A-030

アスペクトのインスタンス化とインスタンス単位でのアスペクトの適用
Aspect Instantiation and Instance Level Aspects

下滝 亜里†
Asato Shimotaki

1. はじめに

アスペクト指向プログラミング（AOP）は、ロギングのような複数のモジュール間に影響を与えるような機能（横断的関心事と呼ばれる）を、アスペクトと呼ばれるモジュール単位にカプセル化するための新しい取り組みである。AspectJ[1]は、アスペクト指向のために Java を拡張した言語であり、ジョインポイント、ポイントカット、アドバイスといったコンセプトを基にして横断的関心事をアスペクトとして記述することを可能にする。ジョインポイントとは、プログラム実行中の時点（あるいは、イベント）であり、どのようなジョインポイントがあるのかは、あらかじめ決められている。たとえば、メソッドの呼び出し時点（call）やフィールドが読み出された時点（get）や、例外が発生した時点（handler）などがある。また、ポイントカットは、それらのジョインポイント（イベント）を選び出すために使用される。アドバイスは、ポイントカットによって選び出されたジョインポイントにおいて実行されるコードである。アドバイスには、before advice、after advice、around advice の 3 つの種類があり、それぞれ、ジョインポイントの前、後、代わりに、を表す。

ロギングは、AOP の有効性を示すために使用される代表例の一つであるが、より複雑なロギングを実現することは、AspectJ が備えているような単純なアスペクトの機能だけでは難しい。これは、AspectJ ではプログラマは任意にアスペクトをインスタンス化できず、また、クラス単位ではなくインスタンス単位でアスペクトを適用できないためである。以下ではこの二つの機能を備えたアスペクトをインスタンスレベルアスペクト[2]と呼ぶ。

インスタンスレベルアスペクトは、ロギングだけでなく、その他の機能の実現の際にも有効であるため、容易に利用できる必要がある。本稿では、AspectJ 環境において、インスタンスレベルアスペクトを実現するための二つのアプローチを提案する。1 つは、コンパイラを拡張することなく AspectJ だけを用いて再利用可能なインスタンスレベルアスペクトを実装するアプローチである。このアプローチでは、要求されるほとんどのインスタンスレベルアスペクトの機能の使用を可能にするが、AspectJ を用いるだけでは解決できないようないくつかの制限が残る。そのため、もう 1 つのアプローチとして、クラスにタグ付けされた情報を基に、そのクラスをインスタンスレベルアスペクトとして用いることができるコード生成システムを提案する。

2 節では、ロギングの例を基に AspectJ におけるアスペクトの機能不足を指摘する。3 節と 4 節では、これらの二つのアプローチを述べる。5 節はまとめである。

† 大阪産業大学大学院工学研究科情報システム工学専攻

2. 問題

図 1 に示すような Point クラスに対する振る舞いをロギングしたいとする。

```
class Point {
  private int x;
  private int y;
  Point(int x, int y) { this.x = x; this.y = y; }
  void setX(int x) {
    this.x = x;
    System.out.println("x = " + x); // 単に動作確認用
  }
  void setY(int y) {
    this.y = y;
    System.out.println("y = " + y); // 単に動作確認用
  }
}
```

図 1 Point クラス

Point クラスのすべてのインスタンスの振る舞いをロギングする場合には、図 2 に示しているように、AspectJ では容易に実現できる。

```
aspect PointLogging {
  after() : call(void Point.setX(int)) {
    System.out.println("x updated" );
  }
  after() : call(void Point.setY(int)) {
    System.out.println("y updated" );
  }
}
```

図 2 PointLogging アスペクト

しかし、特定のインスタンスのみに対する振る舞いをロギングする場合には、AspectJ では適切に実装することは容易でない。最も直接的な実装方法は、アスペクトの適用対象となるオブジェクトを保持しておき、各アドバイスの実行時に、そのジョインポイントにおけるオブジェクトに対してアドバイスが実行可能かどうかをチェックすることである（図 3）。しかし、このような実装では、各アドバイス（例では setX と setY）に対してアドバイスが実行可能かどうかのチェックが必要になるため、十分な解決策とは言えない。AspectJ は、横断的関心事を適切にカプセル化できるとされているが、このように、アドバイス内において発生する横断的関心事については容易にモジュール化できない。

```
aspect PointLogging {
  private List points = new ArrayList();
  after() : call(void Point.setX(int)) {
    Object p = thisJoinPoint.getTarget()
    if (points.contains(p) ) {
      System.out.println("x updated" );
    }
  }
  after() : call(void Point.setY(int)) {
    Object p = thisJoinPoint.getTarget()
    if (points.contains(p) ) {
      System.out.println("y updated" );
    }
  }
  public void addPoint(Point p) { points.add(p); }
}
```


図 3 PointLogging アスペクト

アドバイス実行時のジョインポイントを指定できる adviceexecution ポイントカットを用いることにより、before アドバイスと after アドバイスに関しては、適切にモジュール化できるが、around アドバイスに関しては特別な注意が必要である。around アドバイスを実行しないとは、before アドバイスや after アドバイスの時のように何も実行しないというわけでなく、元々のジョインポイントを呼び出すこと意味しているためである。元々のジョインポイントを呼び出すことは可能であるが、AspectJ の内部機能、つまり、公式には公開されていないクラス（AroundClosure）に頼る必要がある。

また、特定のインスタンスのみに対してアスペクトを適用できるようするだけでは、不十分である。たとえば、新たに、ログの出力先としてファイルとコンソールの指定を可能にする場合、アスペクトのインスタンスが一つだけでは、この要求の実現は難くなる。

図 4 に示すように、出力先のストリームを受け取るメソッドを定義したとする。

```
aspect PointLogging {
 private PrintWriter out;
 public void setOutputStream(OutputStream out) {
   this.out = new PrintWriter(out, true);
 }
 after() : call(void Point.setX(int)) {
  out.println("x updated" );
 }
 after() : call(void Point.setY(int)) {
  out.println("y updated" );
 }
}
```

図 4 ログの出力先の指定

この実装の問題点は、たとえば、コンソールとファイルに同時にログを出力したいという要求が発生したときに起こる。この時、利用できるアスペクトのインスタンスは一つだけであるため、この実装では複数のストリームオブジェクトを設定することができない。また、配列として渡すことも考えられるが、別の問題を招いてしまう[3]。

この要求を満たすことのできる他の実装方法も考えられるが、アスペクトをインスタンス化できるのであれば、この要求を満たすのは容易である（図 5）。

```
PointLogging cLog = new PointLogging(); //コンソール
cLog.setOutputStream(System.out);

PointLogging fLog = new PointLogging(); // ファイル
FileOutputStream file = // …
fLog.setOutputStream(file);
```

図 5 理想的な使用例

しかし、AspectJ では、new を使ってアスペクトのインスタンスを任意に生成できない。

このように、ロギングのような代表的な例であったとしても、アスペクトを任意にインスタンス化できないことやインスタンス単位でアスペクトを適用できないことは、AspectJ の適用範囲を狭め、より複雑な解決策を要求する。次節では、解決策の一つとして、ここで示したような要求を実現できる再利用可能なインスタンスレベルアスペクトの使用例と実装方法を述べる。

3. InstanceLevelAspect による実装

抽象アスペクトである InstanceLevelAspect は、インスタンス単位でアスペクトを適用できる機能と、アスペクトのインスタンス化の両方を可能にする再利用可能なアスペクトである[3]。

InstanceLevelAspect を用いて、PointLogging を実装する例を図 6 に示す。図から分かるように、インスタンスレベルアスペクトとして扱いたいアスペクト（例では PointLogging）は、InstanceLevelAspect を継承すればよい。ただし、それに加えて、以下の二つのルールに従わなければならない。

```
aspect PointLogging extends InstanceLevelAspect {
 static PointLogging newInstance() { return null; }
 void addPoint(Point p) { addObject(p); }
 after() : call(void Point.setX(int)) {
  System.out.println(" x updated " + this);
 }
 after() : call(void Point.setY(int)) {
  System.out.println(" y updated " + this);
 }
 static aspect TargetCheckImpl extends TargetCheck {
  protected pointcut adviceTarget() : target(Point);
 }
}
```

図 6 InstanceLevelAspect による実装

1 つ目は、アスペクトのインスタンスを生成して返すための静的メソッドである、newInstance を定義することである。ただし、InstanceLevelAspect が適切なインスタンスを生成して返すため、実際にコードを実装する必要はなく、単に null を返せばよい。しかし、図 7 に示しているように、インスタンス生成時に引数を受け取る必要がある場合にはより複雑な実装が必要になる。アスペクトは、privileged（アスペクトを他のクラスの内部情報にアクセスできるようにする）として宣言されなければならない。しかし、privileged として宣言されたアスペクトがアスペクトのインスタンスを生成できるようになることは、恐らくバグであ

り、将来のリリース（1.2 以降）では利用できない可能性がある。その場合には、図 7 の実装方法よりもさらに複雑な方法が必要になる。

```
privileged
aspect PointLogging extends InstanceLevelAspect {
  private PrintWriter out;
  static PointLogging newInstance(OutputStream out) {
    PointLogging instance = new PointLogging();
    instance.out = new PrintWriter(out,true);
    return out;
  }
  // 省略
}
```

図 7 newInstance が引数を受け取る場合

2 つ目のルールは、どのクラスのインスタンスがインスタンスレベルアスペクトの適用対象であるのかを任意に示すことである。そのためには TargetCheck アスペクトを継承し、抽象ポイントカットである adviceTarget に具体的な実装を与える必要がある。典型的には、図 6 から分かるように、target を用いれば良い。

また、どのインスタンスがアスペクトの適用対象であるのかを指定するためには、あらかじめ定義されている addObject を用いる。逆に、インスタンスを適用対象から外すためには removeObject を用いる。

PointLogging の使用例とその実行結果をそれぞれ図 8 と図 9 に示す。

```
PointLogging logging1 = PointLogging.newInstance();
PointLogging logging2 = PointLogging.newInstance();
Point p = new Point(1, 2); p.setX(10);
logging1.addPoint(p); p.setY(20);
logging2.addPoint(p); p.setX(100);
```

図 8 PointLogging の使用例

```
x = 10
y = 20
y updated - aj.PointLogging@9931f5
x = 100
x updated - aj.PointLogging@9931f5
x updated - aj.PointLogging@1f1fba0
```

図 9 図 8 の実行結果

AspectJ では、アスペクトのインスタンスは任意（new を使っては）に生成できないが、InstanceLevelAspect の実装では、Class.newInstance() を用いることによりインスタンスを生成している。生成されたインスタンスの一覧は AspectInstanceRepository アスペクト（図には示していない）によって保持される。

図 10 は、newInstance メソッドによってアスペクトのインスタンスが二つ生成されている時の after アドバイスの処理の流れを示している。アドバイスの実行は、adviceexecution によってインターセプト（around アドバイス）された後、生成されて保持されているアスペクトのインスタンスすべてに対して、proceed が呼び出される。proceed によって実行されたアドバイスは、さらにインターセプトされる。そこでは、アドバイス実行中のアスペクトインスタンスに対して、アドバイス適用対象のインスタンスにアドバイスが適用可能かどうかをチェックしている。適用可能であれば、アドバイスが実行される。可能でなければ、アドバイスは実行されない。なお、前述のように、around アドバイスの場合には、何もしない代わりに元々のジョインポイントを呼び出す必要がある。

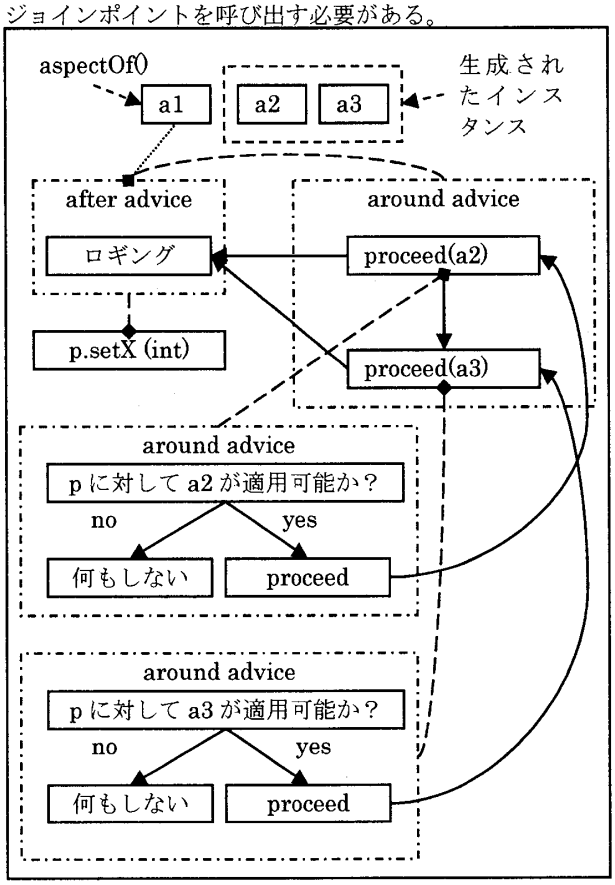

図 10 処理の流れ

InstanceLevelAspect は、インスタンスレベルアスペクトに要求される機能のほとんどを提供するが、アスペクトのインスタンスを生成するためのメソッドを明示的に実装する必要があり、また、アスペクト適用対象も明示的に指定しなければならないという制限がある。これらの制限は、インスタンスレベルアスペクトが直接サポートされていれば、必要のない実装や指定である。次節では、タグ情報とコード生成を基にし、より自然にインスタンスレベルアスペクトを実現するシステムを述べる。インスタンスレベルアスペクトとして扱われるクラスは通常のクラスであるため、new を使ってインスタンス化でき、また、アスペクト適用対象の指定も必要ない。

4. コード生成による実装

提案システムでは、通常のクラスにタグ付けされた情報を基にして、そのクラスをインスタンスレベルアスペクトとして扱えるようにするコードを生成する（図 11）。

たとえば、図 12 に示している PointLogging クラスをジェネレータへ入力すると、図 13 に示しているような PointLoggingAspect のコードが生成される。

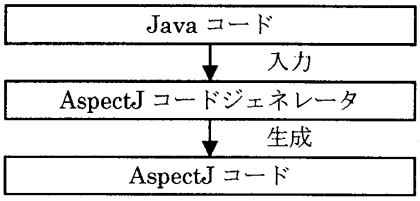

図 11 コード生成の流れ

```
/**
 * @instancelevel-aspect
 */
public class PointLogging {
 /**
  * @advice after() : call( void Point.setX(int) )
  */
 private void xUpdated() {
  System.out.println(" x updated " + this);
 }
 /**
  * @advice after() : call( void Point.setY(int) )
  */
 private void yUpdated() {
  System.out.println(" y updated " + this);
 }
 public void addPoint (Point p) { addObject(p); }
}
```

図 12 PointLogging クラス

```
privileged public aspect PointLoggingAspect {
 private List aspectList = new ArrayList();
 declare parents:
  PointLogging implements InstanceLevelAspect;          } (a)
 after() returning(PointLogging aspect) :
  call( PointLogging.new(..) ) {
   aspectList.add(aspect);
  }                                                     } (b)
 after() : call( void Point.setX(int) ) {
  Object target = thisJoinPoint.getTarget();
  PointLogging[] aspects = getAspects();
  for(int i = 0 ; i < aspects.length; i++) { //(1)
   if ( aspects[i].isApplied(target) ) { //(2)
    aspects[i].xUpdated(); //(3)
   }
  }                                                     } (c)
 }
 // setX の場合と同様
 after() : call( void Point.setY(int) ) {/* 省略 */}
 private PointLogging[]getAspects() {
  // aspectList を PointLogging[]に変換
  // … 省略
  return aspects;
 }
}
```

図 13 生成されたコード

instancelevel-aspect タグ（@instancelevel-aspect）はこのクラスをインスタンスレベルアスペクトとして扱えるようにすることを示す。ポイントカットの定義の方法は、advice タグ（@advice）を用いる以外は、AspectJ と同様である。AspectJ との違いは、アドバイスには通常のメソッドが対応する点である。addObject メソッド（と removeObject メソッド）の役割については、3 節に示したものと同様である。

図 13 から分かるように、アスペクト化されるクラス名の後ろに"Aspect"が追加されたアスペクトが自動的に生成される。(a)アスペクト化されるクラスは、inter-type 宣言を用いて、InstanceLevelAspect インタフェースが実装される。このインタフェースは、addObject や removeObject を定義している。(b)アスペクト化されたクラスのインスタンスは、aspectList フィールドに保持される。(c) @advice タグで指定されたアドバイスは、直接 AspectJ におけるアドバイスに変換される。各アドバイス内では、以下の処理を行っている。

(1) 生成されたアスペクトインスタンスのそれぞれに対して、
(2) 現在アドバイスの適用対象となっているオブジェクト（target）にアドバイスが実行できるかどうかをチェック（isApplied）し、
(3) 実行できる場合には、@advice タグが付けられたメソッドがアドバイスとして実行される。

3 節で述べた方法と違い、ここでの実装は通常のクラスをアスペクトとして扱っているため、より自然に使用できる。インスタンスの生成は特別なメソッド（newInstance）を用いる必要はなく、new を使えばよい。また、明示的にアスペクトの適用対象を指定する必要もない。

5. まとめ

本稿では、ロギングを例に、AspectJ におけるアスペクトの機能不足を指摘した。これらの機能不足を補うために、二つのアプローチを提案した。1 つは、再利用可能なインスタンスレベルアスペクトの実装であり、このアスペクトの使用方法と実装方法を詳しく述べた。もう 1 つは、タグ情報とコード生成を基にしたアプローチであり、通常のクラスをインスタンスレベルアスペクトとして用いるためのコード生成システムである。これらのアプローチを用いることにより、AspectJ では実装するのが複雑になるようなロギング機能を容易に実装できることを示した。

参考文献

[1] AspectJ. http://eclipse.org/aspectj/
[2] Hridesh Rajan and Kevin Sullivan,"Eos: Instance-Level Aspects for Integrated System Design", In the proceedings of the ESEC/FSE 2003.
[3] AspectJ Tip: Instance-level Aspects. http://noselab.ise.osaka-sandai.ac.jp/~asato/doc/aspectj-instance.html