@Before("args(Integer,edu.mum.domain.Product,..)")
```

Notice path to Class

More on args() Pointcut

`@Before("execution(* edu.mum.service.impl.*(Integer))")`

is the same as

`@Before("execution(* edu.mum.service.impl.*(..)) && args(Integer)")`

`public void testExecution()`

However

The args pointcut also provides the “option” of binding the parameters to the Advice method.

It does so by declaring the instance *name* versus the instance *type*:

`@Before ("args (counter) ")`

`public void testExecution(Integer counter) {`



On the otherhand this will result in an Exception:

`@Before("args(Integer)")`

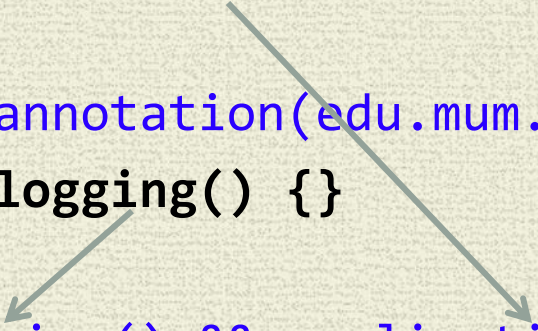
`public void testExecution(Integer counter)`



Because you declared “binding param”

Explicit Pointcut Declaration

- `@Pointcut("execution(* edu.mum..*(..))")`
- `public void applicationMethod() {}`
- `@Pointcut("@annotation(edu.mum.aspect.annotation.Logging)")`
- `public void logging() {}`
- `@Before("logging() && applicationMethod()")`
- `public void logIt(JoinPoint joinPoint) {}`



- Explicit Pointcuts are invoked based on the pointcut method signature.

- They can be logically combined with the following logical operators:

and [&&] or [||] not [!]

The operators can be either symbols or text

Pointcut & Advice Method Signatures

• Advice Method Parameters

- Any advice method **MAY** declare as its first parameter, a parameter of type **org.aspectj.lang.JoinPoint** [AKA application method – can “introspect”]

```
@Before("logging() && applicationMethod()")
public void logIt(JoinPoint joinPoint) {
```

Parameters declared in an (`"args(..)"`) expression will be passed to the advice method [for binding]












```
@Before("args(num,productAdd, name)")
public void method(Integer num,Product productAdd,String name)
```

Parameters can also be declared in an explicitly defined Pointcut

```
@Pointcut("args(name,count)")
public void applicationMethod(String name, Integer count)
```


Configure Spring AOP

pom.xml dependencies:

Dependencies	
	spring-core (managed:4.2.4.RELEASE)
	spring-context (managed:4.2.4.RELEASE)
	spring-tx (managed:4.2.4.RELEASE)
	spring-orm (managed:4.2.4.RELEASE)
	spring-aop (managed:4.2.4.RELEASE)
	aspectjrt (managed:1.8.7)
	aspectjweaver (managed:1.8.7)
	hibernate-entitymanager (managed:4.3.11.Final)
	mysql-connector-java (managed:5.1.38)
	log4j (managed:1.2.17)
	slf4j-log4j12 (managed:1.7.13)

applicatonContext.xml

```
xmlns:aop="http://www.springframework.org/schema/aop"
```

```
http://www.springframework.org/schema/aop
```

```
http://www.springframework.org/schema/aop/spring-aop.xsd ">
```

THIS results in the creation of *Spring AOP proxies*.

```
<aop:aspectj-autoproxy />
```

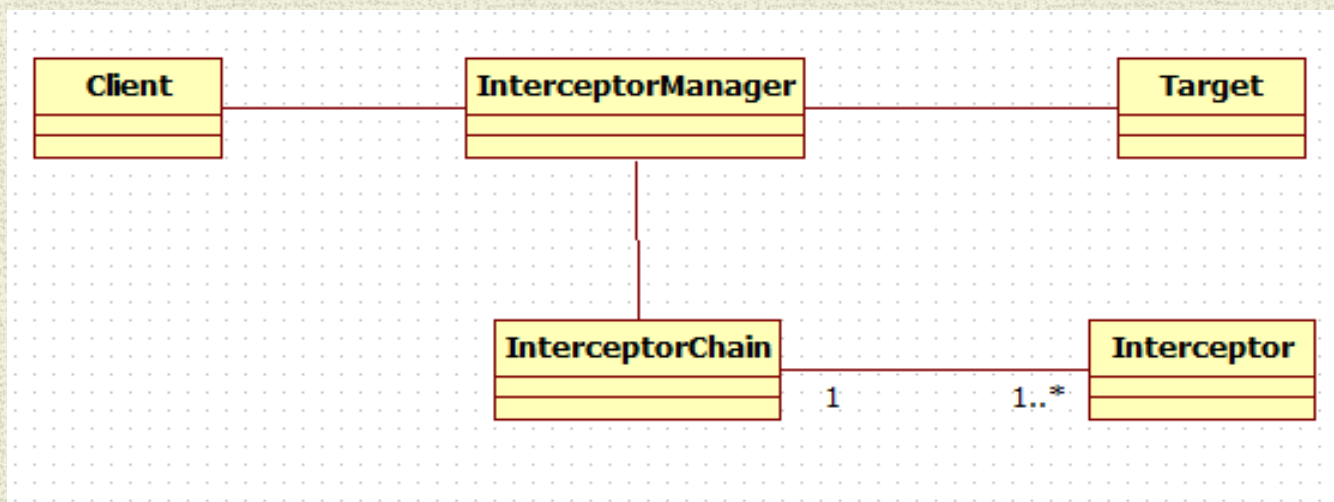
References AOP namespace to
Activate AspectJ annotations

See AOP Demo

Interceptor Chain

Core J2EE Pattern - Intercepting Filter

Spring AOP models an aspect as an *interceptor*, maintaining a chain of interceptors "around" a join point



Annotation based ordering is implemented with the
@Order annotation

```

@Aspect
@Order(1) default is Ordered.LOWEST_PRECEDENCE. = 2147483647
public class LoggingAspect {

```

On the way in to a joinpoint, the aspect with lowest Order value gets executed first.
On the way out from the joinpoint, the aspect with highest Order value gets executed first.

Spring Declarative Transaction Management An AOP Example

Spring Transaction Management is based on Spring AOP

Declarative Transaction Management allows for:

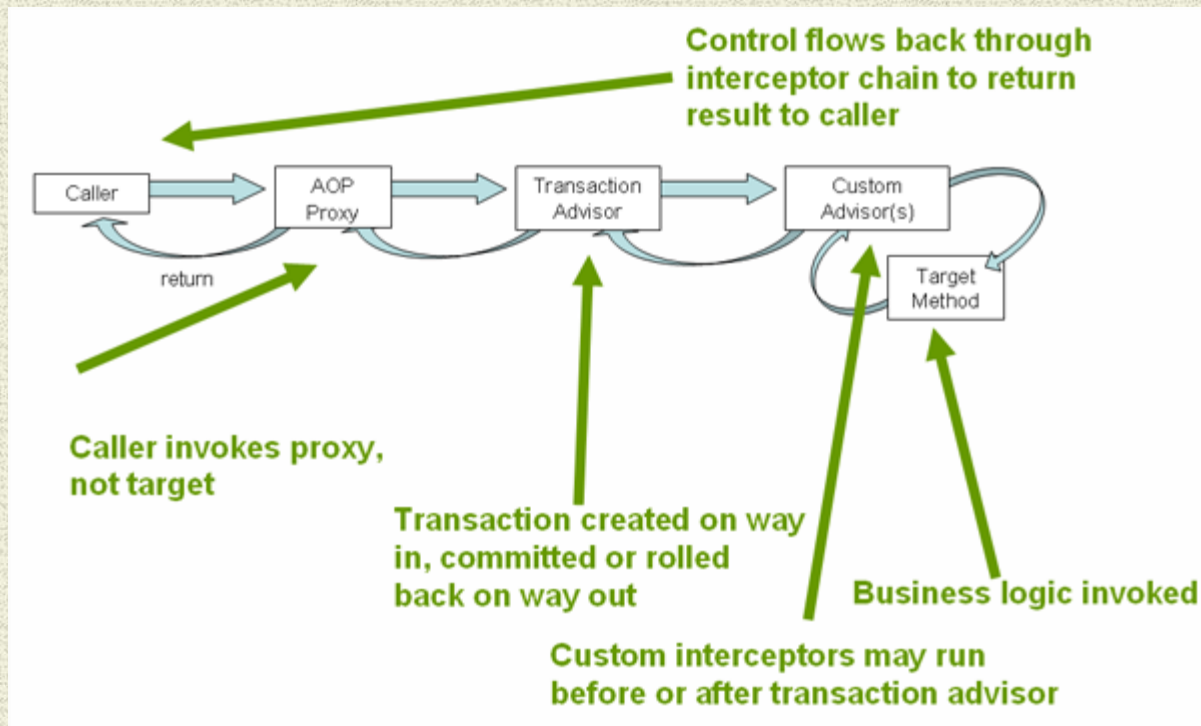
- Customized Transactional Behavior [through AOP]

- Insert custom behavior in the case of transaction rollback.

- Add custom advice along with the transactional advice.

Spring Transaction - AOP-based

Spring Framework's declarative transaction support is based on Spring AOP
Transactional advice is driven by *metadata* (XML- or annotation-based)

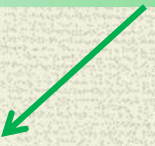


Let's examine at a XML metadata example to see the AOP configuration...

Declarative AOP XML based

- XML-Based Transaction declaration [.vs. @Transactional]
- In applicationContext.xml:
- **Advisor** - Associate pointcut with Advice
- ```
<aop:config>
 <aop:advisor pointcut="within(edu.mum.service.impl.*)"
 advice-ref="transactionAdvice" order="200" />
</aop:config>
```
- ```
<tx:advice id="transactionAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"
      rollback-for="java.lang.Exception" />
  </tx:attributes>
</tx:advice>
```

Interceptor chain order



Downside to AOP

- Functionality gets fragmented across source files and hard to understand.
You don't have a clear overview of what code runs when
Problematic to debug the AOP based application code.
Code Inspection and reviews are challenging
AOP can be too complex for application in the Enterprise
There are only a handful of cross-cutting concerns
It can be “over-used”

**A Little Bit of Aspect-Oriented Programming Is Just Enough
- The Gartner Reports**

Main Point

- AOP applied judiciously can make an application more efficient and effective.
- ***Science of Consciousness: Regular practice of the TM technique makes an individual more efficient and effective.***

