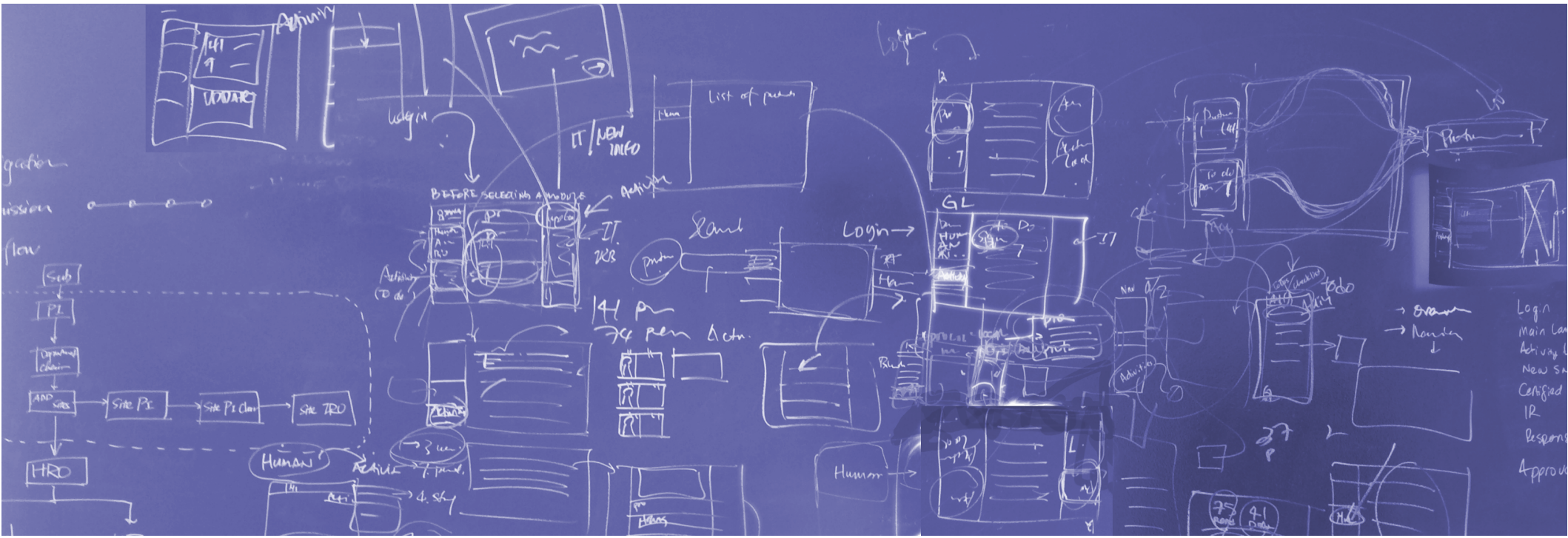
# CS544: Enterprise Architecture




## Lesson 7: Spring: The Application Service Layer & Aspect Oriented Programming

**Maharishi University of Management**

**Computer Science Department**

Orlando Arrocha

September 2016

# CS544: Enterprise Architecture

# Course Overview Chart

**Enterprise Architecture**

## Actions in Accord with Laws of Nature

*Structuring Software on Layers to Support Building Robust Enterprise Applications*

| Week | | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
|---|---|---|---|---|---|---|---|
| **Theme I Hibernate** Rest and Activity are the Steps of Progress | AM | **Introduction to EA** Wholeness is contained in every part | **Entity Mapping, Inheritance, Collections Persistence API** Outer depends on Inner | **Associations & Complex Mapping** Knowledge has organizing power | **Transactions, Concurrency & JPQL** Thought leads to action-achievement-fulfillment | **Optimization & Integration** Enjoy greater efficiency and accomplish more | **Exam** |
| | PM | Lab 1 Setup Env | Lab 2 Creating Entities | Lab 3 Associations | Lab 4 Query Data | Review | Extra Credit Hibernate |
| | Eve | Reading | | | | | |
| **Theme II Spring** Do Less and Accomplish More | AM | **Container & DI** Purification leads to progress | **Service Layer & AOP** Life is found in layers | **Scheduling, Hibernate & Tx** Harmony exists in diversity | **Spring MVC & Validation** Do Less and Accomplish More | **Spring Security & Spring Data** Seek the highest | **Exam** |
| | PM | Lab 5 DI | Lab 6 AOP | Lab 7 Transactions | Lab 8 Web MVC | Review | Extra Credit Spring |
| | Eve | | | | | | |
| **Theme III Integration** The Whole is Greater than the Sum of the Parts | AM | **JMS, AMQP & RMI** Order is present everywhere | **Spring Web Services** Rest and activity are the steps of progress | **Web Flow, Mobile & Integration** Knowledge is gained from inside and outside | **Microservices with Spring** Knowledge is structured in consciousness | **Project** … action leads to achievement … | |
| | PM | Lab 9 Security & AMQP | Lab 10 REST WS | Lab 11 Web Flow | Project Thought leads to action | | |
| | Eve | | | | | | |
| **Theme IV Project** The Field of All Possibilities is the Source of All Solutions | AM | **Project** … | | **Project Presentations** … and achievement leads to fulfillment | | | |
| | PM | | | | | | |
| | Eve | | | | | | |

# Wholeness of the Lesson

**Lesson 7**
**THE APPLICATION SERVICE LAYER & ASPECT ORIENTED PROGRAMMING**
***Life is found in layers***

Do not repeat yourself on code.

***Science of Consciousness:*** *Creative intelligence naturally enriches all levels of life.*

# Main Points

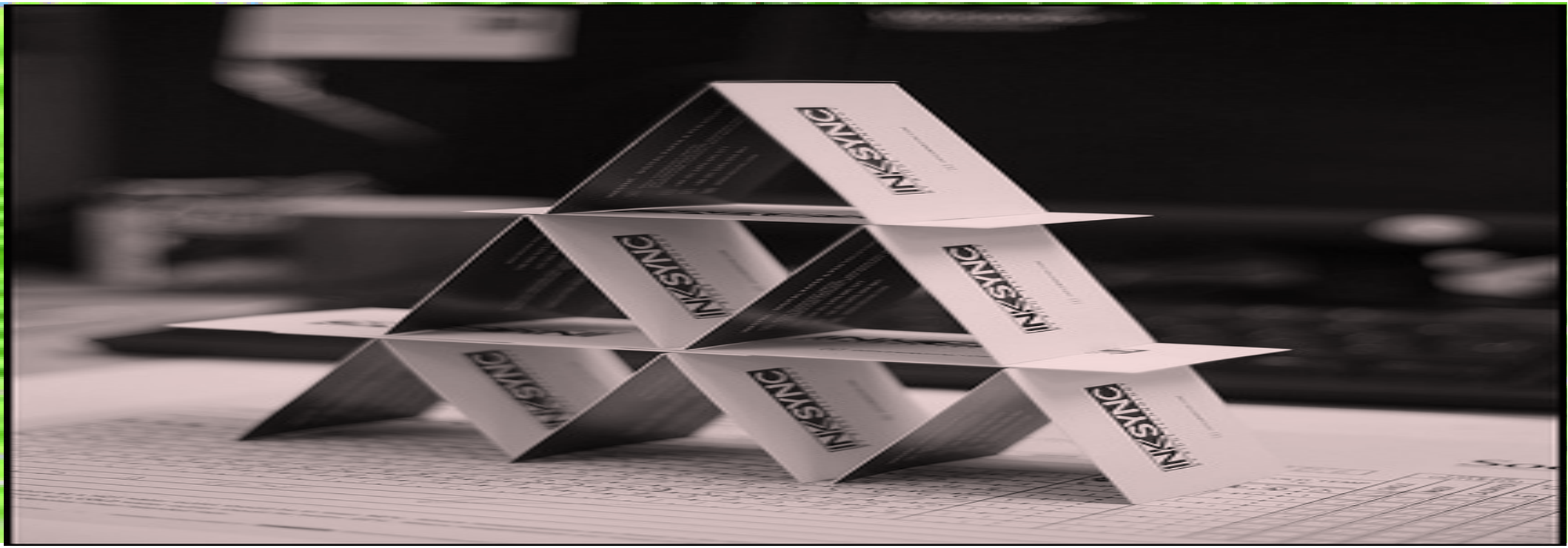
1) Separation of concerns allows us to structure the applications in a modular style, abstracting the logic from upper layers from the actual implementation performed by lower layers, and simplifying the dependencies and support between the layers. ***Science of Consciousness:*** *Living unbounded awareness within boundaries.*

2) Aspect Oriented Programming allows us to consolidate in one place the repetitive code that is scattered out throughout an application, which increases the maintainability and clarity of the logic. ***Science of Consciousness****: Purification leads to progress.*

# Overview

- We will cover
  - The Application Service Layer
    - Most common implementation problems and how to avoid them
  - Aspect Oriented Programming
    - What is it?
    - How to use it with Spring?
    - Types of Advice
    - Pointcut Expressions
    - Advantages of using AOP

# The Application Service Layer

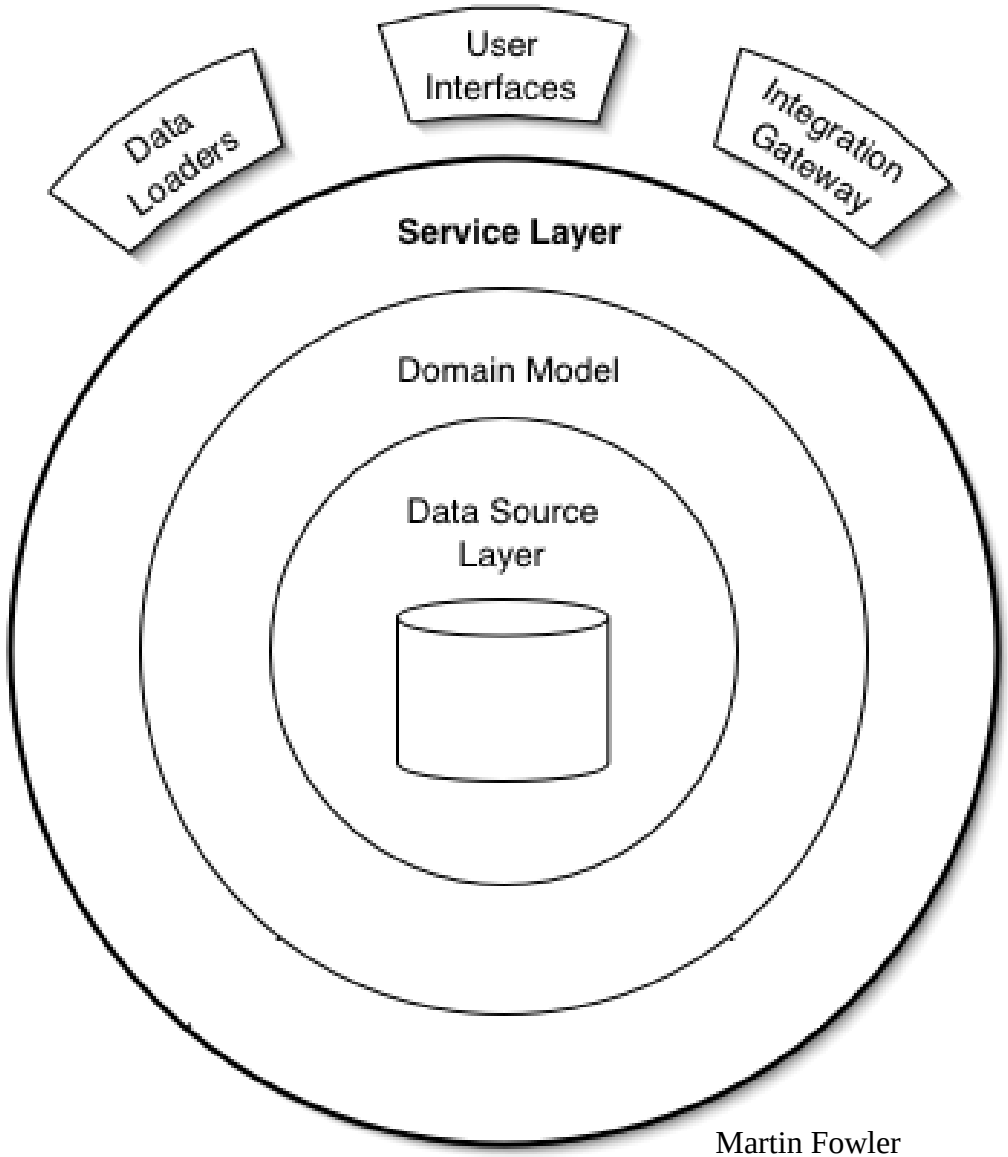



Simplicity is prerequisite for reliability.

**– Edsger W.Dijkstra**

# Typical Application Layer's Separation

- Presentation
  - responsible of processing user's input and returning the correct response back to the user
  - The control layer communicates only with the service layer
- Service layer
  - Acts as a transaction boundary
  - It is responsible of authorization and contains the business logic of our application
  - Manages the domain model objects
  - communicates with other services and the repository layer
- Data Source Layer
  - It is responsible of communicating with the data storage



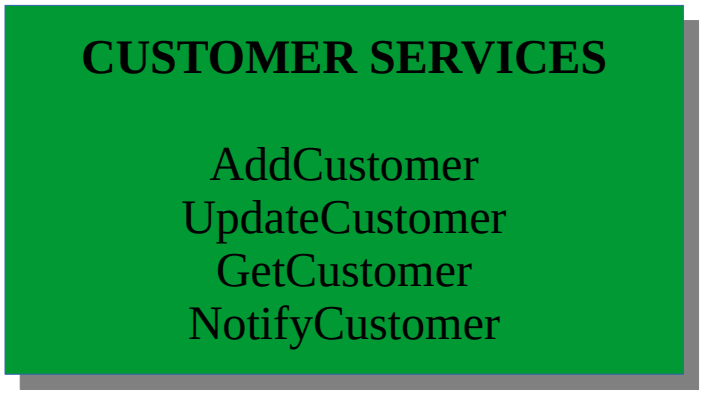

Martin Fowler

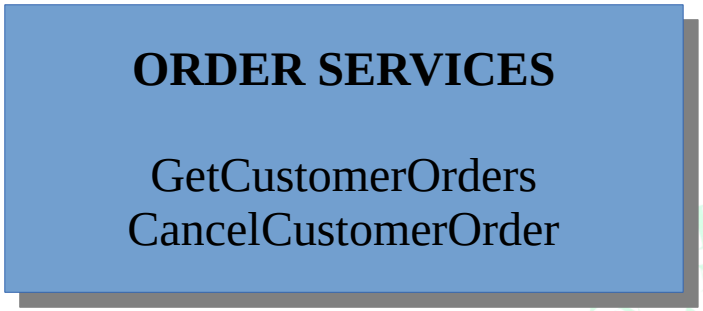
# What's Wrong With the Service Layer?

- Typically the Service Layer becomes a monolithic service
  - TOO MANY Responsibilities
  - The domain model objects are used only to store the data of the application:
    - Session Facade Pattern – Anemic Domain Model Anti-Pattern
  - The business logic lies in the service layer which manages the data of the domain objects
  - The business logic is scattered around the service layer
  - If the same business rule is needed in multiple service classes, the odds are that the rule is simply copied from another service
  - The service layer has one service class per each entity of the application domain model class
  - The Service Layer is not composed of loosely coupled services, with only one responsibility each

# Cohesion


- **Cohesion** refers to the degree to which the elements of a module belong together.

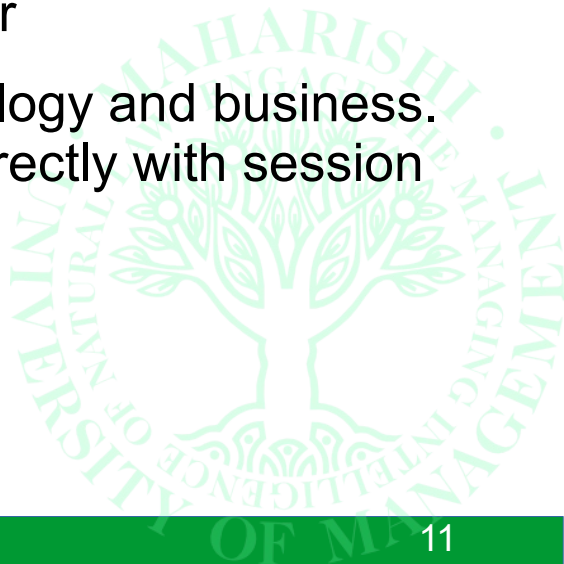



- **Command Query Responsibility Segregation (CQRS) Pattern** promotes further separation. Split the conceptual model into separate models for update and reading.

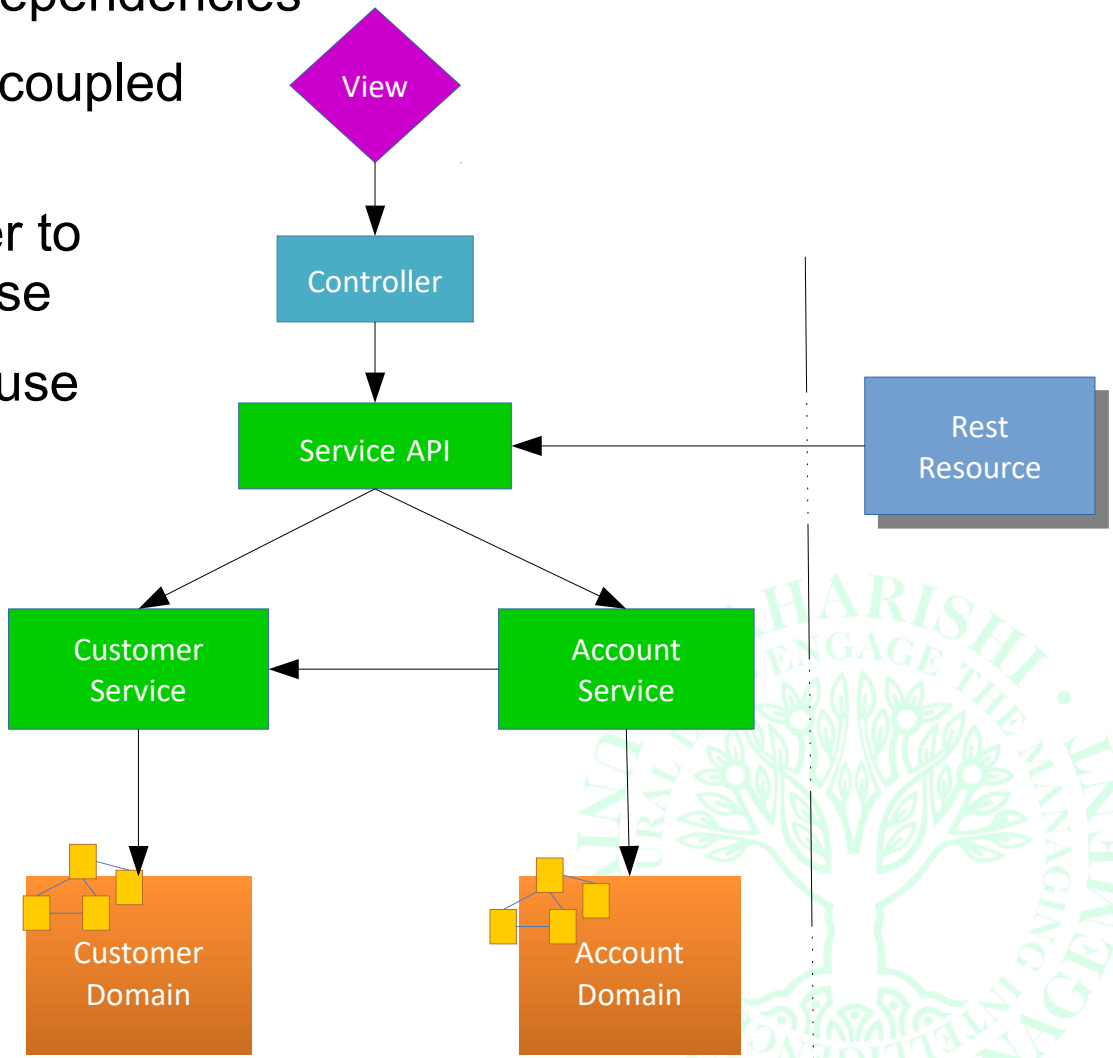
# Put the Business Where it Belongs

- The business layer is responsible of ensuring consistency applying business rules and providing persistence.
- The business layer must be separated from the plumbing artifacts
- Move the business logic from the service layer to the domain model classes
  - The responsibilities of our code should be divided in a logical way
  - The service layer takes care of the application logic and our domain model classes takes care of the business logic
  - The business logic of our application must be found on a single place
  - If we need to verify how a specific business rule is implemented, we always know where to look for it
  - The source code of the service layer becomes cleaner
- It is important to have a clean separation between technology and business. This allows us to implement a domain-driven approach directly with session beans and JPA



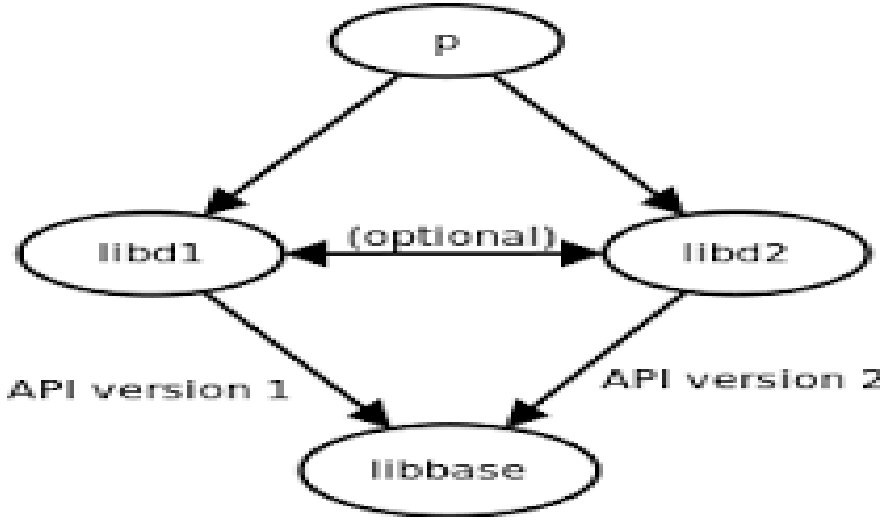
# Divide and Conquer

- Divide the entity specific services into smaller services which serves only a single purpose
  - Each service class has a logical set of responsibilities.
  - Each service class has less dependencies
  - They are smaller and loosely coupled components
  - The service classes are easier to understand, maintain and reuse
- Expose a consistent and easy to use business API



# Anti-Pattern: Cycles & Diamonds

- Circular dependencies
  - When a package/service/layer depends on another, which have a dependency graph that requires the first



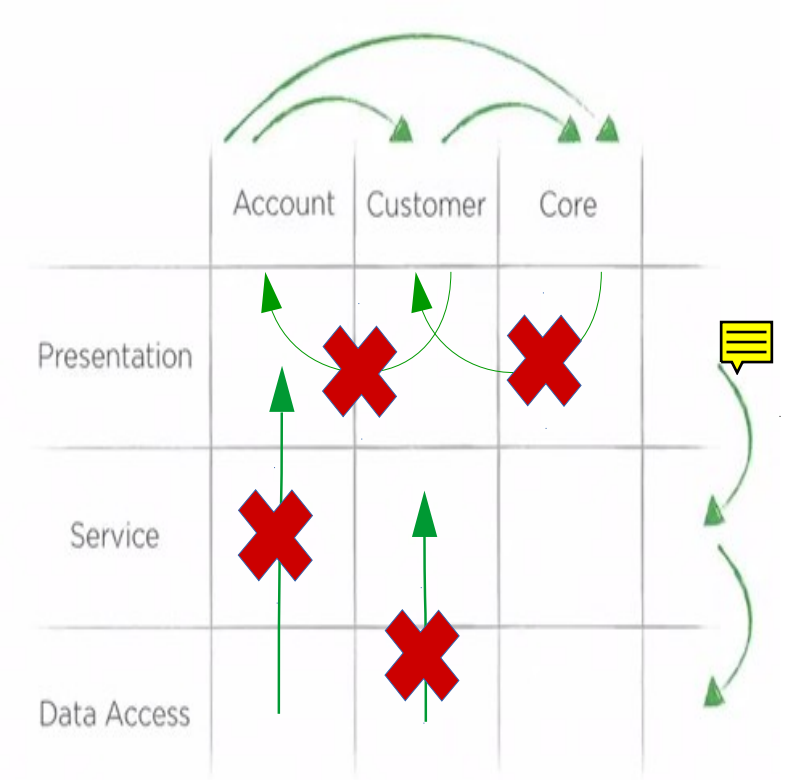

- Use policies for your development to avoid this issue
- Use Java tooling to remove the cycles
  - http://clarkware.com/software/JDepend.html

# Slice Your System into Packages

- Remove cycle dependencies between packages
  - Top layers should use lower layers
  - Division between business logic slices should be enforced
- Separate subsystem first and then the application layers

```
• account.domain.*     .customer.domain.*     .core.domain.*
.account.service.*     .customer.service.*    .core.service.*
.account.repository.*  .customer.repository.* .core.repository.*
```

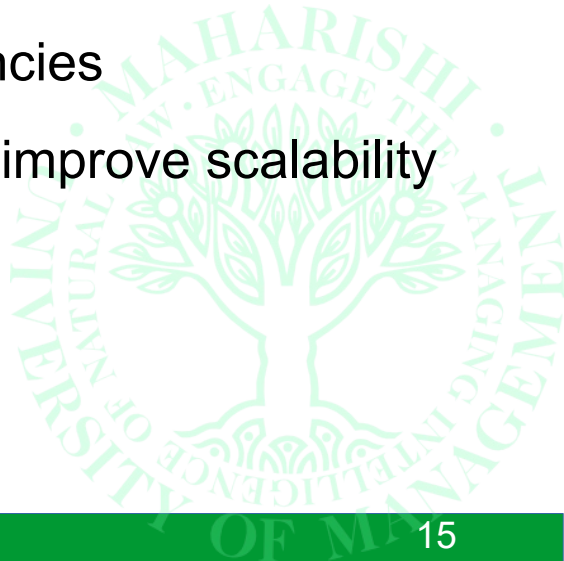
  If you use layers first to divide your packages, then your subsystems will be mixed – exposing internals for everyone that has access to the package




- Subsystem first keeps
  - Implementations internal
  - Domain and business are encapsulated per package
- Expose (make public) only the interfaces and entry points to your subsystem, other classes should be protected

# So what should we do?

- First define the domain and then put the business logic on it
- Define the services of you system and keep things that are related together and separate the other ones
- On the service layer you should put the business logic that couldn't be set kept in the domain
- Don't put business code in controllers – that is to set logic for the application flow
- Keep services as simple as possible
- Create a service API to abstract the implementation of your solution so the external systems that will use your services won't be tied to the implementation
- Make your systems modular, and avoid circular dependencies
- For micro services, try to implement the CQRS pattern to improve scalability



# Spring's Multiple Configuration Files

- Since our application itself is often large, and divided in layers, it makes sense to separate a large Spring configuration file along the same lines

```
public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new
       ClassPathXmlApplicationContext(accountService-context.xml);

    IAccountService accountService = context.
          getBean("accountService", IAccountService.class);
...
```

Spring promotes separation of concerns in many aspects

accountService-context.xml

```
<beans …>
    <import resource="dataAccess-context.xml"/>
    <import resource="jmsService-context.xml"/>
    <bean id="accountService"
              class="bank.service.AccountService">
        <constructor-arg index="0" ref="accountDAO" />
        <constructor-arg index="1" ref="jmsSender" />
    </bean>
</beans>
```

dataAccess-context.xml

```
<beans …>
    <bean id="accountDAO"
              class="bank.dao.AccountDAO" />
</beans>
```

jmsService-context.xml

```
<beans …>
    <bean id="jmsSender"
              class="bank.jms.JMSSender" />
</beans>
```

# Spring's Multiple Configuration Files (Cont.)

## Through the ApplicationContext

```
public class Application {
  public static void main(String[] args) {
    String[] xmlResources ={"accountService-context.xml",
                            "dataAccess-context.xml",
                            "jmsService-context.xml"};
    ApplicationContext context = new
              ClassPathXmlApplicationContext(xmlResources);

    IAccountService accountService = context.
          getBean("accountService", IAccountService.class);
...
```

accountService-context.xml

```
<beans …>
    <bean id="accountService"
              class="bank.service.AccountService">
        <constructor-arg index="0" ref="accountDAO" />
        <constructor-arg index="1" ref="jmsSender" />
    </bean>
</beans>
```

dataAccess-context.xml

```
<beans …>
    <bean id="accountDAO"
              class="bank.dao.AccountDAO" />
</beans>
```

jmsService-context.xml

```
<beans …>
    <bean id="jmsSender"
              class="bank.jms.JMSSender" />
</beans>
```

# Main Points

1) Separation of concerns allows us to structure the applications in a modular style, abstracting the logic from upper layers from the actual implementation performed by lower layers, and simplifying the dependencies and support between the layers. ***Science of Consciousness:*** *Living unbounded awareness within boundaries.*

# Aspect Oriented Programming

When I am working on a problem I never think about beauty.
I think only how to solve the problem.
But when I have finished, if the solution is not beautiful, I know it is wrong.

**– R. Buckminster Fuller**

# Cross-Cutting Concerns

- **An Aspect:** Is a functionality that is scattered or tangled through code, making it very hard to understand and maintain.
- Generally it is the repetitive code to manage things like database connections, transactions, security, or logging messages (a concern).

```
void transfer(Account fromAcc,
              Account toAcc, int amount,
              User user, Logger logger)
    throws Exception {

  logger.info("Transferring money…");

  if (!isUserAuthorised(user, fromAcc)) {
    logger.info("User has no permission.");
    throw new UnauthorisedUserException();
  }
  if (fromAcc.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();
  }

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);
  database.commitChanges();

  logger.info("Transaction successful.");
}
```

*Cross-cutting concerns span over different operations/classes*

Security

Logs

```
public withdraw(float amount) {
  logger.info("Withdraw amount...");
  this.balance -= amount;
  logger.info("Withdraw succesful!");
}
```

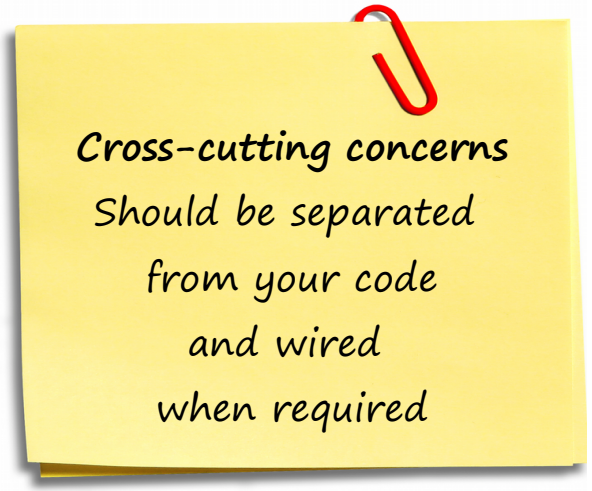
Validations

Transaction

# Cross-Cutting Concerns (CONT)

- While your business logic only needs to do something like this:

```
void transfer(Account fromAcc,
              Account toAcc, int amount){

   fromAcc.withdraw(amount);
   toAcc.deposit(amount);

}
```

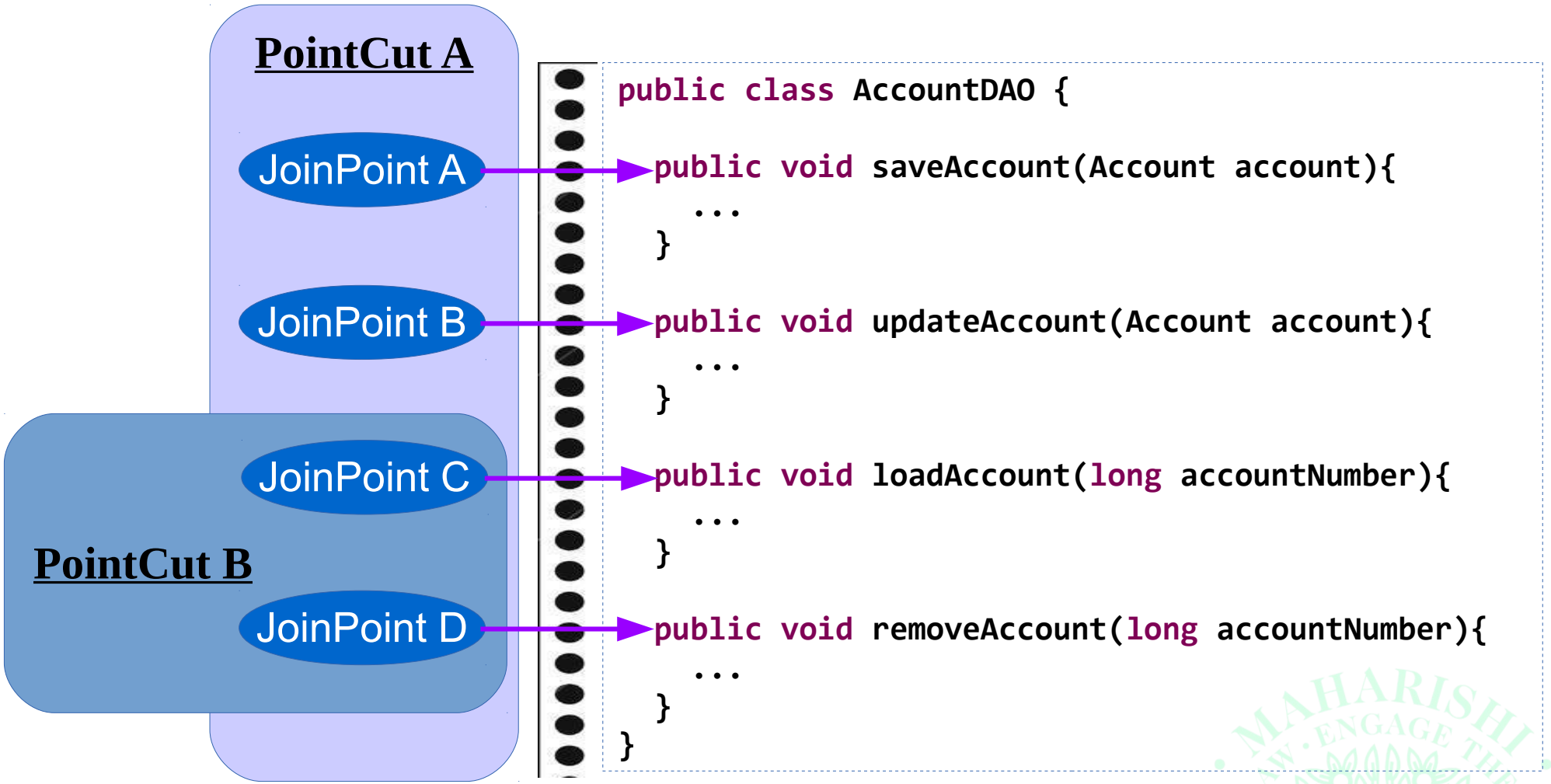



```
public withdraw(float amount) {

  this.balance -= amount;

}
```

# AOP Concepts

- **Aspect:** The modularization of a crosscutting concern into a class
  - *What we want to do*
- **Joinpoint**: A specific point in the code. It could be a field, a constructor, or a method, but Spring only supports method join points
- ***Pointcut****: A predicate expression that matches a collection of one or more joinpoints*
  - *Where it should be applied*
- **Advice:** The execution process of the *aspect* before or after calling the *joinpoint* method at the target object
  - *When it will be applied*
- **Weaving:** Links the *advice* code together with the target code at the corresponding *pointcuts* to generate an advised proxy or class
  - *The process that links all the parts*

# JoinPoint and PointCut




**Pointcut A**: All methods of the AccountDAO class

**Pointcut B**: All methods of the AccountDAO class that have 1 parameter of type long

# Aspect, Advice and Pointcut

```
@Aspect
public class TraceAdvice {

   // When it will be applied
   //                    Where it will applied
   @Before("execution(* accountpackage.AccountDAO.*(..))")
   public void tracebeforemethod(JoinPoint joinpoint) {
      // What we want to do
      System.out.println("before execution of method "
                  +joinpoint.getSignature().getName());
   }

   // When it will be applied
   //                    Where it will applied
   @After("execution(* accountpackage.AccountDAO.*(..))")
   public void traceaftermethod(JoinPoint joinpoint) {
      // What we want to do
      System.out.println("after execution of method "
                  +joinpoint.getSignature().getName());
   }
}
```

Pointcut

The "Before" Advice

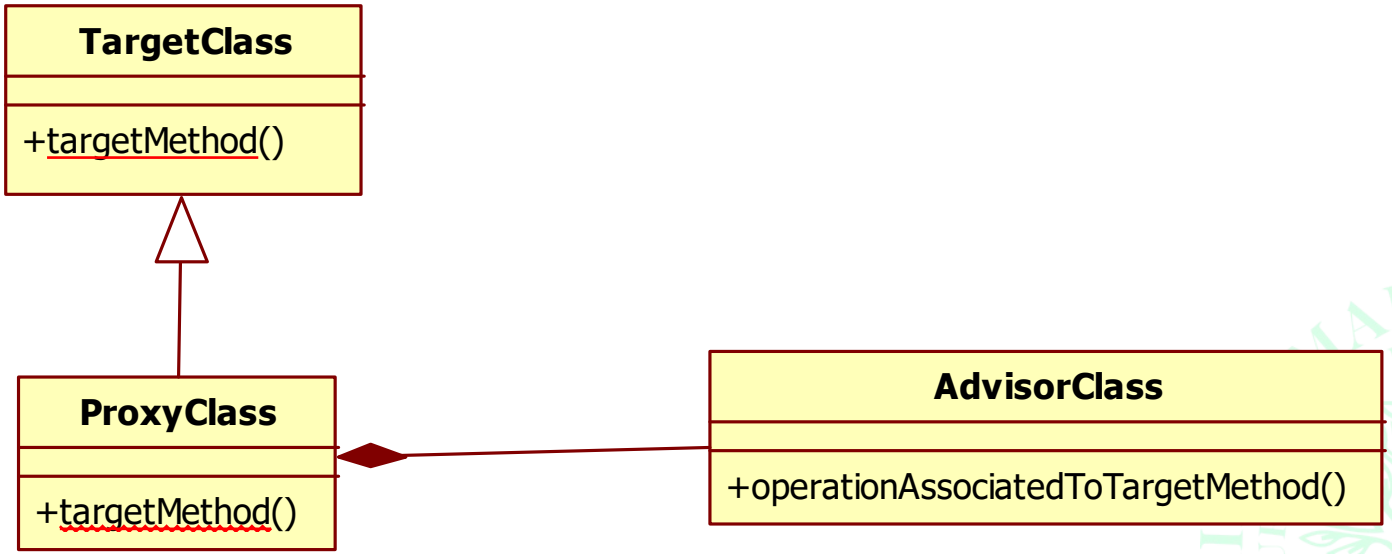
The "After" Advice

The Aspect (Cross-cutting concern) that will be applied

The Aspect (Cross-cutting concern) that will be applied

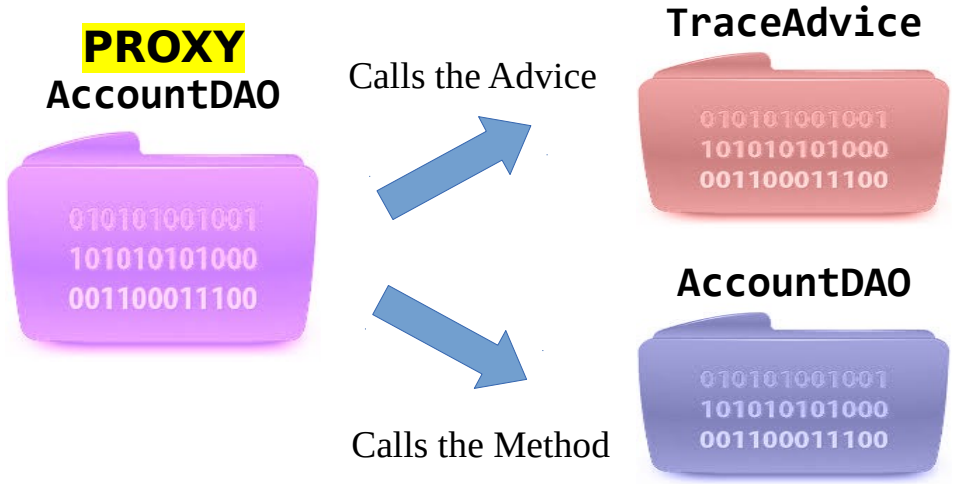
# Weaving with a Proxy

- The framework will create a proxy class that will override the target method so it could add the functionality
- The proxy will call the Advisor class method and the Traget class methods when it is appropriate. For example. If the advice is @Before, when the application calls the targetMethod(), it will be the proxy's targetMethod() and this one will execute the advise and then call the targetMethod() on the TargetClass

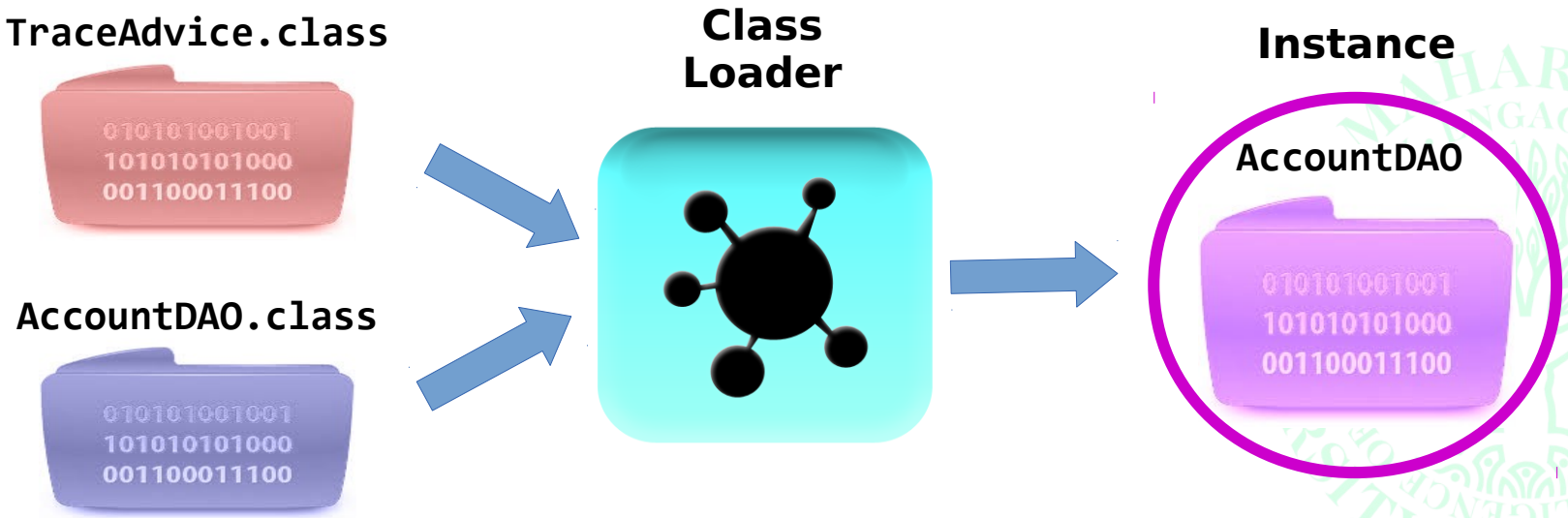


# Weaving

- There are two ways to weave advices with the target code
  - **Proxy-Based Weaving** (Spring's implementation)




  - **Load Time Weaving** (AspectJ's implementation)



# Spring 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-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <aop:aspectj-autoproxy/>

  <bean id="accountDAO" class="accountpackage.AccountDAO"/>
  <bean id="theTraceAdvice" class="aopadvice.TraceAdvice"/>
</beans>
```

```java
public class Application {

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");

    IAccountDAO accountDAO = context.getBean("accountDAO", IAccountDAO.class);
    accountDAO.saveAccount(new Acount("123") );
  }
}
```

# Code Execution

1) The application calls saveAccount()
2) The proxy receives the call
3) Then calls the before execution at the TraceAdvice
4) TraceAdvice prints

   ***before execution of method saveAccount***

5) Proxy calls the saveAccount() at the AccountDAO
6) Then calls the after execution at the TraceAdvice
7) TraceAdvice prints

   ***after execution of method saveAccount***

# Active Learning

- How does AOP help us improve our code?

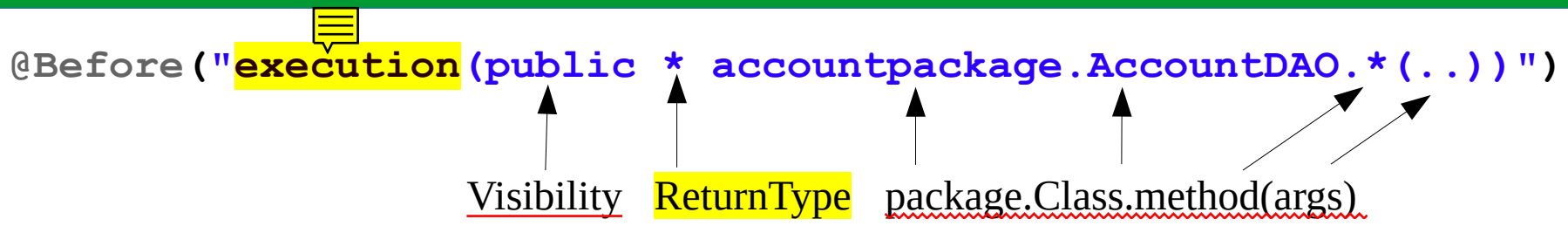
- What is a pointcut?

# Pointcut Execution Language

A good way to stay flexible is to write less code

**– Pragmatic Programmer**

# Declaring a Pointcut




- **Visibility** (OPTIONAL, Cannot be *)
  - private
  - public
  - protected
- **Return type** (REQUIRED, Can be *)
  - The return type of the corresponding method(s)
- **package.Class.method**(**args**)
  - Name of the package (OPTIONAL, Can be *)
  - Name of the class (OPTIONAL, Can be *)
  - Name of the method (OPTIONAL, Can be *)
  - Arguments (REQUIRED, Can be (..))

# Pointcut Expression Designators

| Designators | Description |
|---|---|
| execution | for matching method execution join points. This is the primary pointcut designator you will use when working with Spring AOP |
| within | limits the execution of a method declared within a matching type |
| this | limits matching to join points where the bean reference (Spring AOP proxy) is an instance of the given type |
| target | limits matching to join points where the target object (application object being proxied) is an instance of the given type |
| args | limits matching to join points where the arguments are instances of the given types |
| @target | limits matching to join points where the class of the executing object has an annotation of the given type |
| @args | limits matching to join points where the runtime type of the actual arguments passed have annotations of the given type(s) |
| @within | limits matching to join points that have the given annotation (annotation at the target method) |
| @annotation | limits matching to join points where the subject of the join point has the given annotation (annotation at the target class) |

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html

# Examples

- Any public method:

```
execution(public * *(..))
```

- Any method with a name beginning with "**set**":

```
execution(* set*(..))
```

- Any method defined by the AccountService:

```
execution(* mum.edu.service.AccountService.*(..))
```

- Any method defined in the service package:

```
execution(* mum.edu.service.*.*(..))
```

- Any method defined in the service package or a sub-package:

```
execution(* mum.edu.service..*.*(..))
```

- Any method with the first argument **long** and the second argument is **String**:

```
execution(* *(long,String))
```

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html

# Pointcut Composition

- Pointcut expressions can be combined with the boolean operators:

  &&  ||  !

- Their word forms are allowed as well:

  and  or  not

For more detail see the aspectj docs at:

http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html

# More Examples

- Any method within the mum.edu.service package:

```
within(mum.edu.service.*)
```

- Any method within the mum.edu.service package or a sub-package :

```
within(mum.edu.service..*)
```

- Any method of class order.payment.procesPaymentImpl

```
target(order.payment.ProcesPaymentImpl)
```

- Any method of the class order.procesPayment or order.VerifyPayment

```
target(order.ProcesPayment) || target(order.VerifyPayment)
```

- Any method that is annotated with the `@transactional` annotation

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

- Any method with 3 parameters with types `int, String, and mum.edu.Person`

```
args(int, String, mum.edu.Person)
```

- Any method that does not have one parameter of type `int`

```
!args(int)
```

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html

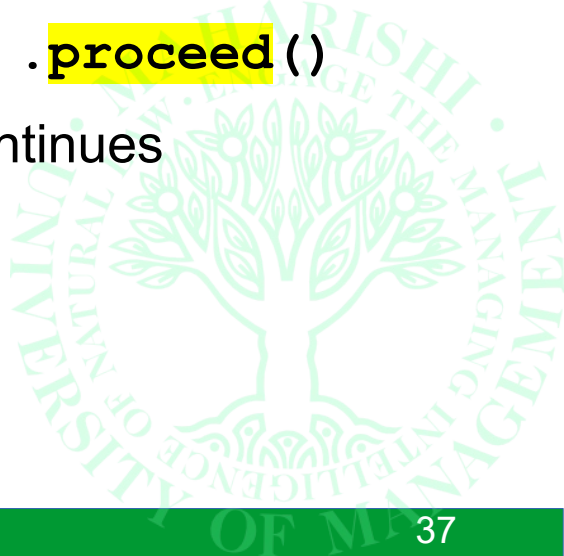
# Get Parameters with args

```
public class Customer {
...
  public String getName() {
    return firstname;
  }
  public void setFullName(String first, String last)
{
    this.firstname = first;
    this.lastname = last;
  }
...
```

```
@Aspect
public class TraceAdvice {
  @Before("execution(* mypackage.Customer.setFullName(String,String)) &&
args(firstName, lastName)")
  public void tracemethod(JoinPoint joinpoint, String firstName, String lastName)
{
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("parameter name ="+firstName);
  }
}
```

# Types of Advice

- Before
  - First calls the advice method, then the target method
- After
  - First calls the target method, then the advice method
    - After Returning
      - Is called when the target object returns a result
    - After Throwing
      - Run after the method throws an exception
- Around
  - First calls the advice method
  - The advice methods can call the target object method `.proceed()`
  - And when the the logic returns, the advice method continues



# Returning, Throwing and Around

**After Returning**

```
@Aspect
public class TraceAdvice {
  @AfterReturning(pointcut="execution(* mypackage.Customer.getName(..))",
                  returning="retValue")
  public void tracemethod(JoinPoint joinpoint, String retValue) {
    System.out.println("method ="+joinpoint.getSignature().getName());
    System.out.println("return value ="+retValue);
  }
}
```

**After Throwing**

```
@Aspect
public class TraceAdvice {
  @AfterThrowing(pointcut="execution(* mypackage.Customer.myMethod(..))",
                 throwing="exception")
  public void tracemethod(JoinPoint joinpoint, MyException exception) {
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("exception message ="+exception.getMessage());
  }
}
```

**Around**

```
@Around("execution(* *.*.*(..))")
public Object profile (ProceedingJoinPoint call) throws Throwable{
  StopWatch clock = new StopWatch("");
  clock.start(call.toShortString());

  Object object= call.proceed();

  clock.stop();
  System.out.println(clock.prettyPrint());
  return object;
}
```

# Get Arguments from Joinpoint and ProceedingJoinPoint

```
@Aspect
public class TraceAdvice {
 @After("execution(* mypackage.Customer.setName(..))")
  public void tracemethodA(JoinPoint joinpoint) {
  Object[] args = joinpoint.getArgs();
  String name = (String)args[0];
  System.out.println("method ="+joinpoint.getSignature().getName());
  System.out.println("parameter name ="+name);
  }
}
```

```
@Aspect
public class CalcAdvice {
  @Around("execution(* Calculator.add(..))")
  public Object changeNumbers (ProceedingJoinPoint call) throws Throwable{
    Object[] args = call.getArgs();
    int x = (Integer)args[0];
    int y = (Integer)args[1];
    System.out.println("CalcAdvice.changeNumbers: x= "+x+"and y= "+y);

    args[0]=5;
    args[1]=9;
    Object object= call.proceed(args);

    System.out.println("CalcAdvice.changeNumbers: call.proceed returns "+object);
    return 26;
  }
}
```

Setting argument values

# Getting the Target Object

```
@Aspect
public class TraceAdvice {
 @After("execution(* mypackage.Customer.setName(..))")
   public void tracemethod(JoinPoint joinpoint) {
     Customer customer = (Customer)joinpoint.getTarget();
     System.out.println("method ="+joinpoint.getSignature().getName());
     System.out.println("customer age ="+customer.getAge());
  }
}
```

# Order of Execution

- When multiple advices will run on the same joinpoint, the order of execution will follow the configuration's file order

```
<aop:aspectj-autoproxy/>
<bean id="accountService" class="accountpackage.AccountService"/>
<bean id="traceAdvice1" class="aopadvice.TraceAdviceA"/>
<bean id="traceAdvice2" class="aopadvice.TraceAdviceB"/>
```

# XML Schema Based AOP with Spring

- Declaring an Aspect

```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">
      <aop:pointcut id="businessService"
         expression="execution(* com.xyz.myapp.service.*.*(..))"/>

      <!-- a before advice definition -->
      <aop:before pointcut-ref="businessService"
         method="doRequiredTask"/>

      <!-- an after advice definition -->
      <aop:after pointcut-ref="businessService"
         method="doRequiredTask"/>

      <!-- an after-returning advice definition -->
      <aop:after-returning pointcut-ref="businessService"
         returning="retVal"
         method="doRequiredTask"/>

      <!-- an after-throwing advice definition -->
      <aop:after-throwing pointcut-ref="businessService"
         throwing="ex"
         method="doRequiredTask"/>

      <!-- an around advice definition -->
      <aop:around pointcut-ref="businessService"
         method="doRequiredTask"/>
   ...
   </aop:aspect>
</aop:config>
```

# XML Schema Based AOP with Spring (CONT

- Declaring Advices

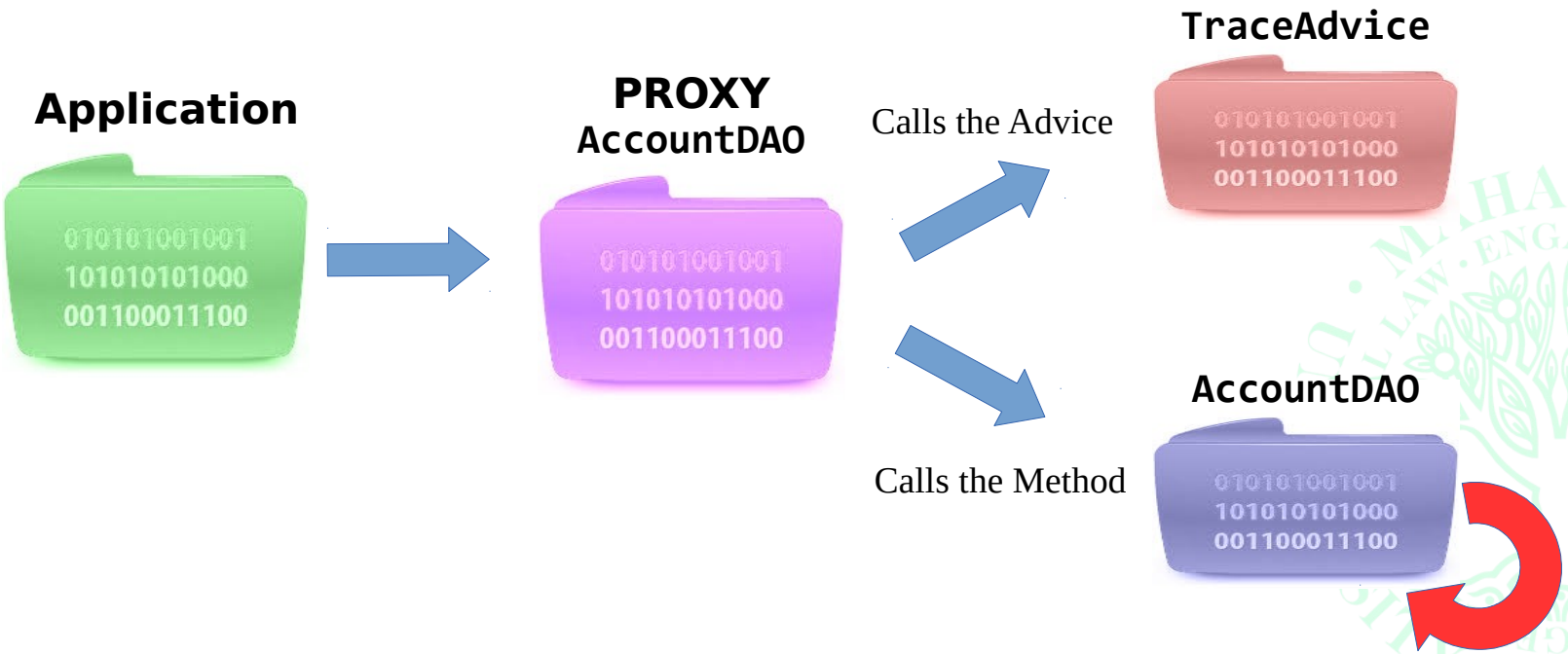
```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">
   ...
   </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

- Declaring a Pointcut

```
<aop:config>
   <aop:aspect id="myAspect" ref="aBean">

   <aop:pointcut id="businessService"
      expression="execution(* com.xyz.myapp.service.*.*(..))"/>
   ...
   </aop:aspect>
</aop:config>
```

# AOP Pros and Cons

- Advantages
  - Clean separation of concerns
  - No scattering code
- Disadvantages
  - No clear overview of how the execution will be performed during runtime
  - Pointcut expressions are parsed at runtime (prompt to errors)
  - When using proxies, be aware that calls from the target object to its methods won't be intercepted by the proxy



# Active Learning

- What are the 5 types of advice?

- Which advice type will allow you to skip the target method?

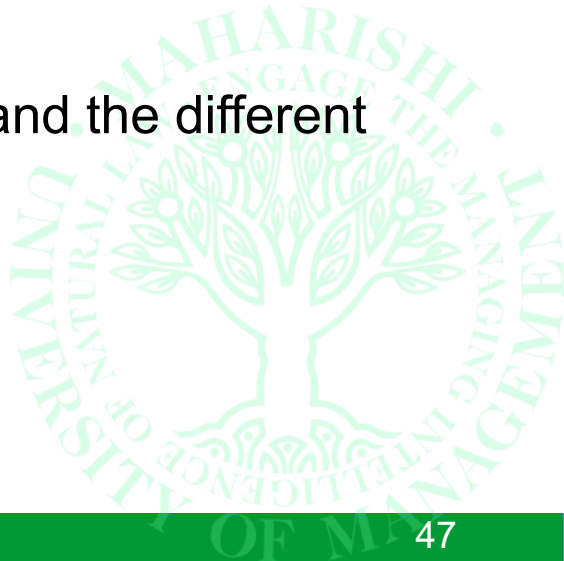
# Main Points

2. Aspect Oriented Programming allows us to consolidate in one place the repetitive code that is scattered out throughout an application, which increases the maintainability and clarity of the logic. ***Science of Consciousness****: Purification leads to progress.*

# Summary

In this lesson we covered

- We talked about the Service Layer
- Separation of Concern (SoC)
  - Separate business logic from (technical) plumbing code
  - Avoid code tangling
- Spring with multiple configuration files
- AOP is all about separation of concerns, the aspects that were previously tangled together are separated into different layers of code
- Don’t Repeat Yourself (DRY)
  - Write functionality at one place, and only at one place
  - Avoid code scattering
- We talked about the pointcut execution expressions, and the different types of Advice that Spring provides



# UNITY CHART

**CONNECTING THE PARTS OF THE KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE**

***The whole is more than the sum of its parts***

1) Loosely coupled applications are easier to maintain and upgrade.
2) Spring promotes best practice principles and helps you write better applications.

---

3) **Transcendental Consciousness:** infinite organizing power.
4) **Impulses in the Transcendental Field:** The all-encompassing nature of creative intelligence allows the development of its qualities in us all at once.
5) **Wholeness moving within itself:** The underlying unity of life is now on the surface of life, and permeating everything, every aspect of life.