Aspect Oriented Programming

2013-2014

Course 11

Course 11 Contents

- AOP Design Patterns

AOP Design Patterns

- Design patterns define solutions to recurring problems.
- A pattern has four essential elements:
  - The pattern name
  - The problem
  - The solution
  - The consequences (results, trade-offs)
- AOP design patterns can help you define solutions when you start applying AOP:
  - the worker object pattern,
  - the wormhole pattern,
  - the participant pattern,
  - the annotation-driven participant pattern.

The worker object pattern

- A *worker object* is an instance of a class that encapsulates a method (called a *worker method*).
- A worker object can be passed around, stored, and invoked.
- In Java, a common implementation of a worker class implements `Runnable`, where the `run()` method invokes the worker method. When you execute such an object, it in turn executes the worker method.
- The worker object pattern offers a way to generate worker objects automatically for join points selected by a pointcut.
- These objects can then be passed around to implement various functionalities.
- Using this pattern, you ensure a consistent behavior in the system and save a lot of code.

The current solution (without pattern)

- Two chief tasks have to be addressed for each method involved:
  - Implement a class that invokes the work method and creates an object of that class.
  - Use that object in place of the original method.

```
Runnable worker = new Runnable() {
   public void run() {
   ... invoke the work method
   }
}
```

- Use the worker object to:

  - send the worker object to a queue for execution,

  - or pass it to another subsystem for execution,

  - or simply call `run()` directly

The current solution (without pattern)

- Depending on the situation, you may use either named or anonymous classes.
- In either case, you implement an interface such as `Runnable` that contains a `run()` method for invoking the work method.
- With either style, you need to replace the normal method call with the creation of a worker object and invoke that object.
- The amount of additional code makes implementation a difficult task.
- It also increases the risk of missing certain changes, which may result in undesirable system behavior.

Pattern overview

- You use an aspect to create objects of anonymous classes (which are the worker objects) automatically.
- You write a pointcut selecting all the join points that need routing through a worker object, and an advice that executes the selected join point inside the `run()` method of the anonymous worker class.
- Normally, when `proceed()` is called directly from within around advice, it executes the selected join point.
- But with the worker object pattern, you create a worker object, have its `run()` method call `proceed()`, and invoke the `run()` method later, perhaps in another thread, to execute the selected join point.

Pattern template

- You write a pointcut selecting the needed join points. You may/may not collect any context in this pointcut (depends on the situation).
- You advise this pointcut with an around advice to create and use the worker object:

```
void around() : <pointcut> {
   Runnable worker = new Runnable () {
        public void run() {
                proceed();
        }
   }
   //... use the worker object
}
```

Pattern example

- Saving a backup copy of a project may be an expensive operation that can be better executed in a separate thread.

```
package ajia.example;
public class ProjectSaver {
   public static void backupSave() {
        System.out.println("Saving backup copy in thread "
                + Thread.currentThread());
   }
}
```

Pattern example

- The aspect **`AsynchronousExecutionAspect`** defines a pointcut that selects operations to be routed asynchronously.

```
package ajia.example;
import ...
public aspect AsynchronousExecutionAspect {
   private Executor executor = Executors.newCachedThreadPool();
public pointcut asyncOperation():execution(*
   ProjectSaver.backupSave())
   /* || ... */;
   void around() : asyncOperation() {
        Runnable worker = new Runnable() {
                public void run() {
                        proceed();
                }
        };
        executor.execute(worker);
   }
}
```

Pattern example

- In the aspect, the **`asyncOperation()`** pointcut selects join points to be routed asynchronously.
- The around advice creates an object—worker—of an anonymous class that implements the **`Runnable`** interface.
- In the **`run()`** method, it calls **`proceed()`** to execute the advised join point.
- Because **`worker`** performs the operation selected by the advised join point, it is the worker object.
- The advice executes the work by passing the worker object to the **`execute()`** method of the executor object.

Getting the return value

- Some of the routed calls could be returning a value to the caller.
- In that case, `proceed()` returns the value of the method when the operation has completed.
- You can keep this value in the worker object and later return it from the around advice.
- For the value to make sense, the caller must wait until the execution of the worker object finishes.

```
public abstract class RunnableWithReturn implements Runnable {
   private Object returnValue;
   public Object getReturnValue() {
        return this.returnValue;
   }
   public void setReturnValue(Object returnValue) {
        this.returnValue = returnValue;
   }
}
```

Getting the return value

```
package ajia.example;
import ...
public aspect SynchronousExecutionAspect {
   public pointcut syncOperation():call(* /*some class*/.*(..));
   Object around() : syncOperation() {
        RunnableWithReturn worker = new RunnableWithReturn() {
          public void run() {
                setReturnValue(proceed());
        }};
        System.out.println("About to run " + worker);
        worker.run();
        return worker.getReturnValue();
   }
}
```

Usually, you pass the worker object to another thread and wait for the execution of the worker.

Getting the return value

- Pattern usages:
  - implementing thread safety in Swing applications,
  - improving the responsiveness of UI applications,
  - performing authorization and transaction management.

The wormhole pattern

- The wormhole pattern makes context information from a caller available to a callee—without having to pass the information as a set of parameters to each method in the control flow.
- Eg: an authorization system, where many methods need to know who invoked them to determine if the caller should be allowed to execute the operation.
- The wormhole allows the methods to access the caller object and its context to obtain such information.
- By creating a direct route between two levels in the call stack, you create a wormhole and avoid linearly traveling through each layer.
- This saves you from having to modify the call chain when you want to pass additional context information, and it prevents API pollution.

The current solutions

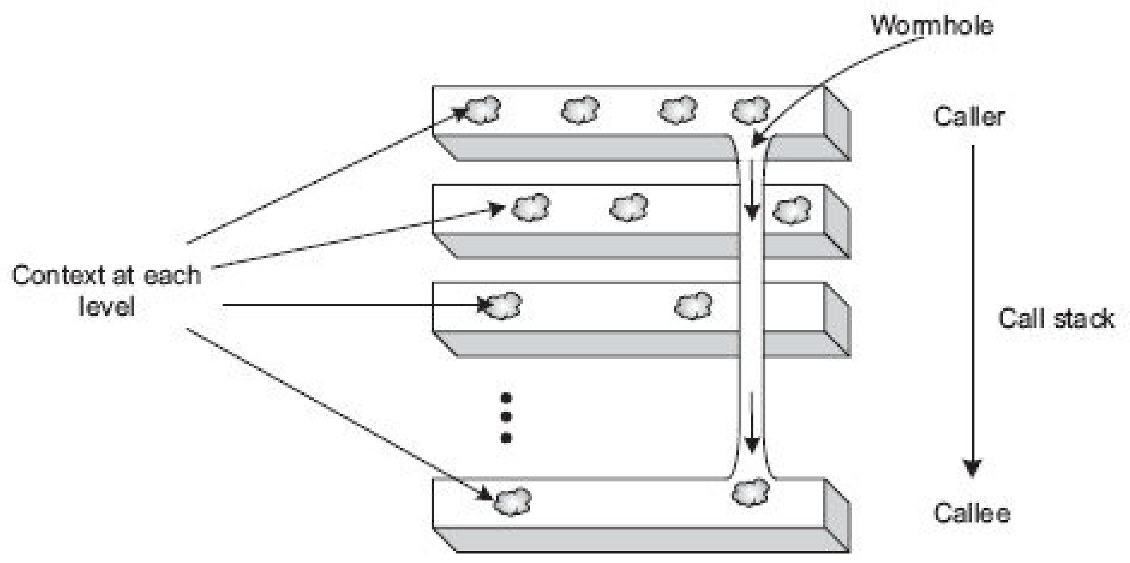
- Without AspectJ, there are two ways to pass the caller's context in a multithreaded environment:
  - Pass the context as additional parameters to methods, which subsequently pass it on. This continues until the context reaches the place of its use. It causes API pollution—every method in the execution stack must have extra parameters to pass on the context collected.
  - Use thread-local storage to set and access the context information. It requires the caller to create a `ThreadLocal` variable to store the context information. Although this approach avoids API pollution, it entails changes in both caller and callee implementation and requires an understanding of how the context is stored. You must also clear the context correctly.
- In either case, multiple modules are involved in the logic to manage the context.

Pattern overview

- The basic idea behind the wormhole pattern is to specify two pointcuts: one for the caller and the other for the callee, with the former collecting the context to be transported through the wormhole.
- Then, you specify the wormhole as the execution of the callee’s join points in the control flow of a caller’s join points.
- To transport context from one level to another, you would normally have to pass it to the next level until it reached the desired location. The wormhole pattern provides a path that cuts directly through the levels, which avoids having the context trickle through the levels from caller to callee.

Pattern overview

- Each horizontal bar shows a level in the call stack.
- The wormhole makes the object in the caller plane available to the methods in the called plane without passing the object through the call stack.



The wormhole pattern template

- You define a pointcut in the caller's space that collects the associated context.
- You define a pointcut in the callee's space.
- The collected context in both cases could be an execution, a target, arguments to the callee join point, or any associated annotations.
- You then create a wormhole through these two spaces using a pointcut that selects the join points selected by the **`calleeSpace`**() pointcut in the control flow of the join points selected by the **`callerSpace`**() pointcut.
- Because you have the context available for both of these join points, you can write advice to the **`wormhole`**() pointcut using this information.

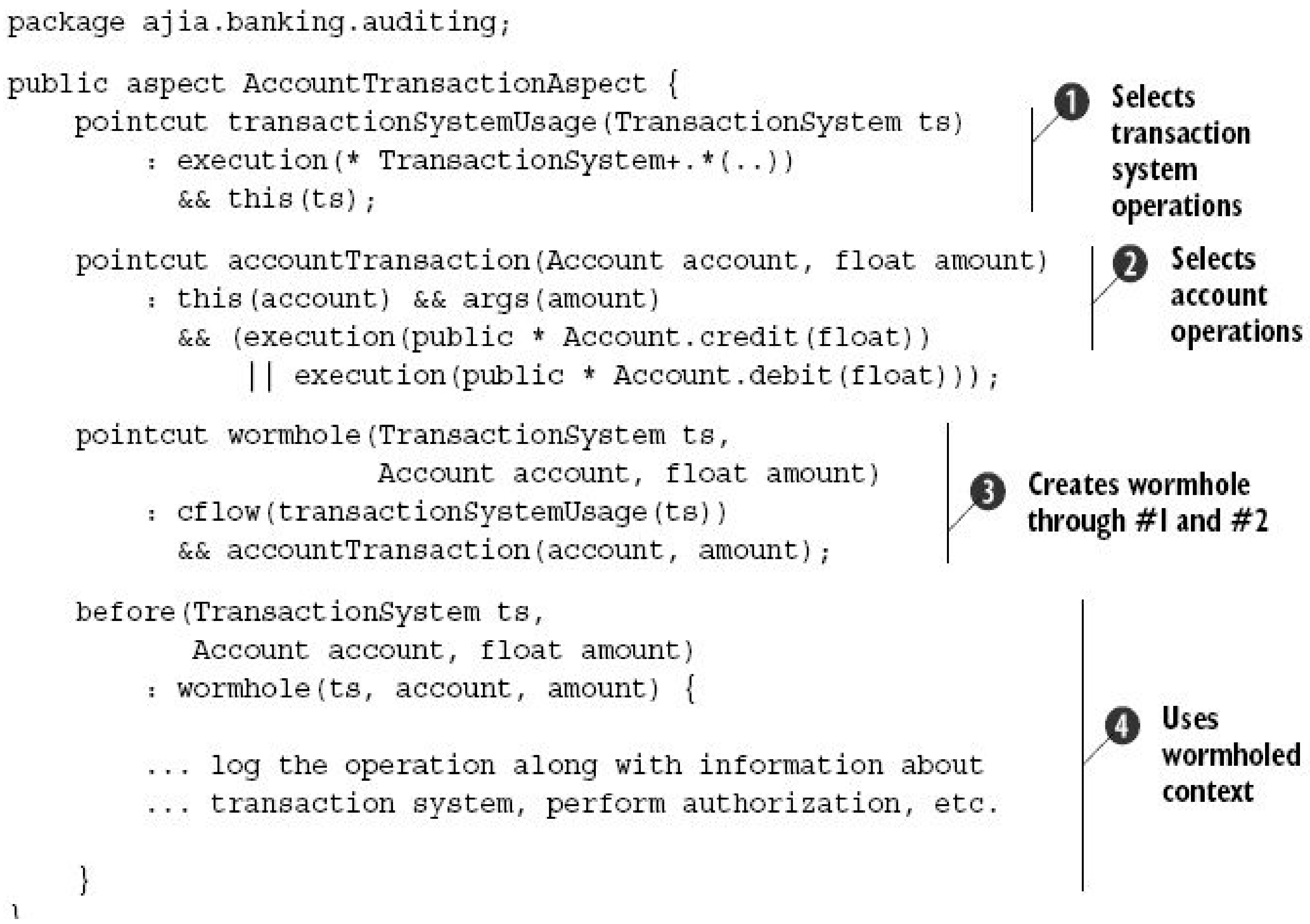
The wormhole pattern template

```
public aspect WormholeAspect {
   pointcut callerSpace(<caller context>): <caller pointcut>;
   pointcut calleeSpace(<callee context>): <callee pointcut>;
   pointcut wormhole(<caller context>, <callee context>)
        : cflow(callerSpace(<caller context>))
        && calleeSpace(<callee context>);
   // advice to wormhole
   before(<caller context>, <callee context>): wormhole(<caller
        context>, <callee context>) {
   ... advice body
   }
}
```

Remark :

Due to the use of the `cflow`() pointcut, this wormhole solution works only with AspectJ weaving. With Spring AOP, you may get a similar effect using **`ThreadLocals`**.

Wormhole pattern example



```
package ajia.banking.auditing;

public aspect AccountTransactionAspect {
    pointcut transactionSystemUsage(TransactionSystem ts)
        : execution(* TransactionSystem+.*(..))
          && this(ts);

    pointcut accountTransaction(Account account, float amount)
        : this(account) && args(amount)
          && (execution(public * Account.credit(float))
              || execution(public * Account.debit(float)));

    pointcut wormhole(TransactionSystem ts,
                      Account account, float amount)
        : cflow(transactionSystemUsage(ts))
          && accountTransaction(account, amount);

    before(TransactionSystem ts,
           Account account, float amount)
        : wormhole(ts, account, amount) {

        ... log the operation along with information about
        ... transaction system, perform authorization, etc.

    }
}
```

1. Selects transaction system operations
2. Selects account operations
3. Creates wormhole through #1 and #2
4. Uses wormholed context

Wormhole pattern example

- The `transactionSystemUsage()` pointcut selects execution join points in a `TransactionSystem` class or its subtypes. It collects the `this` object as the context that you want to pass through the wormhole.
- The `accountTransaction()` pointcut selects the execution of the `credit()` and `debit()` methods in the `Account` class. It collects the account and the amount as the context.
- The `wormhole`() pointcut creates a wormhole between the transaction system and the account operation by selecting join points that match `accountTransaction()` that occur in the control flow of `transactionSystemUsage()`. The pointcut also collects the context selected by the constituent pointcuts, so that advice to it may use it.
- The advice to the `wormhole()` pointcut can now use the context. The advice knows not only the account and the amount but also the transaction system responsible for causing the account activity.

The participant pattern

- Selecting join points with common characteristics across a system is essential in ensuring consistent system behavior.
- The common AOP approach for accomplishing this is to define pointcuts based on the type and name patterns.
- But you often cannot select all the join points sharing a similar characteristic just by using the type and name patterns.
- Many crosscutting concerns, such as transaction management and authentication, tend to span operations that cannot be selected by pointcuts relying solely on type and name patterns.
- Developers usually assign the name of a method based on its core operation. Therefore, the method’s name does not reflect the method’s peripheral characteristics.

The participant pattern

- The participant pattern helps select the methods based on their characteristics and requires the participating classes to collaborate with the aspects explicitly.

REMARK: the participant pattern and its variations require modifications to the core classes, and there is a possibility that you may not identify all the operations.

- You should use the regular pointcuts to the fullest extent possible.

Current solutions

- Two solutions for characteristics-based crosscutting using AOP:
  - A simple technique that allows you to advise join points that share certain characteristics
  - An improvement that makes it easier to maintain.

Current solutions

- Solution 1:
  - The aspect defines a pointcut that lists all the methods that have the desired characteristics (eg. being slow).
  - The problem with this approach is that tight coupling exists between the aspect, all the classes, and the methods in the list. If a new class is added to the system containing a method with the same characteristics, it will not be advised until it is added to the list.
  - Similarly, if a method that is originally part of the list changes its implementation so that it is no longer slow, it will continue to be advised until the aspect is changed to remove the method from the pointcut definition.
  - Both the classes and aspects need to be aware of the existence of each other to remain coordinated.
  - There is a fragile pointcut definition.

Current solutions

```
package ajia.example;

public aspect WaitCursorManagementAspect {
    public pointcut slowOperation() :
            execution(* Analytics.monteCarloSimulation())
        || execution(* Analytics.updateDatabase())
        || execution(* Document.save())
        || execution(* Document.backupSave())
        /* || ... */;

    before() : slowOperation() {
        WaitCursor.set();
    }

    after() : slowOperation() {
        WaitCursor.unset();
    }
}
```

Current solutions

- Solution 2:
  - A class can include pointcuts (but not advice). Because a class knows about the characteristics of its methods, it can specify pointcuts identifying those characteristics.
  - Then you can write an aspect to advise both slowOperation() pointcuts defined in Analytics and Document.
  - This version is better than the earlier solution. Instead of being aware of classes and methods, the aspect is now aware of only the classes because it uses the pointcuts defined in them to select the methods.
  - Nevertheless, the need to be explicitly aware of all the classes doesn't make it an optimal solution. If a new class is added to the system, the aspect won't advise the new class until you add it to the aspect.

Remark:

It is not correct to specify a pointcut such as *.slowOperation(). You must explicitly enumerate all the pointcuts you want to advise from each class.

Current solutions

```
public class Analytics {

    // Analytics' implementation

    public pointcut slowOperation() :
        execution(* Analytics.monteCarloSimulation())
        || execution(* Analytics.updateDatabase());
}

public class Document {

    // Document's implementation

    public pointcut slowOperation() :
        execution(* Document.save())
        || execution(* Document.backupSave());
}

public aspect WaitCursorManagementAspect {
    public pointcut slowOperation() :
            Analytics.slowOperation()
        || Document.slowOperation();

    // advice code
}
```

Pattern overview

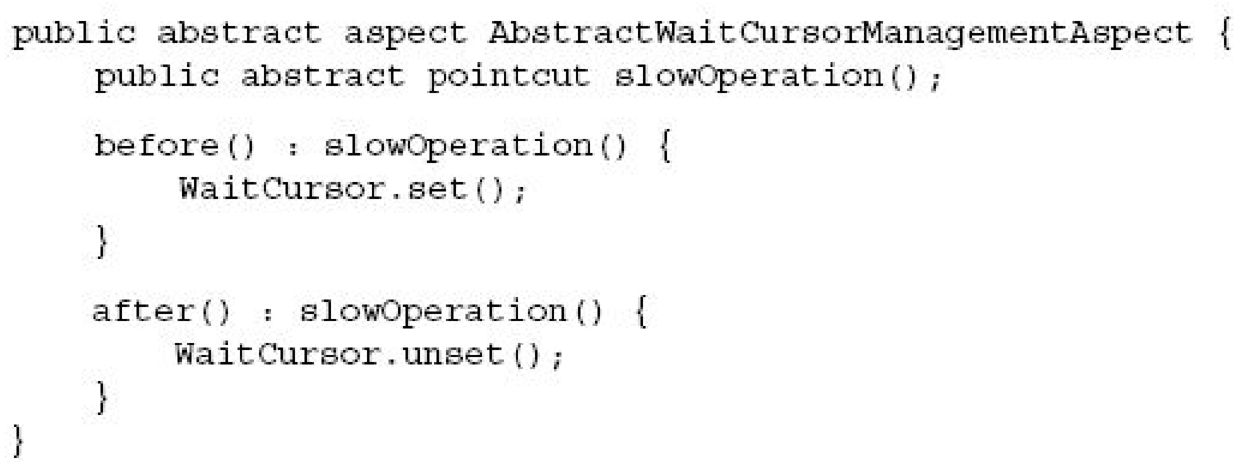
- The participant pattern builds on the idea of classes that contain a pointcut denoting certain characteristics.
- Instead of including a pointcut definition directly inside each class and using those pointcuts in an aspect, the classes define a subaspect that extends the advising aspect and provides the pointcut definition.
- In a way, this pattern reverses the roles—instead of making aspects aware of classes, it makes the classes aware of the aspects.

Pattern structural overview

- You write an abstract aspect that contains abstract pointcut(s) denoting join points with the desired characteristics. These pointcuts form a kind of aspectual interface.
- The aspect also advises each pointcut (or a combination of them) with the required behavior. An inviting aspect—it invites others to participate in the functionality it offers.
- Each class that wants to participate in such a behavior includes a concrete subaspect extending the abstract inviting aspect. Each of these subaspects provides the implementation of the abstract pointcut for the enclosing class.
- The concrete subaspects do not have to be nested aspects of the class—they can be peer aspects.
- This way, each class that wants to participate in the collaboration needs to do so explicitly.

The participant pattern template

- An abstract aspect that contains the core logic for implementing the concern; but it defers the definition of the `slowOperation`() pointcut to subaspects. The required crosscutting behavior is in the advice to the abstract pointcut.



```
public abstract aspect AbstractWaitCursorManagementAspect {
    public abstract pointcut slowOperation();

    before() : slowOperation() {
        WaitCursor.set();
    }

    after() : slowOperation() {
        WaitCursor.unset();
    }
}
```

The participant pattern template

- The subaspect defines the base aspect’s abstract pointcut.

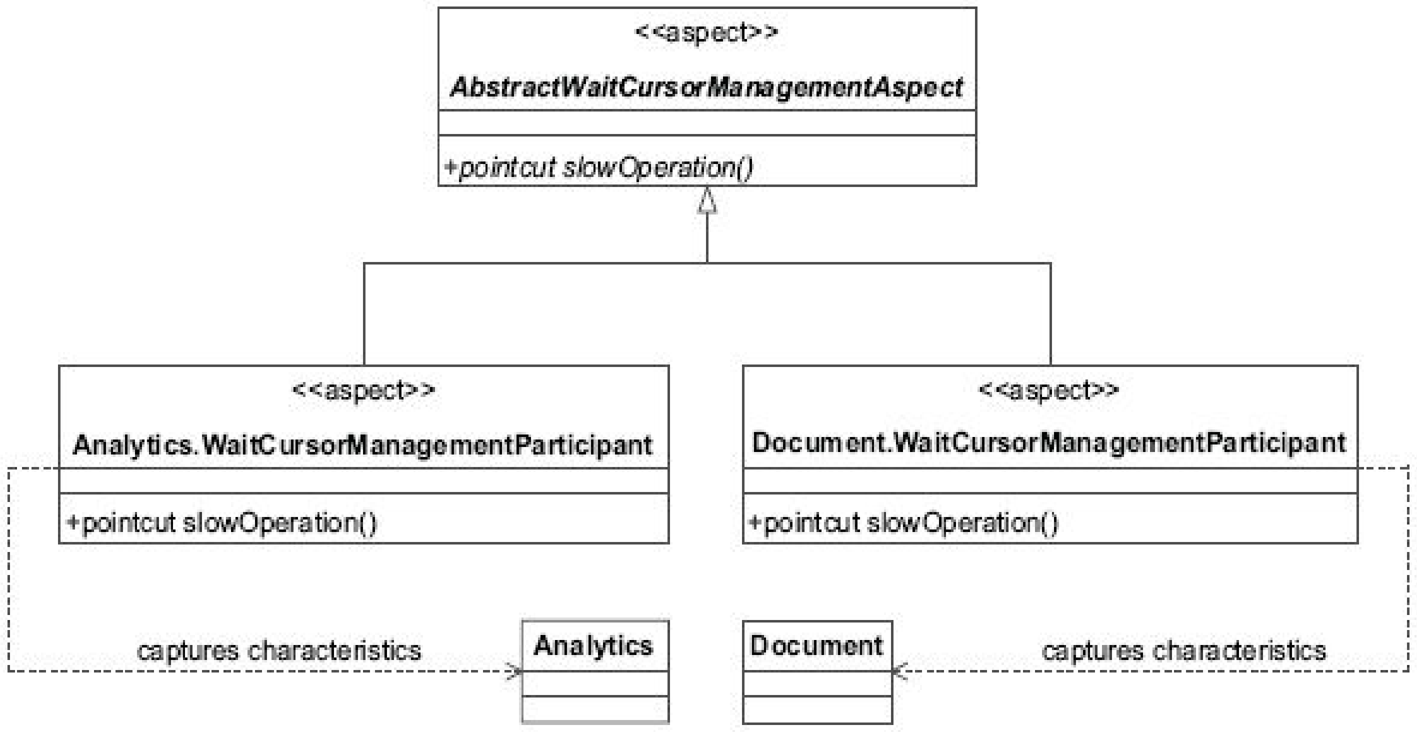
```
public class Analytics {

    // Analytics' implementation

    private static aspect WaitCursorManagementParticipant
        extends AbstractWaitCursorManagementAspect {
        public pointcut slowOperation() :
            execution(* Analytics.monteCarloSimulation())
            || execution(* Analytics.updateDatabase());
    }
}
public class Document {

    // Document's implementation

    private static aspect WaitCursorManagementParticipant
        extends AbstractWaitCursorManagementAspect {
        public pointcut slowOperation() :
            execution(* Document.save())
            || execution(* Document.backupSave());
    }
}
```

The participant pattern template

- There can be many more participants in addition to `Analytics` and `Document`. Each of the participating nested subaspects provide a definition to select the join points in their enclosing class, thus applying the functionality of the base aspect to those join points.
- A subaspect may use a naming pattern or other signature components to define the pointcut instead of enumerating the methods.
- With the participant pattern, the collaborating classes explicitly participate in the implementation of the crosscutting concerns by extending an inviting abstract aspect and providing the definition for its abstract pointcut.
- It is possible to have one participant per class hierarchy or package. But in such cases, because the aspect is not nested in the class, you must remember to modify the pointcut in the participant aspect whenever the list of methods matching the desired characteristic changes.

Structural overview

- For each class, a nested subaspect exists to make the class participate in the collaboration offered by the base aspect.


<<aspect>>
AbstractWaitCursorManagementAspect
+pointcut slowOperation()
<<aspect>>
Analytics.WaitCursorManagementParticipant
+pointcut slowOperation()
<<aspect>>
Document.WaitCursorManagementParticipant
+pointcut slowOperation()
captures characteristics
Analytics
Document
captures characteristics

The participant pattern template

- Crosscutting concern implementations often require some behavior specific to the advised type to be invoked from advice.
- For example, an authorization aspect may need the roles that are authorized to access specific functionality. In many cases, the base aspect can leverage join point context to deduce the needed behavior.
- In such cases, you need collaboration from classes not only in supplying the pointcuts but also in augmenting the behavior.
- The setup for the participant pattern easily accommodates such scenarios.
- You include one or more abstract methods in the base aspect. Each participating aspect can then implement these methods.
- This pattern applies the template method design pattern in an aspect setting: the abstract methods form the template method, and the base aspect invokes them as a part of some template logic.

The participant pattern template

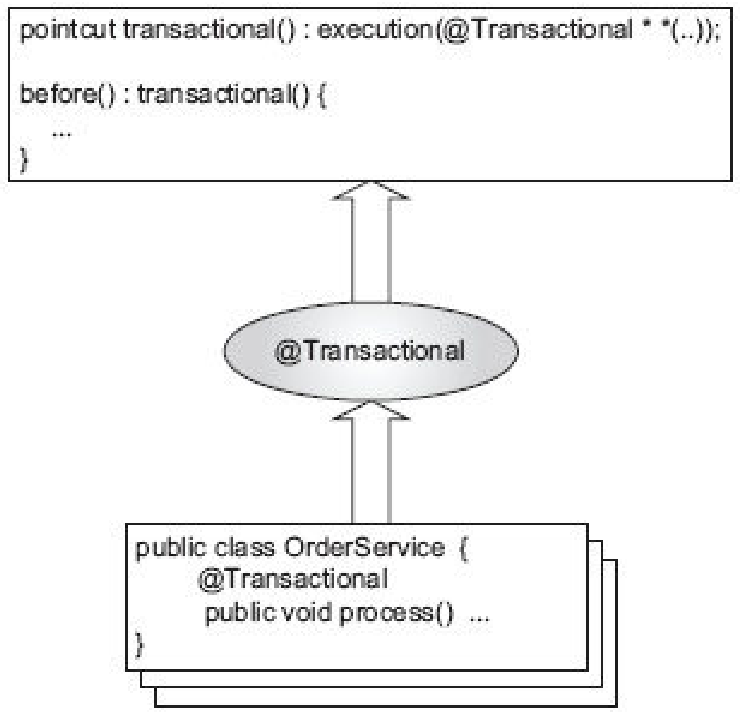
- The participant pattern provides a good solution, but it suffers from a couple of shortcomings:
  - It requires participants to be directly aware of the base aspect.
  - It makes it difficult to remove the aspect. Because the classes refer to the base aspect, you always need it to successfully compile the classes.
- You can overcome some of these limitations with a variation of the pattern that uses annotations.

Annotation-driven participant pattern

- The core concept in the participant pattern is explicit participation by the participants (classes, typically).
- There is a variation of the classical participant pattern: the annotation-driven participant pattern (ADPP).
- The key to annotation driven participation is changing how the participants show interest in the functionality offered by an aspect.
- Whereas the classical version of the pattern uses a pointcut, the ADPP uses one or more annotation types and expects the participants to attach the annotations to appropriate elements.

Annotation-driven participant pattern

- The annotation-driven participant pattern uses an aspect that expects program elements to carry certain annotations. The classes participate in the functionality offered by marking program elements with the expected annotation.



```
pointcut transactional() : execution(@Transactional * *(..));

before() : transactional() {
    ...
}
```

```
public class OrderService {
        @Transactional
         public void process() ...
}
```

Current solution

- The classical participant pattern is the most common current solution that addresses the problem of selecting join points of the desired characteristics.
- It represents one of the best options when there is not a way to use annotations.
- The classical participant pattern also offers a few customization possibilities that are harder to achieve with ADPP.
- For example, in the classical pattern, you can create multiple subaspects, each to suit a specific need. Each subaspect can also provide a variation of a certain behavior by overriding methods in the base aspect.
- The use of an abstract base aspect should be preserved even in ADPP to allow for such customization.

Pattern Template

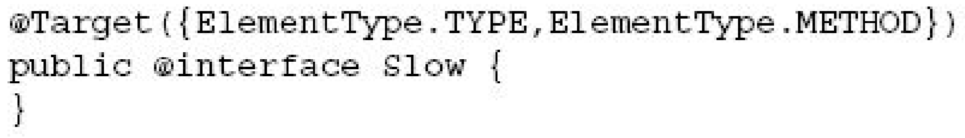
- The aspect extends the base aspect developed for the classical pattern. The slowOperation() pointcut selects methods with the @Slow annotation or methods in the classes with the @Slow annotation.

```
public aspect SlowAnnotationDrivenWaitCursorManagementAspect
    extends AbstractWaitCursorManagementAspect {
    public pointcut slowOperation()
        : execution(@Slow * *(..)) || execution(* (@Slow *).*(..));
}
```

- You still use the base aspect that does not use any annotations and a subaspect that defines the pointcuts using annotations. This arrangement provides freedom regarding the type of annotation used and lets you go back to the classical participant pattern, if necessary.

Pattern Template

- The subaspect uses the @Slow annotation. It specifies that it may be used to annotate types and methods.



```
@Target({ElementType.TYPE,ElementType.METHOD})
public @interface Slow {
}
```

- You need to mark individual methods or classes (thus designating all methods) with the @Slow annotation.

Pattern Template

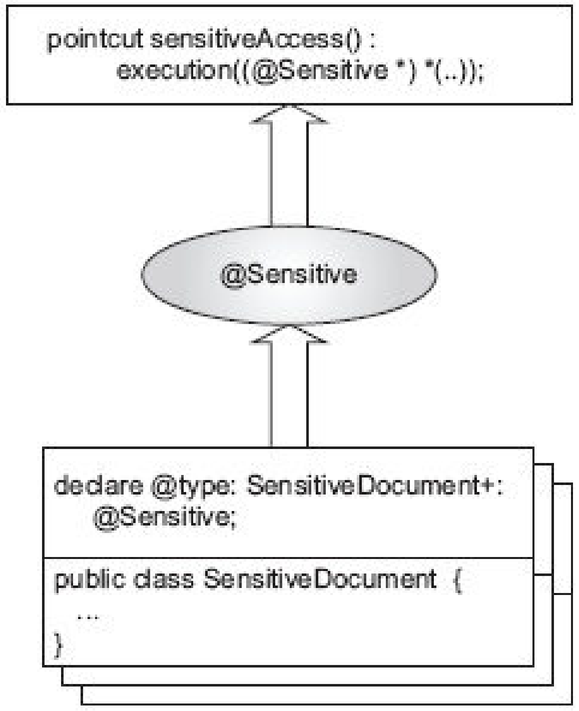
```
 public class Analytics {
     @Slow
     public void monteCarloSimulation() {
         ...
     }

     @Slow
     public void updateDatabase() {
         ...
     }

     ...

 }
@Slow
public class Document {
    public void save() {
        ...
    }

    public void backupSave() {
        ...
    }

    ...

}
```

Bridged participation pattern

- A few annotations make some developers uncomfortable.
- They have the sense that using annotations (especially framework-specific annotations) strips Plain Old Java Objects of their plainness.
- The bridged participation pattern helps to overcome this dilemma.
- In this case, you can use an aspect that supplies the annotation expected by a framework aspect.
- The core idea of this pattern is to use some characteristic of existing program elements to supply the expected annotations to those elements using `declare @type`,`declare @method`, `declare @constructor`, or `declare @field` statements in AspectJ.
- With this arrangement, the classes do not directly depend on the framework, but they benefit from its functionality.
- This pattern also helps when you cannot modify existing source code to add annotations.

Bridged participation pattern

- The aspect advises join points based on the associated annotations just as in ADPP. But instead of marking program elements with annotations, additional aspects attach annotations to those elements through declare statements.



```
pointcut sensitiveAccess() :
        execution((@Sensitive *) *(..));
```

```
declare @type: SensitiveDocument+:
    @Sensitive;
```

```
public class SensitiveDocument {
    ...
}
```

Bridged participation pattern

- Annotation bridging has a few limitations when implemented in AspectJ.
  - There is no way for a bridge to transfer annotation property values.
  - There is no way to use annotation values in making decisions to supply annotations.

  For example, you cannot supply the @Slow annotation if the @Timing annotation's value attribute exceeds 1000.
- A future version of AspectJ may remove these limitations.

Role of ADPP in library aspects

- ADPP provides great benefits in writing library aspects by simplifying the collaboration needed by the library users.
- The aspects in the library often use the bridged participation pattern.
- Instead of writing subaspects and pointcuts (which can be a daunting task for programmers not yet experienced with AOP), library users mark program elements with annotations to participate in the collaboration.
- The Spring Framework, uses a few annotations to simplify how library users participate in the provided functionality:
    - The @Transactional annotation lets you designate classes and methods that need to be executed in a transactional context.
    - The @Secured annotation lets you designate a secured method and specify the authorization requirements.
    - The @Configurable annotation offers a way for objects that are not Spring beans to have their dependencies injected.