

Architecture of Enterprise Applications 14

Aspect-Oriented Programming

Haopeng Chen

***RE**liable, **IN**elligent and **Sc**alable Systems Group (**REINS**)*

Shanghai Jiao Tong University

Shanghai, China

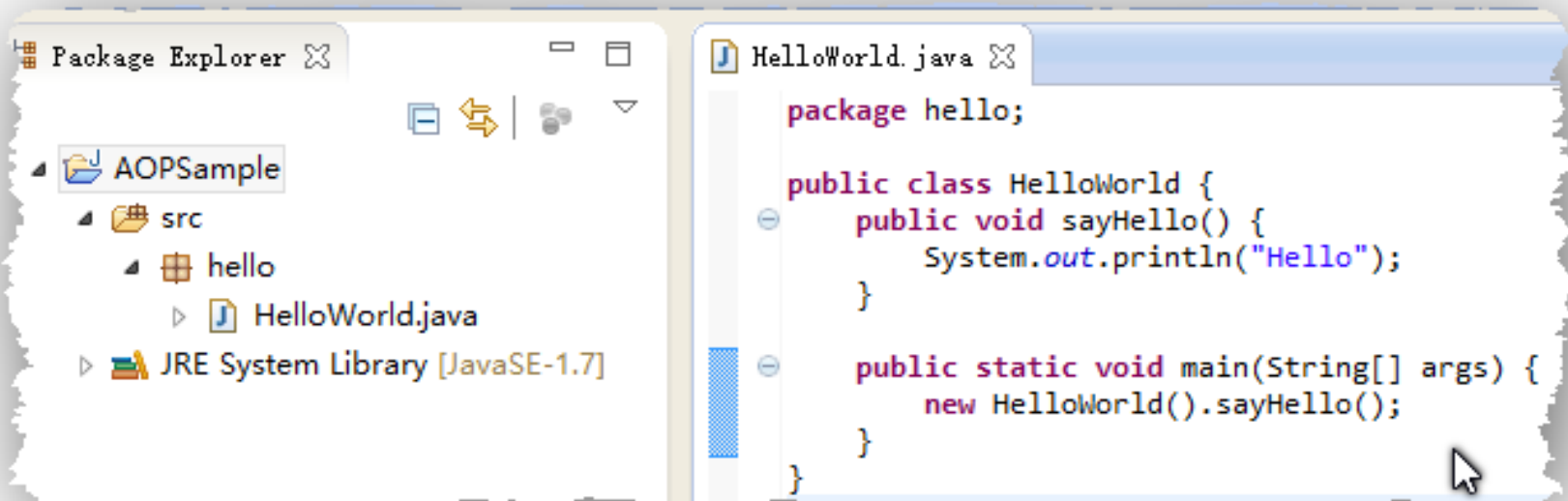
<http://reins.se.sjtu.edu.cn/~chenhp>

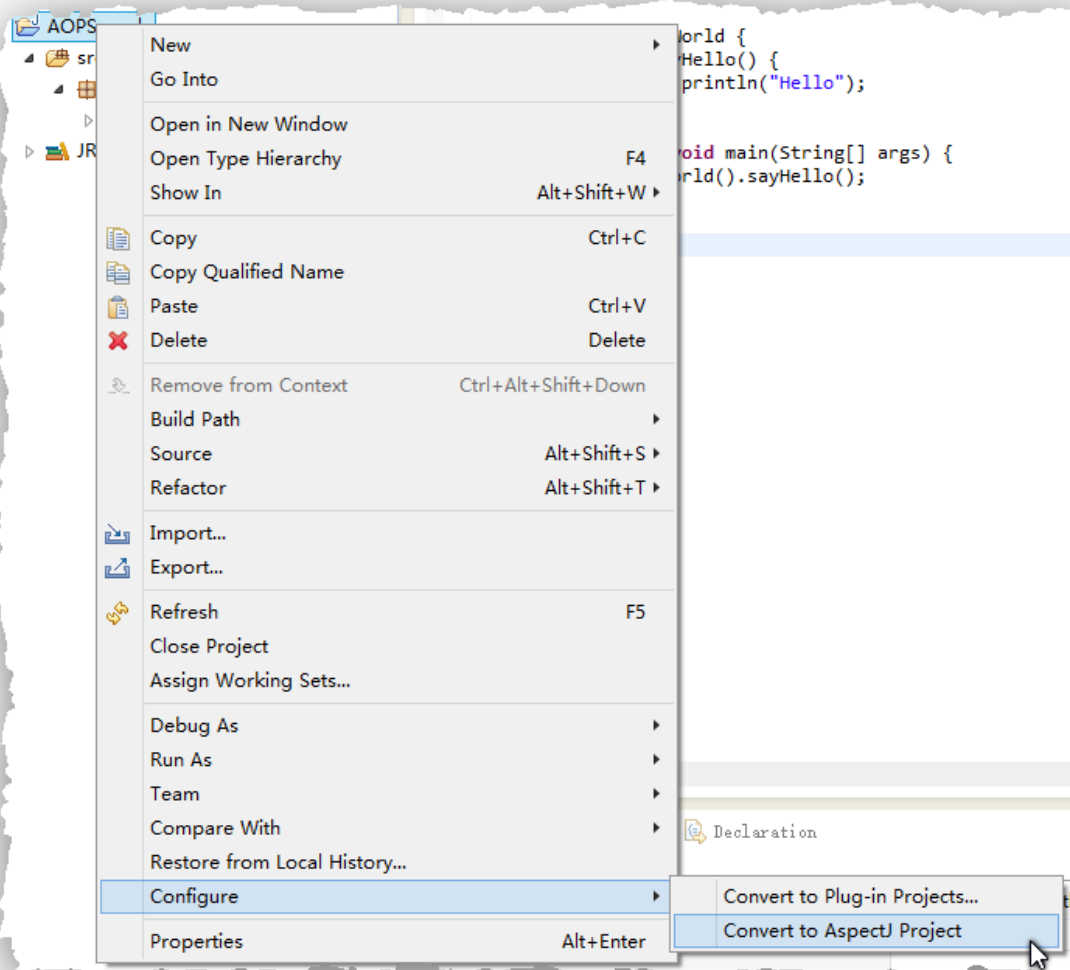
e-mail: chen-hp@sjtu.edu.cn

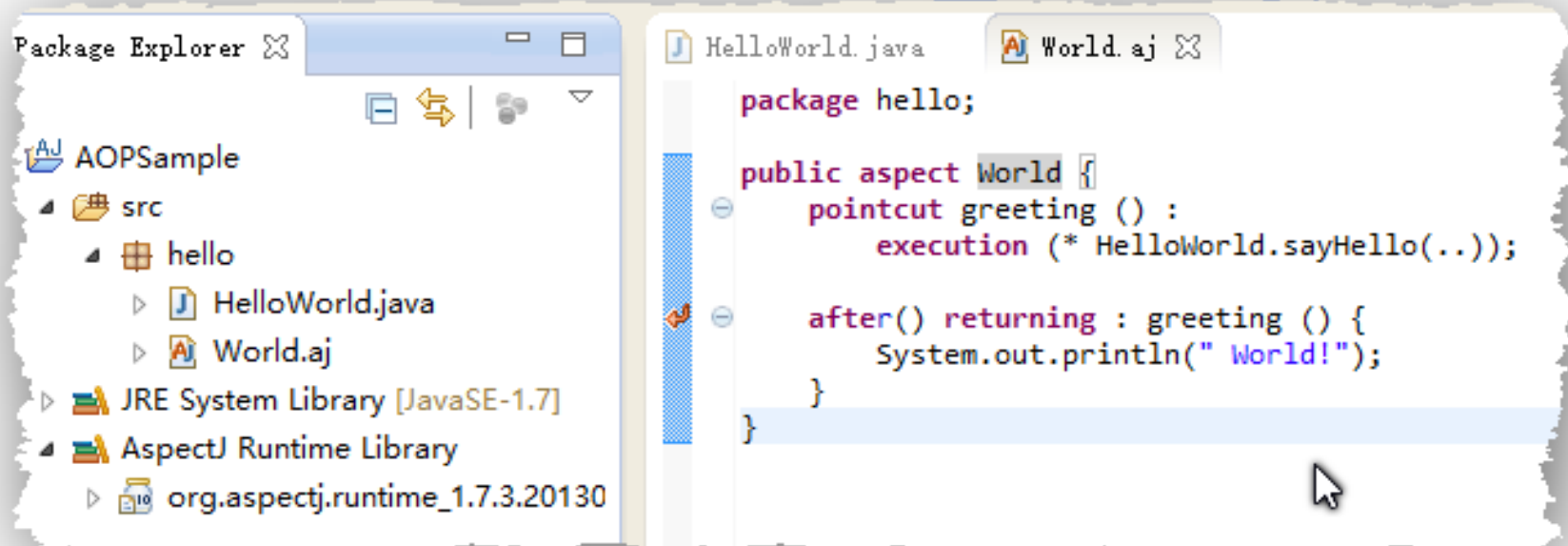
- AOP
 - Concepts
 - Type of advice
 - AspectJ
 - Examples

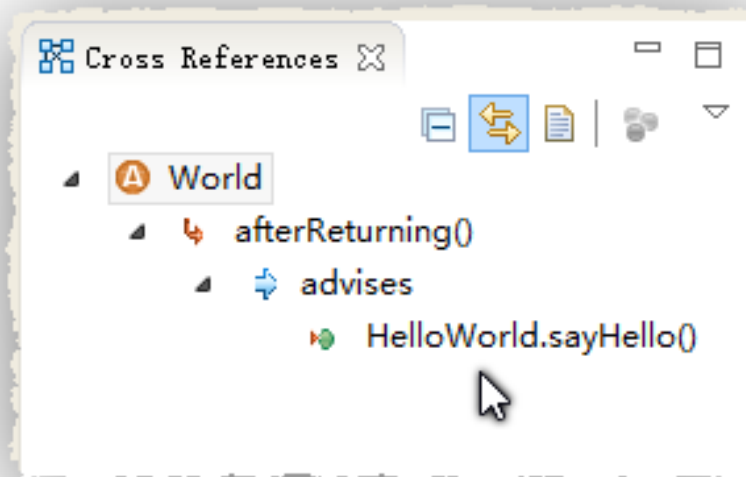
- For object-oriented programming languages, the natural unit of modularity is the **class**.
 - But some aspects of system implementation,
 - such as **logging, error handling, standards enforcement** and **feature variations**
 - are notoriously difficult to implement in a modular way.
 - The result is that code is **tangled across** a system and leads to quality, productivity and maintenance problems.
- Aspect-oriented programming is a way of modularizing **crosscutting concerns**
 - much like object-oriented programming is a way of modularizing **common** concerns.

- *Aspect-Oriented Programming (AOP)*
 - complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure.
 - The key unit of modularity in **OOP** is the **class**,
 - Whereas in **AOP** the unit of modularity is the *aspect*.
- Aspects enable the modularization of **concerns**
 - such as transaction management that **cut across** multiple types and objects.
 - (Such concerns are often termed *crosscutting* concerns in AOP literature.)

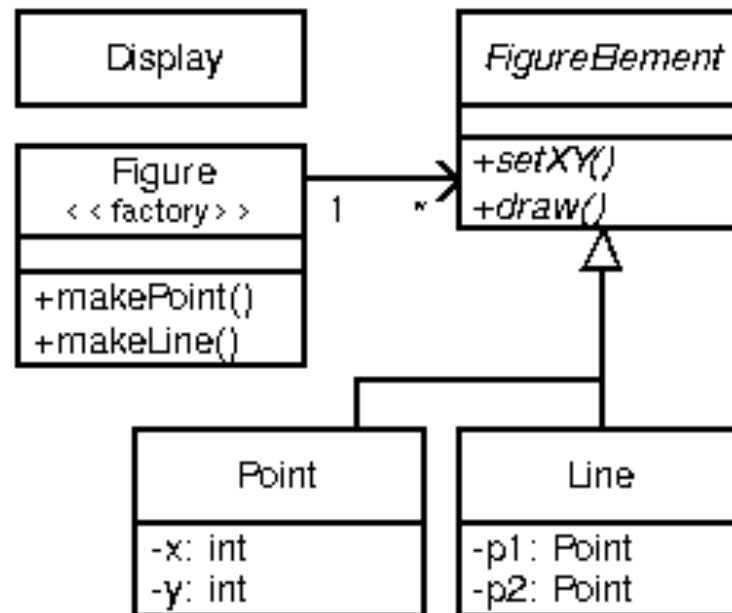








- A simple figure editor system.
 - A **Figure** consists of a number of **FigureElements**, which can be either **Points** or **Lines**. The **Figure** class provides factory services. There is also a **Display**.



- Let us begin by defining some central AOP concepts and terminology:
 - *Join point*: a point during the execution of a program, such as the execution of a method or the handling of an exception.
 - In Spring AOP, a join point *always* represents a method execution.
 - AspectJ provides for many kinds of join points, but the most common one is: *method call join points*.
 - Each method call at runtime is a different join point, even if it comes from the same call expression in the program.

- Let us begin by defining some central AOP concepts and terminology:
 - *Pointcut*: a predicate that matches join points.
 - Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).
 - The concept of join points as matched by pointcut expressions is central to AOP.

- In *AspectJ*, *pointcuts* pick out certain join points in the program flow.
 - For example, the pointcut
`call(void Point.setX(int))`
 - picks out each join point that is a call to a method that has the signature `void Point.setX(int)` — that is, `Point`'s `void setX` method with a single `int` parameter.
 - A pointcut can be built out of other pointcuts with and, or, and not (spelled `&&`, `||`, and `!`). For example:
`call(void Point.setX(int)) ||`
`call(void Point.setY(int))`
 - picks out each join point that is either a call to `setX` or a call to `setY`.

- In our example system, this pointcut captures all the join points when a **FigureElement** moves.
 - So AspectJ allows programmers to define their own named pointcuts with the pointcut form. So the following declares a new, named pointcut:

```
pointcut move():  
    call(void FigureElement.setXY(int,int)) ||  
    call(void Point.setX(int)) ||  
    call(void Point.setY(int)) ||  
    call(void Line.setP1(Point)) ||  
    call(void Line.setP2(Point));
```
 - and whenever this definition is visible, the programmer can simply use `move()` to capture this complicated pointcut.
- The above pointcuts are all based on explicit enumeration of a set of method signatures.
 - We sometimes call this *name-based* crosscutting.

- AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name.
 - We call this *property-based* crosscutting.
- The simplest of these involve using wildcards in certain fields of the method signature.
 - For example, the pointcut
`call(void Figure.make*(..))`
 - picks out each join point that's a call to a void method defined on **Figure whose the name begins with "make"** regardless of the method's parameters.
 - In our system, this picks out calls to the factory methods `makePoint` and `makeLine`.
 - The pointcut
`call(public * Figure.* (..))`
 - picks out **each call to Figure's public methods**.

- Let us begin by defining some central AOP concepts and terminology:
 - *Advice*: action taken by an aspect at a particular join point.
 - Different types of advice include "**around**," "**before**" and "**after**" advice.
 - Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.

- Types of advice:
 - *Before advice*:
 - Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
 - *After returning advice*:
 - Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
 - *After throwing advice*:
 - Advice to be executed if a method exits by throwing an exception.
 - *After (finally) advice*:
 - Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
 - *Around advice*:
 - Advice that surrounds a join point such as a method invocation.
 - This is the most powerful kind of advice.
 - Around advice can perform custom behavior before and after the method invocation.
 - It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

- AspectJ has several different kinds of advice.
 - *Before advice* runs as a join point is reached, before the program proceeds with the join point.

```
before(): move() {  
    System.out.println("about to move");  
}
```
 - *After advice* on a particular join point runs after the program proceeds with that join point. there are three kinds of after advice: after returning, after throwing, and plain after.

```
after() returning: move() {  
    System.out.println("just successfully moved");  
}
```
 - *Around advice* on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point.

- Exposing Context in Pointcuts
 - Pointcuts not only pick out join points, they can also expose part of the execution context at their join points.
 - Values exposed by a pointcut can be used in the body of advice declarations.
 - An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses. For example, the after advice

```
after(FigureElement fe, int x, int y) returning:  
    ...SomePointcut... {  
        ...SomeBody...  
    }
```
 - uses three pieces of exposed context, a `FigureElement` named `fe`, and two `ints` named `x` and `y`.

- Exposing Context in Pointcuts
 - The advice's pointcut publishes the values for the advice's arguments. The three primitive pointcuts **this**, **target** and **args** are used to publish these values. So now we can write the complete piece of advice:

```
after(FigureElement fe, int x, int y) returning:  
    call(void FigureElement.setXY(int, int))  
    && target(fe)  
    && args(x, y) {  
    System.out.println(fe + " moved to (" + x + ", " + y + ")");  
}
```
 - The pointcut exposes three values from calls to **setXY**: the target **FigureElement** -- which it publishes as **fe**, and the two **int arguments** -- which it publishes as **x** and **y**.
 - So the advice prints the figure element that was moved and its new **x** and **y** coordinates after each **setXY** method call.

- Exposing Context in Pointcuts

- A named pointcut may have parameters like a piece of advice.
- When the named pointcut is used (by advice, or in another named pointcut), it publishes its context by name just like the this, target and args pointcut.

- So another way to write the above advice is

```
pointcut setXY(FigureElement fe, int x, int y):  
    call(void FigureElement.setXY(int, int))  
    && target(fe)  
    && args(x, y);
```

```
after(FigureElement fe, int x, int y) returning:  
    setXY(fe, x, y) {  
    System.out.println(fe + " moved to (" + x + ", " + y + ").");  
}
```

- Let us begin by defining some central AOP concepts and terminology:
 - **Introduction**: declaring additional methods or fields on behalf of a type.
 - Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object.
 - For example, you could use an introduction to make a bean implement an **IsModified** interface, to simplify caching.
 - Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies.
 - They may declare members that cut across multiple classes, or change the inheritance relationship between classes.
 - Unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time.

- Suppose we want to have **Screen** objects observe changes to **Point** objects, where **Point** is an existing class.
 - We can implement this by writing an aspect declaring that the class **Point** has an instance field, **observers**, that keeps track of the Screen objects that are observing Points.
 - The **observers** field is private, so only **PointObserving** can see it. So observers are added or removed with the static methods **addObserver** and **removeObserver** on the aspect.
 - Along with this, we can define a pointcut **changes** that defines what we want to observe, and the after advice defines what we want to do when we observe a change.
 - Note that neither **Screen**'s nor **Point**'s code has to be modified, and that all the changes needed to support this new capability are local to this aspect.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }

    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }

    pointcut changes(Point p):
        target(p) && call(void Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }

    static void updateObserver(Point p, Screen s) { s.display(p); }
}
```

- Let us begin by defining some central AOP concepts and terminology:
 - **Aspect**: a modularization of a concern that cuts across multiple classes.
 - Transaction management is a good example of a crosscutting concern in enterprise Java applications.
 - Like classes, aspects may be instantiated, but AspectJ controls how that instantiation happens -- **so you can't use Java's new form to build new aspect instances.**
 - By default, **each aspect is a singleton**, so one aspect instance is created. This means that advice may use non-static fields of the aspect, if it needs to keep state around:

```
aspect Logging {  
    OutputStream logStream = System.err;  
    before(): move() {  
        logStream.println("about to move");  
    }  
}
```


- Let us begin by defining some central AOP concepts and terminology:
 - *Target object*: object being advised by one or more aspects.
 - Also referred to as the *advised* object.
 - Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.
 - *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on).
 - *Weaving*: linking aspects with other application types or objects to create an advised object.
 - This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime.
 - Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

- It is a simple tracing aspect that prints a message at specified method calls.

- In our figure editor example, one such aspect might simply trace whenever points are drawn.

```
aspect SimpleTracing {  
    pointcut tracedCall():  
        call(void FigureElement.draw(GraphicsContext));  
    before(): tracedCall() {  
        System.out.println("Entering: " + thisJoinPoint);  
    }  
}
```

- This code makes use of the `thisJoinPoint` special variable.
 - Within all advice bodies this variable is bound to an object that describes the current join point.
 - The effect of this code is to print a line like the following every time a figure element receives a draw method call:

```
Entering: call(void FigureElement.draw(GraphicsContext))
```

- The following aspect counts the number of calls to the rotate method on a Line and the number of calls to the `set*` methods of a `Point` that happen within the control flow of those calls to rotate:

```
aspect SetsInRotateCounting {  
    int rotateCount = 0;  
    int setCount = 0;  
    before(): call(void Line.rotate(double)) { rotateCount++; }  
    before(): call(void Point.set*(int))  
        && cflow(call(void Line.rotate(double))) {  
        setCount++;  
    }  
}
```
- In effect, this aspect allows the programmer to ask very specific questions like
 - How many times is the rotate method defined on Line objects called?
 - And
 - How many times are methods defined on Point objects whose name begins with "set" called in fulfilling those rotate calls?
 - questions it may be difficult to express using standard profiling or logging tools.

- AspectJ makes it possible to implement pre- and post-condition testing in modular form.

```
aspect PointBoundsChecking {
```

```
    pointcut setX(int x):
```

```
        (call(void FigureElement.setXY(int, int)) && args(x, *)) ||  
        (call(void Point.setX(int)) && args(x));
```

```
    pointcut setY(int y):
```

```
        (call(void FigureElement.setXY(int, int)) && args(*, y)) ||  
        (call(void Point.setY(int)) && args(y));
```

```
    before(int x): setX(x) {
```

```
        if ( x < MIN_X || x > MAX_X ) throw new IllegalArgumentException("x is out of bounds.");  
    }
```

```
    before(int y): setY(y) {
```

```
        if ( y < MIN_Y || y > MAX_Y ) throw new IllegalArgumentException("y is out of bounds.");  
    }  
}
```

- For example, the following aspect enforces the constraint that only the well-known factory methods can add an element to the registry of figure elements.

```
aspect RegistrationProtection {  
    pointcut register():  
        call(void Registry.register(FigureElement));  
    pointcut canRegister():  
        withincode(static * FigureElement.make*(..));  
  
    before(): register() && !canRegister() {  
        throw new IllegalAccessException("Illegal call " +  
            thisJoinPoint);  
    }  
}
```

- This advice throws a runtime exception at certain join points, but AspectJ can do better.
- Using the declare error form, we can have the *compiler* signal the error.

```
aspect RegistrationProtection {  
    pointcut register():  
        call(void Registry.register(FigureElement));  
  
    pointcut canRegister():  
        withincode(static * FigureElement.make*(..));  
  
    declare error: register() && !canRegister():  
        "Illegal call"  
}
```

- The pointcut move captures all the method calls that can move a figure element. The after advice on move sets the dirty flag whenever an object moves.

```
aspect MoveTracking {  
    private static boolean dirty = false;  
    public static boolean testAndClear() {  
        boolean result = dirty;  
        dirty = false; return result;  
    }  
  
    pointcut move():  
        call(void FigureElement.setXY(int, int)) ||  
        call(void Line.setP1(Point)) ||  
        call(void Line.setP2(Point)) ||  
        call(void Point.setX(int)) ||  
        call(void Point.setY(int));  
  
    after() returning: move() { dirty = true; }  
}
```

- Requirement
 - Try AspectJ in your BookStore.com to add logging for important operation.

- Spring Framework Reference Documentation
 - <http://docs.spring.io/spring/docs/4.0.4.RELEASE/spring-framework-reference/htmlsingle/>
- The AspectJ™ Programming Guide
 - <http://eclipse.org/aspectj/doc/released/progguide/index.html>



Thank You!