Aspect Oriented Programming

with

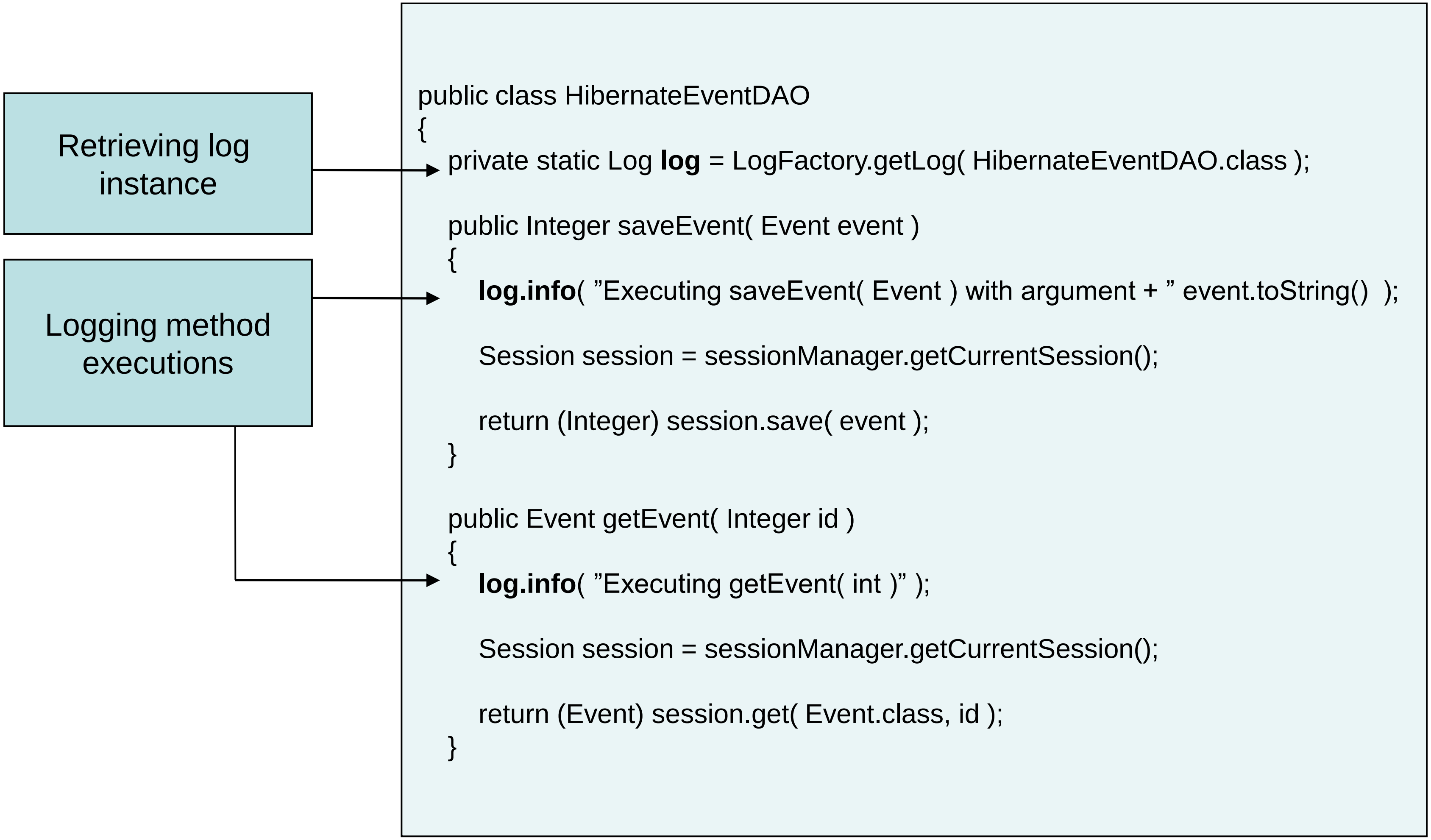
Spring

Problem area

- How to modularize concerns that span multiple classes and layers?

- Examples of *cross-cutting* concerns:
  - Transaction management
  - Logging
  - Profiling
  - Security
  - Internationalisation

Logging: A naive approach

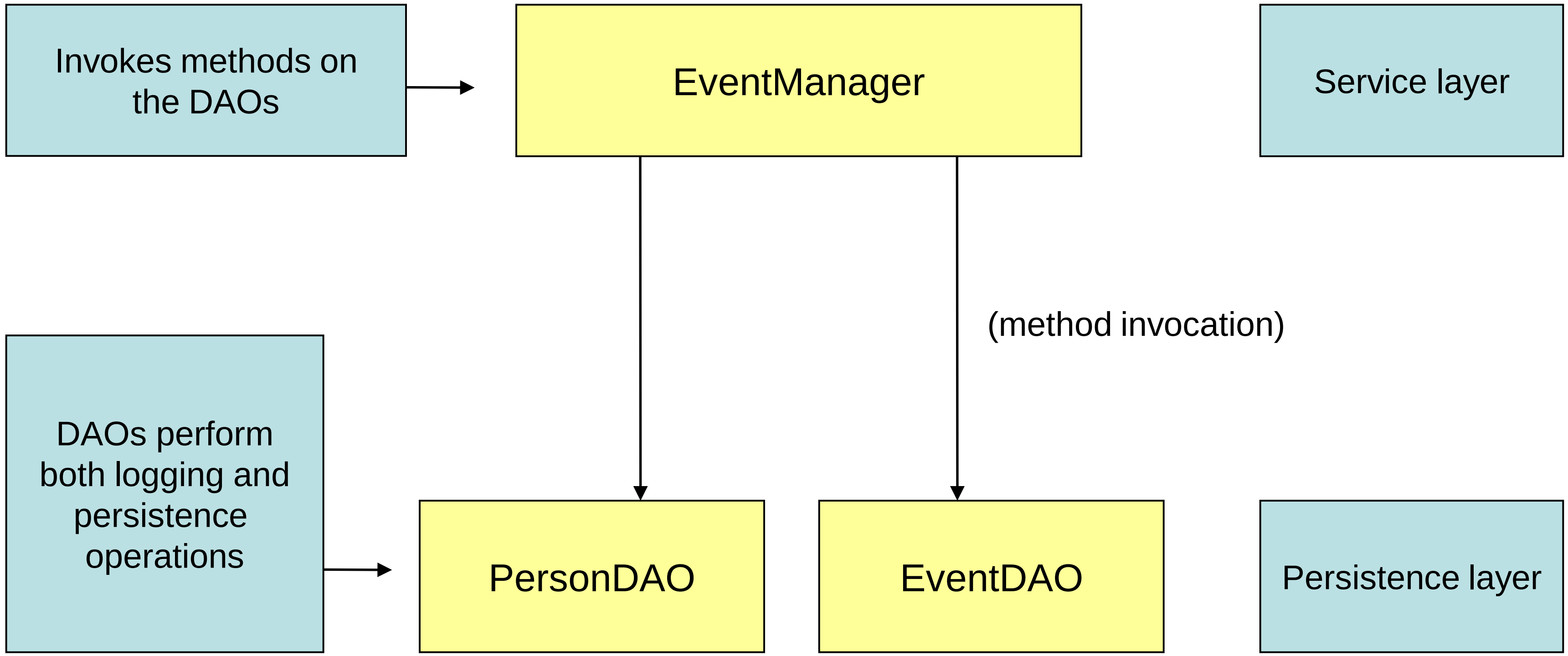


Retrieving log instance

Logging method executions

```
public class HibernateEventDAO
{
   private static Log log = LogFactory.getLog( HibernateEventDAO.class );

   public Integer saveEvent( Event event )
   {
      log.info( ”Executing saveEvent( Event ) with argument + ” event.toString()  );

      Session session = sessionManager.getCurrentSession();

      return (Integer) session.save( event );
   }

   public Event getEvent( Integer id )
   {
      log.info( ”Executing getEvent( int )” );

      Session session = sessionManager.getCurrentSession();

      return (Event) session.get( Event.class, id );
   }
```

Logging: A naive approach



Invokes methods on the DAOs

EventManager

Service layer

(method invocation)

DAOs perform both logging and persistence operations

PersonDAO

EventDAO

Persistence layer

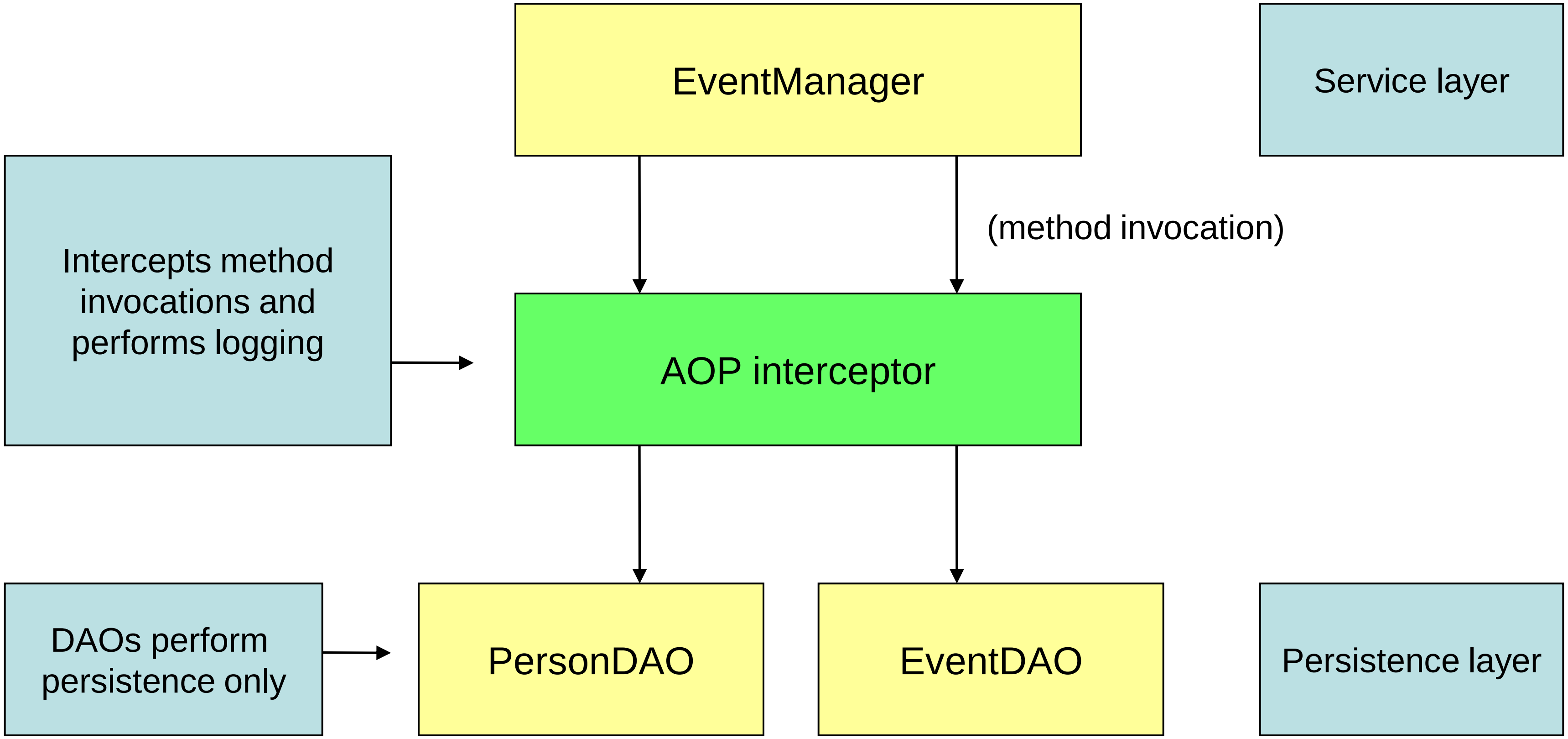
Shortcomings of naive approach

- Mixes persistence and logging functionality
  - Violates the principle of *separation of concerns*
  - Increases complexity and inter-dependency

- Involves repetition of code
  - Violates the *DRY principle*
  - Makes it difficult to change

- Couples the LogFactory to the HibernateEventDAO
  - Prevents *loosely coupled design*
  - Makes change, re-use and testing problematic

Logging: The AOP approach



Advantages of AOP approach

- Separates persistence and logging functionality
  - The logging concern taken care of by the interceptor
  - Makes it easier to understand, manage and debug

- Promotes code reuse and modularization
  - The AOP interceptor is used by all methods in the DAOs
  - Makes it easier to change

- Decouples the LogFactory from the DAO impl’s
  - The HibernateEventDAO is unaware of being logged
  - Makes change, re-use and testing simple

Aspect Oriented Programming

- Definition: Enables encapsulation of functionality that affects multiple classes in separate units

- Complements object oriented programming

- Most popular implementation for Java is *AspectJ*
  - Aspect oriented extension for Java
  - Based on Eclipse, available as plugin and stand-alone

AOP with Spring

- The *AOP framework* is a key component of Spring
  - Provides declarative enterprise services (transactions)
  - Allows for custom aspects
- Aims at providing integration between AOP and IoC
- Integrates – but doesn’t compete – with AspectJ
- Provides two techniques for defining aspects:
  - @AspectJ annotation
  - XML schema-based

AOP concepts

- Aspect
  - A *concern* that cuts across multiple classes and layers
- Join point
  - A *method invocation* during the execution of a program
- Advice
  - An implementation of a concern represented as an *interceptor*
- Pointcut
  - An *expression* mapped to a join point

@AspectJ support

- Style of declaring aspects as regular Java classes with Java 5 annotations
- Requires *aspectjweaver* and *aspectjrt* on the classpath
- Enabled by including the following information in the Spring configuration file:

```
<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-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <aop:aspectj-autoproxy/>
```

Declaring an aspect

- A *concern* that cuts across multiple classses and layers

@Aspect annotation

Any bean with a class annotated as an aspect will be automatically detected by Spring

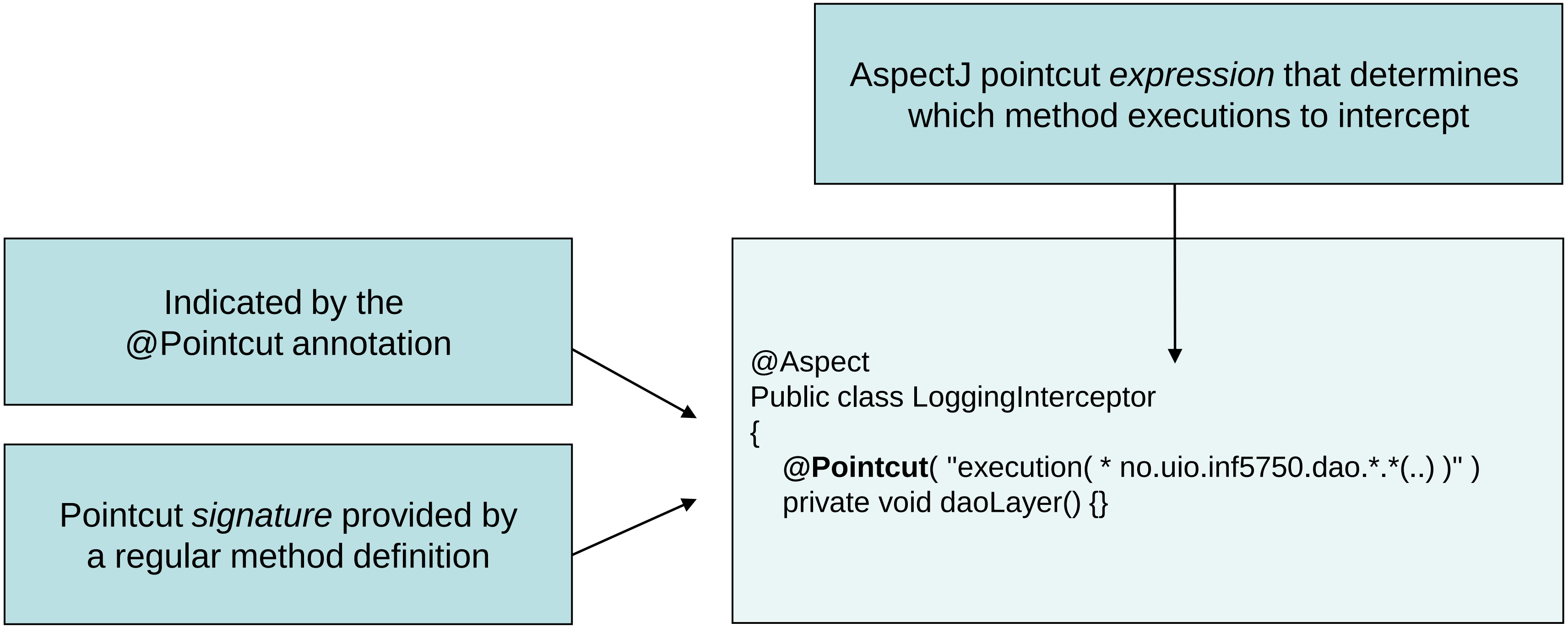
```
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class LoggingInterceptor
{
    // ...
}
```

Regular bean definition pointing to a bean class with the @Aspect annotation

```
<bean id="loggingInterceptor"
    class="no.uio.inf5750.interceptor.LoggingInterceptor"/>
```

Declaring a pointcut

- An *expression* mapped to a *join point* (method invocation)



AspectJ pointcut *expression* that determines which method executions to intercept

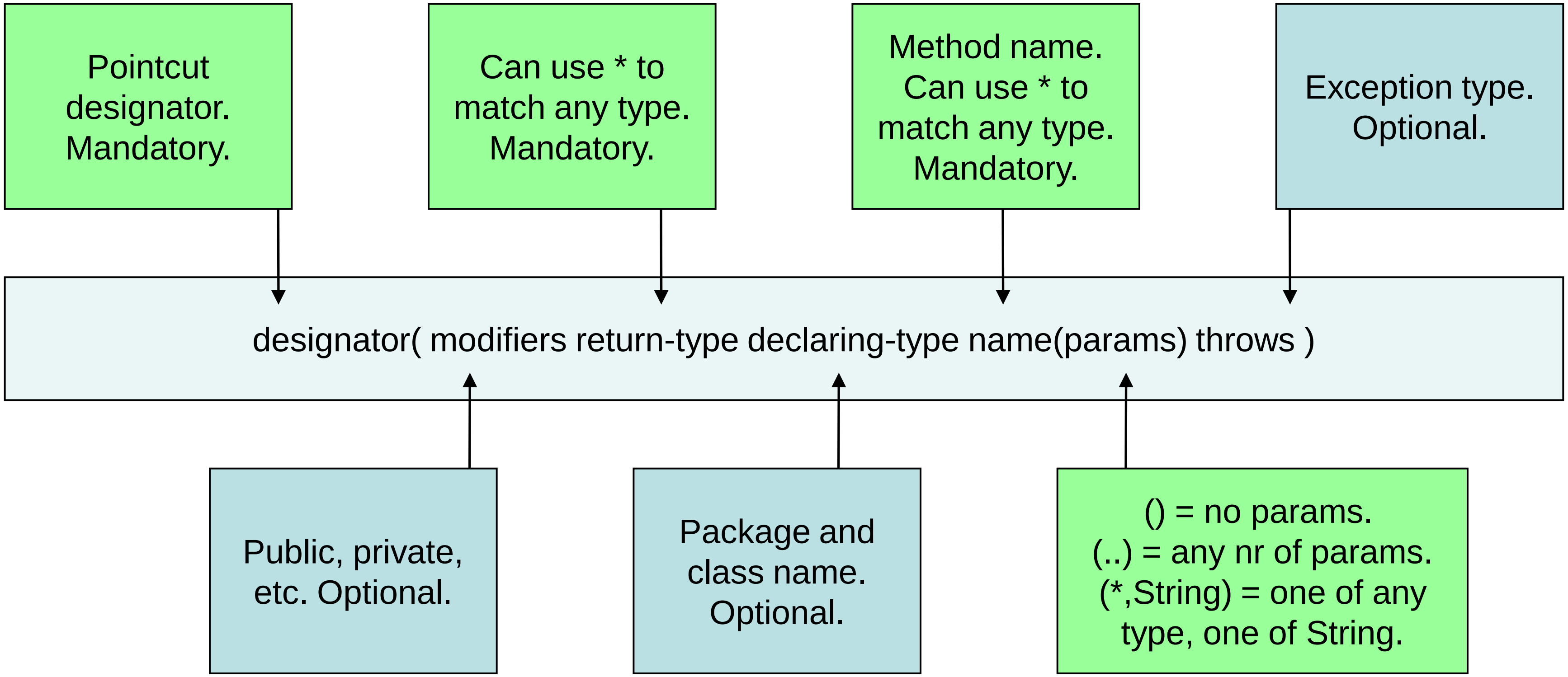
Indicated by the @Pointcut annotation

Pointcut *signature* provided by a regular method definition

```
@Aspect
Public class LoggingInterceptor
{
   @Pointcut( "execution( * no.uio.inf5750.dao.*.*(..) )" )
   private void daoLayer() {}
```

Pointcut expression pattern

- The *execution* pointcut designator is used most often



Pointcut designator. Mandatory.

Can use * to match any type. Mandatory.

Method name. Can use * to match any type. Mandatory.

Exception type. Optional.

designator(modifiers return-type declaring-type name(params) throws)

Public, private, etc. Optional.

Package and class name. Optional.

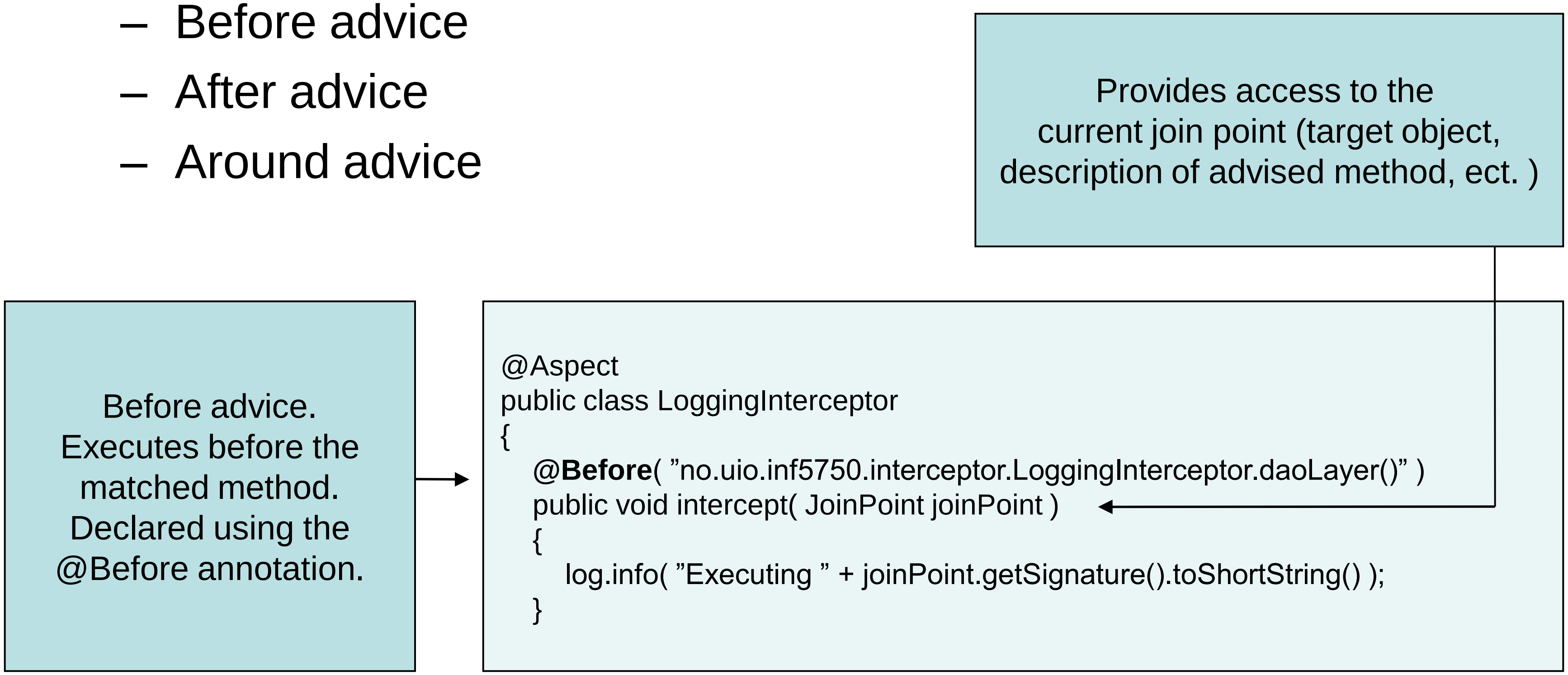
() = no params.
(..) = any nr of params.
(*,String) = one of any type, one of String.

Pointcut expression examples

Any public method	execution(public * *(..))
Any public method defined in the dao package	execution(public * no.uio.inf5750.dao.*.*(..))
Any method with a name beginning with *save*	execution(* save*(..))
Any method defined by the EventDAO interface with one param	execution(* no.uio.inf5750.dao.EventDAO.*(*))

Declaring advice

- Implementation of concern represented as an *interceptor*
- Types
  - Before advice
  - After advice
  - Around advice



Provides access to the current join point (target object, description of advised method, ect.)

Before advice. Executes before the matched method. Declared using the @Before annotation.

```
@Aspect
public class LoggingInterceptor
{
    @Before( ”no.uio.inf5750.interceptor.LoggingInterceptor.daoLayer()” )
    public void intercept( JoinPoint joinPoint )
    {
        log.info( ”Executing ” + joinPoint.getSignature().toShortString() );
    }
```

After returning & throwing advice

After returning advice. Executes after the matched method has returned normally. Declared using the @AfterReturning annotation.

```
@Aspect
public class LoggingInterceptor
{
    @AfterReturning( ”no.uio.inf5750.interceptor.LoggingInterceptor.daoLayer()” )
    public void intercept( JoinPoint joinPoint )
    {
        log.info( ”Executed successfully ” + joinPoint.getSignature().toShortString() );
    }
```

After throwing advice. Executes after the matched method has thrown an exception. Declared using @AfterThrowing.

```
@Aspect
public class LoggingInterceptor
{
    @AfterThrowing( ”no.uio.inf5750.interceptor.LoggingInterceptor.daoLayer()” )
    public void intercept( JoinPoint joinPoint )
    {
        log.info( ”Execution failed ” + joinPoint.getSignature().toShortString() );
    }
```

Around advice

- Can do work both before and after the method executes
- Determines when, how and if the method is executed

Around advice.

The first parameter must be of type ProceedingJoinPoint – calling proceed() causes the target method to execute.

Declared using the @Around annotation.

```
@Aspect
public class LoggingInterceptor
{
    @Around( ”no.uio.inf5750.interceptor.LoggingInterceptor.daoLayer()” )
    public void intercept( ProceedingJoinPoint joinPoint )
    {
        log.info( ”Executing ” + joinPoint.getSignature().toShortString() );

        try
        {
            joinPoint.proceed();
        }
        catch ( Throwable t )
        {
            log.error( t.getMessage() + ”: ” + joinPoint.getSignature().toShortString() );
            throw t;
        }

        log.info( ”Successfully executed ” + joinPoint.getSignature().toShortString() );
}
```

Accessing arguments

- The *args binding form* makes argument values available to the advice body
- Argument name must correspond with advice method signature

Makes the object argument available to the advice body

Will restrict matching to methods declaring at least one parameter

```
@Aspect
public class LoggingInterceptor
{
   @Before( ”no.uio.inf5750.interceptor.LoggingInterceptor.daoLayer() and ” +
      ”args( object, .. )” )
   public void intercept( JoinPoint joinPoint, Object object )
   {
      log.info( ”Executing ” + joinPoint.getSignature().toShortString() +
         ” with argument ” + object.toString() );
   }
```

Accessing return values

- The *returning binding form* makes the return value available to the advice body
- Return value name must correspond with advice method signature

Makes the object return value available to the advice body

Will restrict matching to methods returning a value of specified type

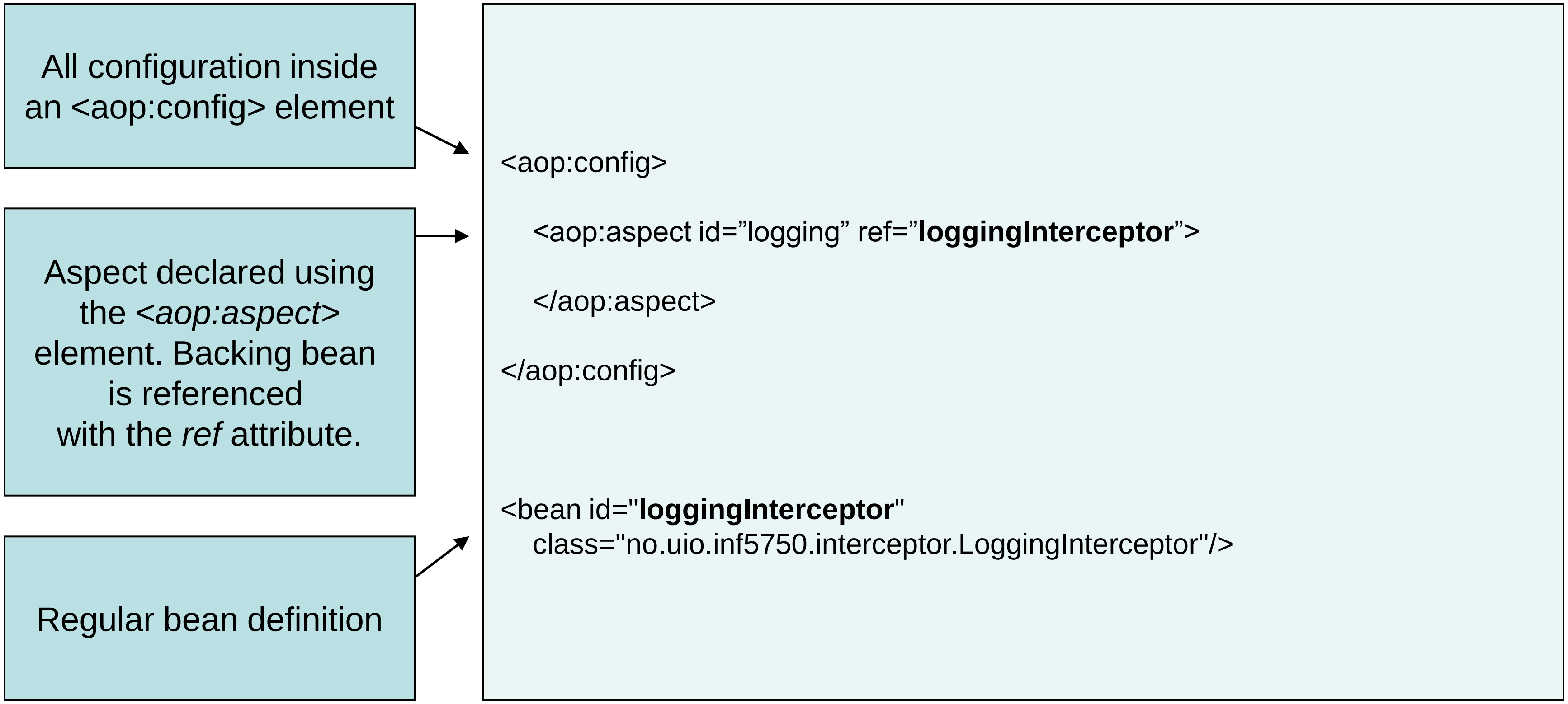
```
@Aspect
public class LoggingInterceptor
{
    @AfterReturning(
        pointcut=”no.uio.inf5750.interceptor.LoggingInterceptor.daoLayer() ”,
        returning=”object” )
    public void intercept( JoinPoint joinPoint, Object object )
    {
        log.info( ”Executed ” + joinPoint.getSignature().toShortString() +
            ” with return value ” + object.toString() );
    }
```

Schema-based support

- Lets you define aspects using the *aop namespace* tags in the Spring configuration file
- Enabled by importing the Spring aop schema
- Pointcut expressions and advice types similar to @AspectJ
- Suitable when:
  - You are unable to use Java 5
  - Prefer an XML based format
  - You need multiple joinpoints for an advice

Declaring an aspect

- An aspect is a regular Java object defined as a bean in the Spring context



All configuration inside an <aop:config> element

Aspect declared using the <*aop:aspect*> element. Backing bean is referenced with the *ref* attribute.

Regular bean definition

```
<aop:config>

    <aop:aspect id=”logging” ref=”loggingInterceptor”>

    </aop:aspect>

</aop:config>



<bean id="loggingInterceptor"
    class="no.uio.inf5750.interceptor.LoggingInterceptor"/>
```

Declaring a pointcut

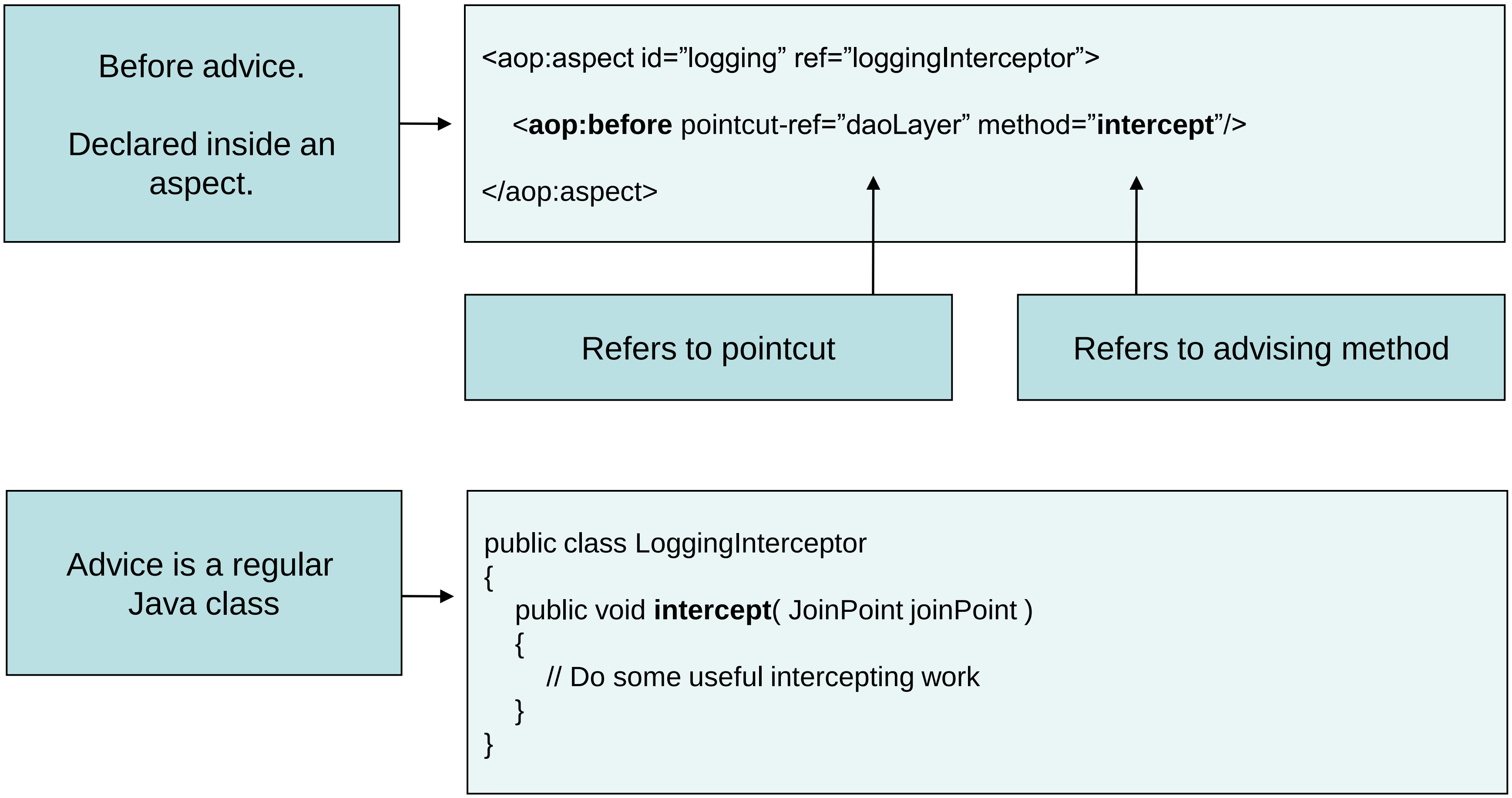
- Pointcut expressions are similar to @AspectJ
- A pointcut can be shared across advice

Pointcut declared inside <aop:config> element using the <aop:pointcut> element

Can also be defined inside aspects

```
<aop:config>

    <aop:pointcut id=”daoLayer”
        expression="execution( * no.uio.inf5750.dao.*.*(..) )”/>

</aop:config>
```

Declaring advice



Before advice.

Declared inside an aspect.

```
<aop:aspect id=”logging” ref=”loggingInterceptor”>

    <aop:before pointcut-ref=”daoLayer” method=”intercept”/>

</aop:aspect>
```

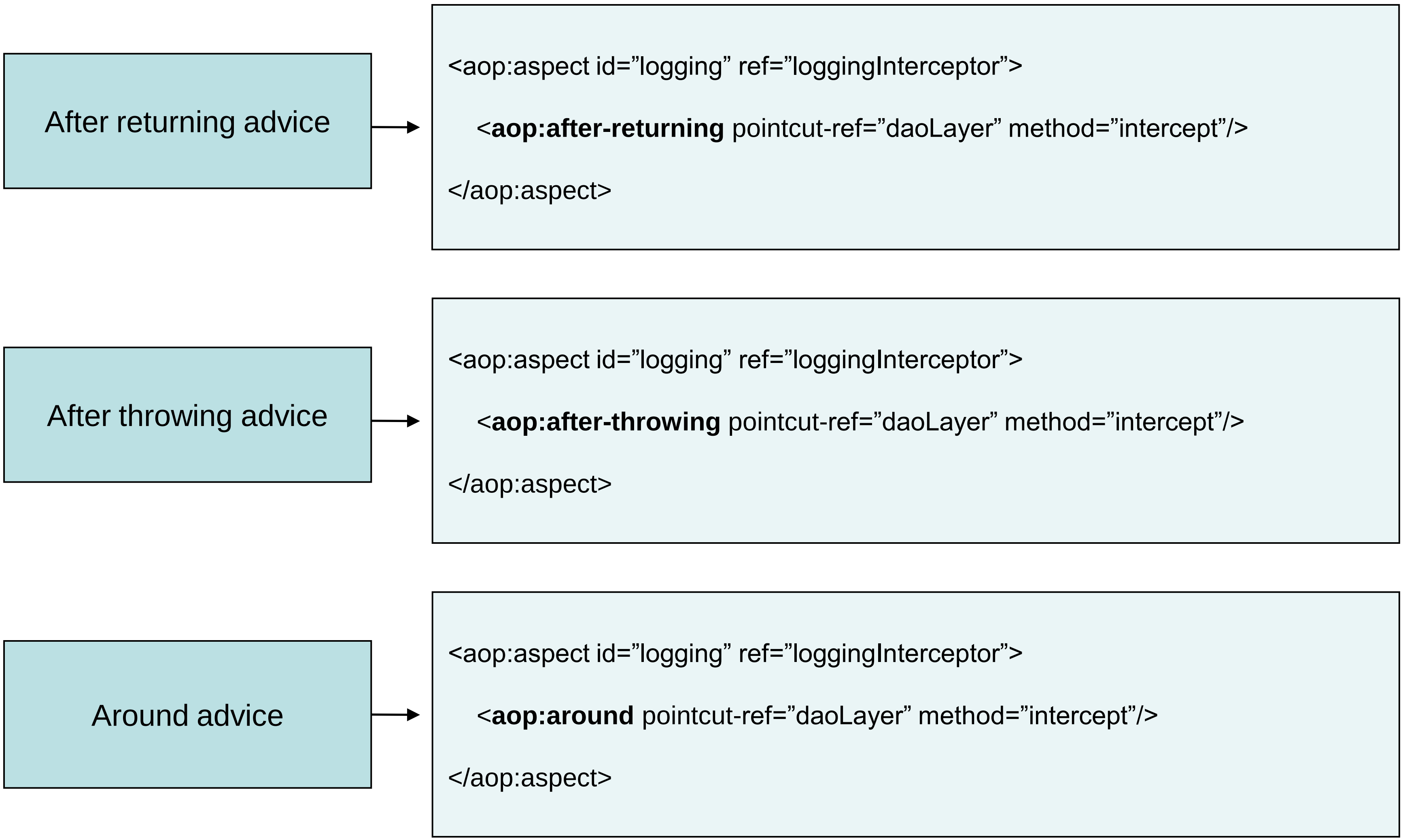
Refers to pointcut

Refers to advising method

Advice is a regular Java class

```
public class LoggingInterceptor
{
    public void intercept( JoinPoint joinPoint )
    {
        // Do some useful intercepting work
    }
}
```

Declaring advice



After returning advice

```
<aop:aspect id=”logging” ref=”loggingInterceptor”>

   <aop:after-returning pointcut-ref=”daoLayer” method=”intercept”/>

</aop:aspect>
```

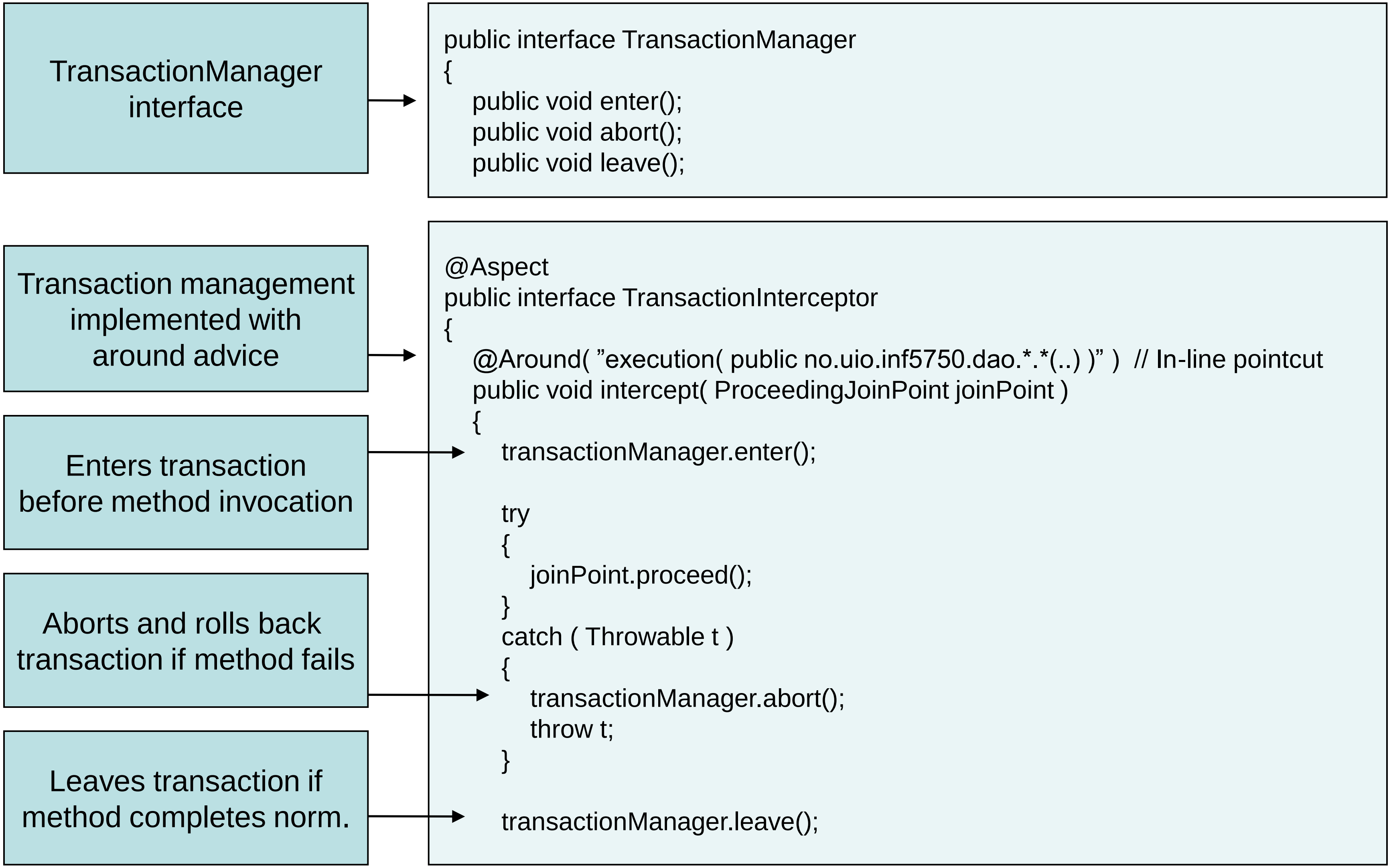
After throwing advice

```
<aop:aspect id=”logging” ref=”loggingInterceptor”>

   <aop:after-throwing pointcut-ref=”daoLayer” method=”intercept”/>

</aop:aspect>
```

Around advice

```
<aop:aspect id=”logging” ref=”loggingInterceptor”>

   <aop:around pointcut-ref=”daoLayer” method=”intercept”/>

</aop:aspect>
```

AOP - Transaction Management



TransactionManager interface

```
public interface TransactionManager
{
    public void enter();
    public void abort();
    public void leave();
```

Transaction management implemented with around advice

Enters transaction before method invocation

Aborts and rolls back transaction if method fails

Leaves transaction if method completes norm.

```
@Aspect
public interface TransactionInterceptor
{
    @Around( ”execution( public no.uio.inf5750.dao.*.*(..) )” )  // In-line pointcut
    public void intercept( ProceedingJoinPoint joinPoint )
    {
        transactionManager.enter();

        try
        {
            joinPoint.proceed();
        }
        catch ( Throwable t )
        {
            transactionManager.abort();
            throw t;
        }

        transactionManager.leave();
```

@AspectJ or Schema-based?

- Advantages of schema style
  - Can be used with any JDK level
  - Clearer which aspects are present in the system

- Advantages of @AspectJ style
  - One single unit where information is encapsulated for an aspect
  - Can be understood by AspectJ – easy to migrate later

Summary

- Key components in AOP are *aspect*, *pointcut*, *join point*, and *advice*

- AOP lets you encapsulate functionality that affects multiple classes in an *interceptor*

- Advantages of AOP:
  - Promotes separation of concern
  - Promotes code reuse and modularization
  - Promotes loosely coupled design

References

- The Spring reference documentation - Chapter 8
  - www.springframework.org

- AOP example code
  - www.ifi.uio.no/INF5750/h07/undervisningsplan.xml