Spring AOP

Code for DAO

Public void addAcountId(Acount acc,String userId)

{

// add code for loggin code

// add code for security chcek

// get session

//session.save

}

New Requirement

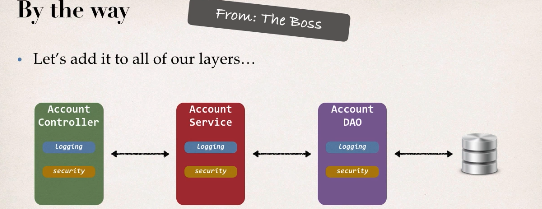
Add Logging methods to DAO : loggiin stament before the strt of dao method

New Requirement

Add Security methods to DAO : make user is authorized before running of dao method

TPSreport?

If requiremrent to add to all layers



Two main problme

1 code tangling : fro given method logging and security will be tnagled

2 code scattering : need to change looging and security we need to update all classes (think 100 classes)

Other possible solutions

Inheritance: every class would need to inheri from a base class

Can all classs extend from base class ? plus no multiple inheritnace

Delegation : classes would delegete logging,security calls

- still would need to update classes if we wnated to

- add / remove logging or security

- add new feature like auditing,APi management, instrumentation

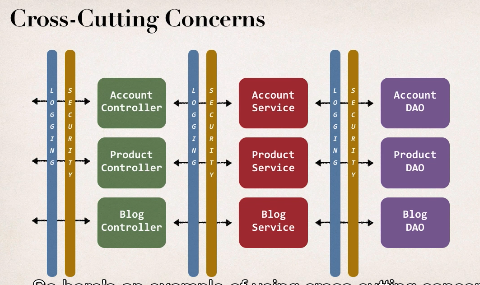
SOlution

AOP :Aspect

1 programming technique based on ASPECT

2 ASPECt encapsulet cross cutting logic :

**cross Cutting concerns :** means logic or functionlity : basic infrastructure that our all applicaion is required



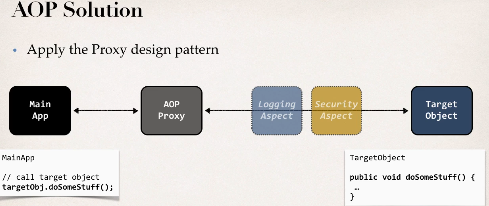
Common logic on respective layers

Aspect

1 can be reused at multiple loaction

2 same Aspect/class applied on base configuration



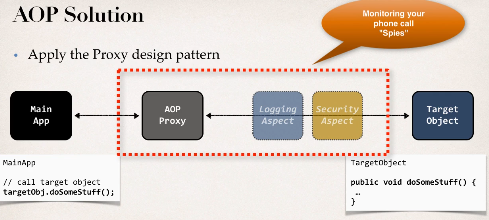


AOP proxy design pattern

- main app calls the target object

-method will be having some logic in target object

-Main app have no idea about AOP and no idea about any aspect behind secnce we dnt know its working will call just method



Befits of AOP

1 code for ASPECT is deisgned in a sinlge class

-much better then being scatrred every where

- Promtes code reuse and easier to change

2 Bussiness code in our application is cleaner

- only applies to bussiness funtionlity: addAccount

-reduce code complexity

3 Configurable

-based on configuration, apply aspects slectively to differt parts of app

-no need to make changes to main application code very important

Additonal use case

Most common: logging , security , tranaction

Audit loggin: who, what when ,where

Exception Handling:Log exception and notify DevOps team via SMS/email

API management:

-how many times has a method been called user

-analytics what are the peak times? Wt is Avg load ? who is top user

Advanteges and Disadvantages

Advantage: reusable modules, resolve code tangling, resolve code scatter, Applied slectively based on confiurtaion

Disadvantage: too many aspects and app flow is hard to follow, minor performance cost for aspect execution(run -tome -weaving)

Comparing SPringAOP and ASPECTJ

**AOP Terminology**

Aspect: module for code for a cross cutting concern(logging , security)

Advice: what actions should be taken and when it should applied

Join Point : when to apply code during program execution

Point Cut: A predict expression for where advice should be applied

**Advice Types**

Before Advice: run before method

After Finally Advice: run after the method (finally)

After returning Advice: run after the method (success excutio)

After Throwing Advice: run after method (if exception thrown)

Around Advice: run before and after method

**Weaving**

**-** connecting aspects to target objects to create an advice object

- differnet types of weaving : compile , run ,load -time

- regarding performance: run time weaving is the slowest,

AOP Frameworks

2 leading Framework SPRINGAOP and ASPECTJ

Spring AOP support

-key components: Security,tranaction ,cacheing etc

- uses run -time weaving aspects : usesr proxy pattern

ASPECTJ

- orinal AOP framwork relased in 2001

- provides compler suppport of AOP

- Rich support for AOP : Join points : method -level,constructor, field

Code weaing: compile time,post compile -time and laod time

SPRIGN AOPComparision

Spring AOP advanatage

- similat to use then ASPECTJ

- use proxy pattern

- can migrate to AspectJ when using @Aspect annotation

Disadvantage

- only support method level join points

- can apply to aspects to beans created by Spring app Context

- minor performance cost for aspect execution(run time weaving)

ASPECTJ Comparision

Advantages:

-Support all join points

-works with many pojo not just bean from app context

- faster compared to Spring AOP

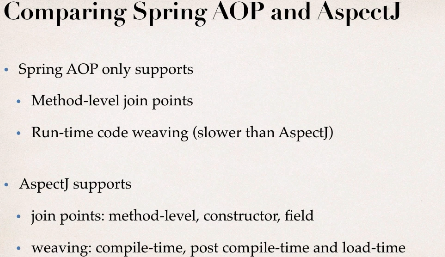
- complet AOP support

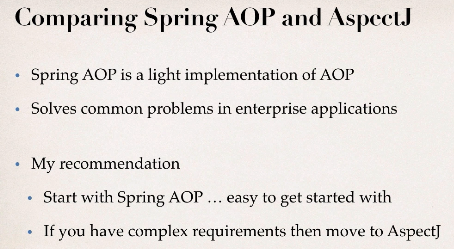
Disadvantage

-Compile time waving requires extra compilation steps

- AspectJ pointCut syntax can become complex

Comparing Both





Road Map

1 create Aspect

2 Develop Advice

- before,After returning ,after throwing

-after finally , Around

3 create point cut expression

4 apply to cRM project

@Before Advice

Usecase:

Most commn:

logging ,security,transaction

Ex @Tranasctional works the same behind scecne called begin and end tranasction

Audit Logging: who,when where

API management

Adding AspetJ jar

Because Spring AOP uses some AspetJ annotaions, and classes

Develeopmetn process

1 createing target Object : ACCount DAO

2 creating simple java config file

3 create main app

4 create an aspect with @Before advice

Best practice of Advice and Aspects

-keep the code small and fast

-dnt perform any expensive or slow operations

- get in and out as sson as possible

Deveolpment

Create Java project add spring jars and aspectj weaver jar

1 @Component

**public** **class** AccountDAO {

**public** **void** addAccoount() {

System.***err***.println(getClass()+ " Doing DB work: addin in account");

}

}

2

// spring pure java configuration

@Configuration

@EnableAspectJAutoProxy //Sring can AOP objects for calling (Spring AOP proxy support)

@ComponentScan("com.spring.demoaop")

**public** **class** DemoConfig {

}

3

**public** **class** MainDemoApp {

**public** **static** **void** main(String[] args) {

//read sring java class

AnnotationConfigApplicationContext context= **new** AnnotationConfigApplicationContext(DemoConfig.**class**);

// get the beans form spring container

AccountDAO accountDAO=context.getBean("accountDAO",AccountDAO.**class**);

// call bussiness method

accountDAO.addAccoount();

System.***err***.println("lets call bussiness method again");

// lets call bussiness method again

accountDAO.addAccoount();

// clsoe context

context.close();

}

}

4

@Aspect

@Component

**public** **class** MyLoggingAspect {

// we had all related advices

// lets starts with @Before advice

// public void addAccoount() ==> point cut expression

// run this code BEFORE- tearget Object method "public void addAccoount()"

@Before("execution(public void addAccoount())")

**public** **void** beforeAddAccountAdvice() {

// custom code

System.***err***.println("=====> execuing @Before advice on addAccount()");

}

}

AOP terminology

1 pointcut: a predicate expression where advice shoul be applied

PointCUT Expression Language:

Spring AOP uses ASPECTJ pointcut expression language

- we will start with execution point cuts

-aaplies to execution methods

Match on Method name:

execution(modifers-pattern? Retrun-type-pattern declaring-type-pattern? Method-name-pattern(param-pattern) throws-patern?)

modifers-pattern: spring AOP only supprots public or \*

Retrun-type-pattern: void,String ,boolean,etc..

declaring-type-pattern: class name

Method-name-pattern: method name/wild card option

Method-name-pattern(param-pattern): method with pararmeter type to match

throws-patern: exception type to match

? Is optional

Pattern can be wild cards for that we can use \* (matches on everything)

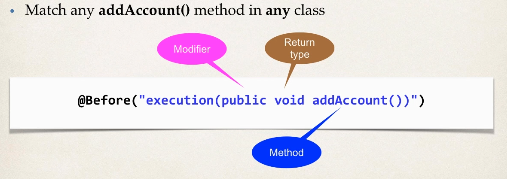
Match on Method names

-match only addAccount() method in AccountDAO class

@Before(“execution(public void com.spring.demoaop.dao.AccountDAO.addAccoount())”)

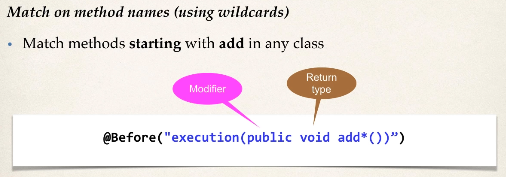


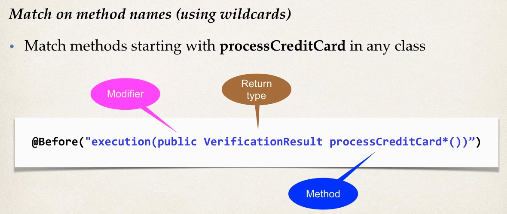
-match addAccount() method in any class



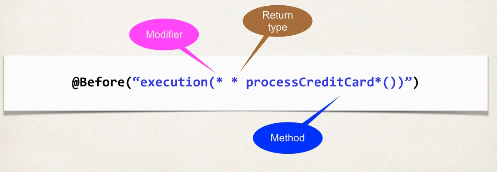
match on methods using wild cards

- match methods statrting with add any class

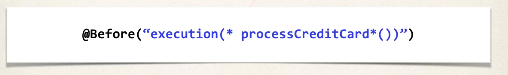




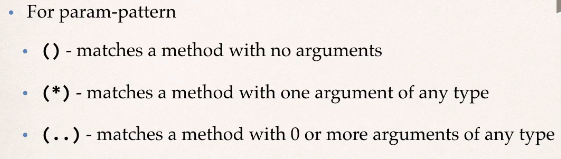
-use wild card entry on modifers and return type



-modifiers optional



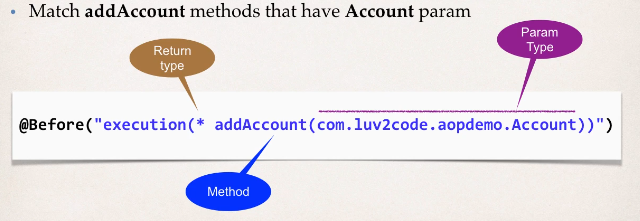
Match methos parameter types

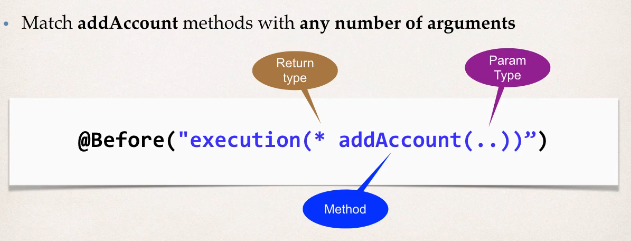


-match addAccount methods with no arguments

@Before("execution(\* add\*())")

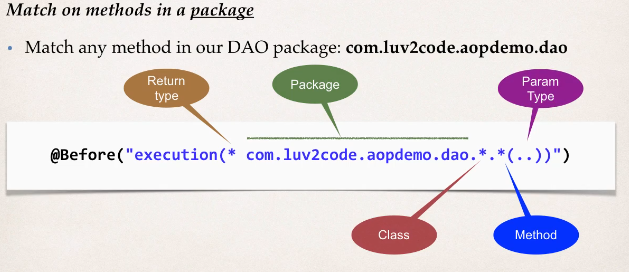
match addAccount methods that have acouunt param





Match on Package

Match any method In DAO pacakge



AOP pointcut Decalrations

Probelm

1 how can we reuse point cut expression

2 want to apply to multiple advices

@Before("execution(\* com.spring.demoaop.dao.\*.\*(..))")

**public** **void** beforeAddAccountAdvice()

Solutions

- copy paste method

@Before("execution(\* com.spring.demoaop.dao.\*.\*(..))")

**public** **void** beforeupdateAccountAdvice()

Ideal solution is

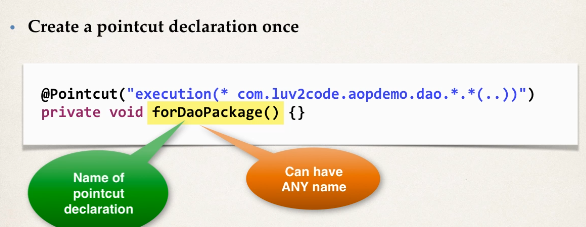
-create a point cut declaration once

- apply it to multiple advices

Development process

1 create a point cut declaration

2 apply point cut decalartion to advices





Benift

- easily reuse point cut expression

-update point cut in one loaction

-can combine point cu expression