Aspect Oriented Programming

Engineering 7893 – Software Engineering

A Formal Definition

- "Aspect-oriented programming is a programming paradigm that increases modularity by allowing the separation of cross-cutting concerns, forming a basis for aspect-oriented software development."

  (Wikipedia)

AspectJ

- Possibly the most common and easily accessible Aspect Oriented Programming language is AspectJ. AspectJ is an extension of the Java language, and as such will generally have familiar syntax and concepts. There is a plugin available for Eclipse, the link will appear in the references at the end of the presentation. All of the following definitions and examples will be using the AspectJ terminology and language.

Some Key Concepts

- Cross-Cutting Concerns
  - These are issues which are common to more than one class, potentially across more than one package. Some examples of these kinds of concerns include (but are in no way limited to):
    - Security
    - Logging
    - Debugging
    - Pre-Conditions
    - Post-Conditions
    - Change Monitoring

Some Key Concepts

- Aspect
  - The AOP equivalent of a class. It contains pointcut definitions and advice which are executed at the join points defined by the combination of the advice and the pointcuts. It may also contain inter-type declarations.
- Inter-Type Declarations
  - This concept allows users to add methods to pre-existing classes which may not necessarily belong in a single class, but may be useful over the bounds of several classes.

Some Key Concepts

- Join Points
  - Join Points are places in the body of a code segment where a piece of advice could happen. Note: Join Points are a concept. They are never defined or realized in any way.
- Pointcut
  - A definition of all the places (Join Points) where a piece of advice will be executed.
- Advice
  - The code which will be executed when a Join Point which is defined by a pointcut is reached.

Concept Summary

- When using Aspect Oriented Programming, we declare aspects (classes) which may contain advice (methods) as well as pointcuts (A way of calling the methods at particular times.) So essentially, we can create a class which is able to call it's own methods at specific times which are defined by the programmer.

Putting it Together

- When defining pointcuts inside an aspect, a programmer is not limited to choosing join points from within a single class or even package inside the project. This allows a single piece of advice to be applied to any number of different locations in a project. As such, concerns which have merit inside of completely unrelated regions can be addressed without having to relate segments of code which should not be related.

Simple Examples

- The following is a series of simple examples which will be used to explain the basic concepts of Aspect Oriented Programming.
- All of the simple examples revolve around the Point class in Java. This means that it doesn't matter where in a section of code a Point may have been used as a part of another data structure, the changes made by the AOP section of code will apply to every Point which is created.

Pointcut Examples

- call(void Point.setX(int))
  - This is a very simple pointcut. It defines a join point which occurs when a call is made to the setX(int) method of the Point class.
- call(void Point.set*(int))
  - This is another simple pointcut, this time using a wild card. It defines a join point which occurs when a call is made to any method of the Point class which begins with "set" and takes an int as a parameter.

Pointcut Examples

- call(void Point.set*(..))
  - Building on the previous example, this pointcut defines a join point which occurs whenever a call is made to any method of the Point class which begins with "set", but this time it can take any parameters at all instead of just a single int.
- call(public * Point. *(..))
  - This pointcut defines a join point which happens whenever a call is made to a public method of the Point class which can be called anything, can return anything, and can take any parameters at all.

Pointcut Examples

- pointcut set(): call(void Point.set*(..));
  - Pointcuts can also be named, which allows them to be used to specify when advice should happen simply by using their name.
- pointcut access(): call(void Point.set*(..)) | | call(int Point.get*(..));
  - Pointcuts can be joined using the logical operators or (| |), and (&&), and not (!). This defines a join point that happens when a call is made to any method that begins with "set" or any method that begins with "get" and returns an int.

Advice Examples

- There are three kinds of advice, advice which is run before a join point is executed, advice which is run after a join point is executed, and advice which is run around a join point. Around advice is a more complex topic, and won’t be covered in this presentation.

Advice Examples

- before(): set() {

  System.out.println("About to change position.");

  }

  - Before advice is run when a join point is reached, but before the call defined by the pointcut is made.

Advice Examples

- after() returning: set() {

  System.out.println(“Position has been changed.”);

  }
  - After advice is run when the control of the program is returned to the join point. This happens when the call defined by the pointcut returns.
  - Because a Java method can either return normally, or throw an exception, after advice can have three different forms. after() returning, after() throwing, and after(). The plain after() happens after either throwing or returning.

Advice Examples

```
before(Point p, int x, int y):
    call(void Point.setLocation(int, int)) && target(p) && args(x, y) {
    System.out.println("Position of " + p + " will be changed to: ( " + x + ", " + y + ")");
    }
```

- Advice and pointcuts can be modified to expose information about the call being made at that pointcut. This is done using the target and args primitives. "target" returns the object in which the method is being called. "args" returns the arguments passed to the function in the same order as they are being passed.

Aspect Example

```
aspect SimpleAspect {

pointcut set(): call(void Point.set*(..));

before(): set() {

System.out.println("About to move!");

}

after(): set() {

System.out.println("Just moved!");

}

}
```

- An aspect is much like a class in that it wraps up a group of related concerns and packages them into one convenient location so that it is easily found and modified.
- Aspects never need to be instantiated, they are all similar in functionality to singleton classes in that there is only one instance of each aspect and it is created at compile time. An aspect will perform it's own advice based on it's own pointcuts.

Inter-Type Declarations

```
aspect PointExtension {

private double Point.orientation = 0;

public static double getOrientation(Point
    p){

return p.orientation;

}

after(int x, int y): call(void
    Point.setLocation(int, int)) && args(x,y) {

orientation = Math.atan2(x,y);

}

}
```

- Aspects can be used to add information to a class. In the example to the left, the Point class has been made to have an orientation field which updates itself automatically whenever a call is made to setLocation().
- This example is not terribly useful, as it could just as easily be realized by extending the point class. This functionality is more useful as shown on the following slide when used to define interactions between more than one type of class. Assume that a "Screen" is equivalent to a Canvas.
- The example on the following page is taken from the AspectJ introduction which is listed in the references.

Inter-Type Declarations

```
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); }

}
```

Situational Examples

- So far, most of the examples which have been given were used simply to explain the syntax and basic concept of Aspect Oriented Programming. The following few slides contain examples of how these concepts can be put to use in a practical way, which may help explain the purpose behind Aspect Oriented Programming.
- There is not time to discuss all of the possible examples, so only a few will be mentioned here.

Tracing / Debugging

One very practical use of AOP is in debugging code. Using Eclipse's built in debugging mode is the can be a pain, having to switch between perspectives and sift through all kinds of information to find the single value you were looking for. Often times it is easiest just to put a System.out call in a method to print the value you'd like to know. If you're trying to trace a problem, you may like to know what a value is before and after a given call, so you may put in a whole series of System.out calls. This is great, but tracking them down to remove them when you have solved to problem is usually painful.

Tracing / Debugging

Using AOP, you can group a whole series of debugging calls into a single aspect, which is easy to find after debugging has been completed. The use of before() and after() advice along with the ability to expose information from a join point allows reasonably precise debugging of what a value is over the course of a series of calls.

Conditions

- AOP is also useful for implementing things like pre-conditions and post-conditions. For example:

```
aspect PreCondition {

before(int x, int y): call(void Point.setLocation(int, int)) &&
   args(x,y)) {

if (x < 0) throw new IllegalArgumentException(“Invalid x”);

if (y < 0) throw new IllegalArgumentException(“Invalid y”);

}

}
```

Conclusion

- In simple terms, AOP is basically a way of writing singleton classes which have the ability to call themselves throughout the entirety of a piece of code. This allows them to provide the same functionality to several unrelated locations in the code without having to change the existing relationships between segments of code. This can be extremely useful.

References

- http://en.wikipedia.org/wiki/Aspect-oriented_programming
- http://en.wikipedia.org/wiki/AspectJ
- http://en.wikipedia.org/wiki/Pointcut
- http://en.wikipedia.org/wiki/Join_point
- http://www.eclipse.org/aspectj/doc/released/progguide/index.html
- http://eclipse.org/aspectj/