アスペクト指向プログラミングで、モジュール性を改善する

Java言語の世界にAOPをもたらしたAspectJ

Nicholas Lesiecki

2002年 1月 01日

新しいプログラミング手法であるアスペクト指向プログラミングを利用すれば、プログラマーは横断的関心 (crosscutting concerns、つまり、ロギングのように、典型的な責任の分担を越えて横断する動作) をモジュール化できます。AOPはアスペクト (aspect、様相) の考え方を採用しています。これは、多数のクラスに作用する動作を、再使用可能なモジュールとしてカプセル化することです。Xerox PARCが最近リリースしたAspectJのおかげで、Java開発者はAOPモジュール化の利点を大いに利用できるようになりました。この記事ではAspectJについて紹介し、AspectJがもたらす設計上の利点を説明します。

典型的なプログラムの動作はただ1つのプログラム・モジュールに自然に収まり切らない、いや、互いに密接に関連したプログラム・モジュールのセットにさえ収まり切らないことが多い - こんな認識から、アスペクト指向プログラミング (Aspect-oriented programming、AOP) が生み出されました。AOPの先駆者たちは、このようなタイプの動作が特定のプログラミング・モデルの中で典型的な複数の役割分割をまたがって横切ることから、このような動作を横断 (crosscutting) と名付けました。たとえばオブジェクト指向プログラミングの場合、クラス単位でモジュール化するのが自然であり、複数のクラスにまたがる関心事が横断的関心です。横断的関心の例には、ロギング、コンテキストに依存したエラー処理、パフォーマンス最適化、デザイン・パターンなどがあります。

横断的関心を処理するコードに取り組んだことのある読者であれば、モジュール性の欠如からくる問題を認識しているでしょう。横断動作のインプリメンテーションはあちこちに散らばっているので、こうした動作を設計、インプリメント、および変更することは開発者にとって至難の業です。たとえば、ロギングを行うコードは、ロギング以外の処理を主に担当するコードと複雑に絡み合います。扱おうとしている対象の関心がどれほど複雑なものか、またどれほど広範囲かによって、コードの「もつれ」はささいであったり、あるいは重大であったりします。アプリケーションのロギング・ポリシーを変えようとすると、何百もの編集作業を覚悟しなければならないかもしれません。仮にそれが可能であっても、きわめて煩雑な作業です。これとは対照的なのが、アスペクト指向プログラミングの基本的なケースです。AspectJの作成者たちが書いた「Aspect-Oriented Programming」というタイトルの記事では、768行のプログラムをパフォーマンス最適化したら35,213行に膨らんだケースが取り上げられています。アスペクト指向の技法を

使ってこのコードを書き直したところ、パフォーマンス最適化機能をほとんど保持したまま、わずか1,039行に縮まりました。

AOPはオブジェクト指向プログラミングを補います。AOPにおけるモジュール化は、広範囲にわたる横断的関心のインプリメンテーションをただ1つの単位にまとめ上げます。このような単位をアスペクト (様相) と呼び、ここからアスペクト指向プログラミングの名が生まれました。アスペクト・コードを区分けしていくと、横断的関心は扱いやすくなります。システムのさまざまなアスペクトをコンパイル時に変更、挿入、または除去できます。再使用さえ可能です。

実例から学ぶ

アスペクト指向プログラミングとは一体どんなものかをもっと理解するために、Xerox PARCのJavaプログラム言語のアスペクト指向拡張であるAspectJについて見ていきましょう。今回の例では、AspectJを使ってロギングを処理します。サンプルは、サーバー側Javaコンポーネントのテストを単純化するCactusオープン・ソース・フレームワークから取られました。このフレームワークへの開発貢献者たちは、フレームワーク内のすべてのメソッド呼び出しをトレースすることにより、デバッグを支援することにしました。Cactusバージョン1.2はAspectJを使わずに書かれ、メソッドは、たとえばリスト1のようになります。

リスト1. 各メソッドに手書きで挿入されるログ呼び出し

```
public void doGet(JspImplicitObjects theObjects) throws ServletException
{
  logger.entry("doGet(...)");

  JspTestController controller = new JspTestController();
  controller.handleRequest(theObjects);

  logger.exit("doGet");
}
```

このプロジェクトのコード規則の1つとして、すべての開発者は自分の書くあらゆるメソッドに上記のような行を挿入するよう求められました。さらに開発者たちは、それぞれのメソッドのパラメーターをすべてログに記録するよう求められました。プロジェクトの管理者が各コードを入念にチェックしなければ、こうした規則の徹底は困難です。バージョン1.2では、約80の異なるログ呼び出しが15個のクラスにまたがって存在しました。バージョン1.3のフレームワークでは、80の呼び出しがたった1つのアスペクトに置き換えられました。このアスペクトは、パラメーターとその戻り値、およびメソッドの入り口と出口を自動的にログに記録します。リスト2は、このアスペクトをかなり単純化して示しています (たとえば、パラメーターと戻り値のロギングが省略されています)。

リスト2. 各メソッドに自動的に適用されるログ呼び出し

```
public aspect AutoLog{

  pointcut publicMethods() : execution(public * org.apache.cactus..*(..));

  pointcut logObjectCalls() :
    execution(* Logger.*(..));

  pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

  before() : loggableCalls(){
    Logger.entry(thisJoinPoint.getSignature().toString());
  }

  after() : loggableCalls(){
    Logger.exit(thisJoinPoint.getSignature().toString());
  }
}
```

このサンプルを分析して、アスペクトとは何かを調べましょう。まず最初に気付くのは、アスペクト宣言です。アスペクトの宣言はクラスの場合と似ていて、クラスと同様にJavaタイプを定義します。宣言に加えて、このアスペクトにはpointcut とadvice が含まれています。

pointcutとjoin point

pointcutを理解するには、まずjoin point について知る必要があります。join point (結合点、継ぎ目) とは、プログラムの実行局面の中で、適切に定義された (well-defined) ポイントを表します。AspectJにおける 典型的なjoin pointには、メソッド呼び出し、クラス・メンバーへのアクセス、例外ハンドラー・ブロックの実行などがあります。join pointの中に、他のjoin pointが含まれることもあります。たとえば、1つのメソッド呼び出しの結果として、そこから値が戻される前に、他の複数のメソッドが呼び出される場合があります。次にpointcutとは、定義された基準に基づいて一連のjoin pointを集める言語構造体です。例の中で最初のpointcutである`publicMethods` は、`org.apache.cactus` パッケージ内のすべてのパブリック・メソッド実行を選び出します。`execution` はプリミティブpointcutです (ちょうど`int` がプリミティブJavaタイプであるのと同様)。このpointcutは、括弧内に定義されたシグニチャーに一致するすべてのメソッドの実行を選び取ります。シグニチャーにはワイルドカードを使用でき、この例の中では複数のワイルドカードが使用されています。2番目のpointcutは`logObjectCalls` であり、`Logger` クラス内のすべてのメソッド実行を選び出します。3番目のpointcutは`loggableCalls` で、前の2つのpointcutを`& !` を使用して結合します。つまり、`org.apache.cactus` の中で、`Logger` クラス内のものを除くすべてのパブリック・メソッドを選びます。(ログ・メソッドそのものをロギングすると、無限の再帰が発生します。)

advice

ログに記録すべきポイントを定義した後、アスペクトはadviceを使って実際のロギングを行います。adviceとは、join pointの前 (before)、後 (after)、または前後 (around) に実行されるコードです。adviceはpointcutと関連付けて定義します。たとえば「ログに記録するすべてのメソッド呼び出しの後で、このコードを実行せよ」という言い方をします。したがって、adviceは次のようになります。

```
before() : loggableCalls(){
    Logger.entry(thisJoinPoint.getSignature().toString());
}
```

このadviceが使用するLoggerクラスの入り口と出口メソッドは、以下のようなものです。

```
public static void entry(String message){
   System.out.println("entering method " + message);
}
```

この例で、ロガーに渡されるStringはthisJoinPointから派生します。thisJoinPointは特別な自己反映的オブジェクトで、join pointが実行されるランタイム・コンテキストを察知できるようにします。実際にCactusで使われているアスペクトでは、ログ対象の各メソッド呼び出しに渡されるメソッド・パラメーターを検索するために、adviceがこのオブジェクトを使用します。このロギング・アスペクトをコードに適用すると、メソッド呼び出しの結果はたとえば以下のようになります。

リスト3. AutoLogアスペクトの出力

```
entering method: void test.Logging.main(String[])
entering method: void test.Logging.foo()
exiting method: void test.Logging.foo()
exiting method: void test.Logging.main(String[])
```

around advice

Cactusの例では、before()およびafter()というadviceが定義されています。第3のタイプのadviceはaround()です。これを使用すれば、アスペクト作成者は特別なproceed()構文を使って、join pointを実行するかどうか、また、いつ実行するかに作用できます。以下の (ちょっと意地悪な) adviceは、Helloクラス内のsayメソッドの実行を運命の手にゆだねます。

```
void around(): call(public void Hello.say()){
  if(Math.random() > .5){
      proceed();//go ahead with the method call
  }
  else{
      System.out.println("Fate is not on your side.");
  }
}
```

AspectJを使った開発

アスペクト・コードがどんなものか少し理解できたところで、アスペクト作成作業について簡単に説明しましょう。つまり、「上記のコードを実際に動作させるにはどうすればよいか」という質問にお答えします。

通常のクラス・ベースのコードに作用するアスペクトの場合は、対象のコードの中にそれらのアスペクトを織り込む(weave)必要があります。AspectJを使ってこれを行うには、ajcコンパイラーを使ってクラスとアスペクトをコンパイルしなければなりません。ajcはコンパイラーまたはプリコンパイラーとして実行でき、有効な .classファイルまたは .javaファイルを生成します。その後、(わずかなランタイムJARを追加すれば) これらのファイルをいずれかの標準的なJava環境でコンパイルして実行できます。

AspectJを使ってコンパイルするには、特定のコンパイルに含めるすべてのソース・ファイル (アスペクトとクラスの両方) を明示的に指定する必要があります。ajcはjavacとは異なり、関連するインポートのクラスパス検索を単純化してくれません。それも道理にかなっています。という

のも、標準的なJavaアプリケーション内のそれぞれのクラスは独立したコンポーネントのような動作をします。クラスが正しく動作するには、そのクラスが直接参照する一連のクラスがただ存在すればいいだけです。アスペクトは、複数のクラスにまたがる集約動作を表します。したがって、AOPプログラムは一度に1つのクラスずつではなく、集合体としてコンパイルする必要があります。

さらに、特定のコンパイルに含めるファイルをすべて指定することによって、システム内のさまざまなアスペクトをコンパイル時に追加または削除できます。たとえば、先程のロギング・アスペクトをコンパイルに追加したりコンパイルから除去することによって、アプリケーション作成者は、Cactusフレームワークからメソッド・トレース機能を追加または除去することができます。

AspectJはオープン・ソース

Xeroxは、Mozilla Public LicenseのもとでAspectJを入手できるようにしています。オープン・ソース支持者にとって、これは歓迎すべきことです。さらに、近い将来にAspectJを採用したいすべての人にとっても、歓迎すべきことです。製品は無料で、重大なバグが存在すると思われる場合には、ソース・コードを吟味できるからです。さらに、オープン・コード・ベースのAspectJソース・コードは、市場で普及する前に、貴重な批評をコミュニティーから受けてきました。

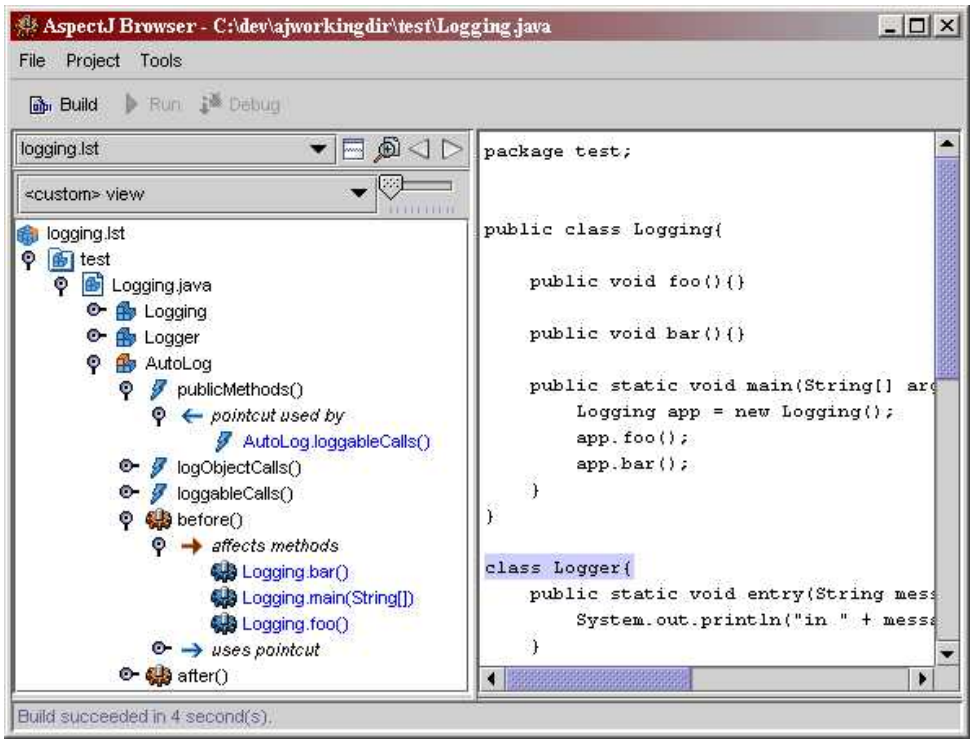
AspectJの現行バージョンでは、重要な制限があります。コンパイラーは、ソースが存在するコードの中にしかアスペクトを織り込むことができません。言いかえると、ajcを使用して、プリコンパイル済みのクラスにadviceを追加することはできません。AspectJチームはこの制限をほんの一時的なものと考えています。AspectJのWebサイトによると、将来のバージョン (公式には2.0) ではバイトコードの変更も可能になるとのことです。

ツール・サポート

AspectJのリリースには、いくつかの開発ツールも付属しています。こうして、AspectJの作成者たちは開発者にフレンドリーなAspectJを目指していますから、AspectJの将来は有望です。ツール・サポートは、アスペクト指向のシステムにとって特に重要です。プログラム・モジュールが、見ず知らずの他のモジュールから作用を受けることを可能にするからです。

AspectJに付属してリリースされている重要なツールの中には、アスペクトが他のシステム・コンポーネントとどのように関連しているかを視覚的に表すグラフィカルな構造ブラウザーも含まれます。この構造ブラウザーは、広く普及しているIDEのプラグインとして、あるいはスタンドアロン・ツールとして利用できます。図1は、先程のロギング・サンプルを表したビューです。

図1. AutoLogによるadvice対象のメソッドなどを表示する、AspectJ付属のグラフィカル構造ブラウザ



構造ブラウザーとコア・コンパイラーに加えて、アスペクトを認識できるデバッガー、javadocツール、Antタスク、EmacsプラグインなどをAspectJ Webサイトからダウンロードできます。

言語の特徴に話を戻しましょう。

クラス構造への作用: イントロダクション

pointcutsやadviceを使用すれば、プログラムの実行に動的に作用を与えることができます。一方、イントロダクション（導入、採用）を使用すれば、アスペクトはプログラムの静的な構造を変えることができます。アスペクトはイントロダクションを使って、新しいメソッドや変数をクラスに追加したり、クラスがあるインターフェースをインプリメントすることを宣言したり、例外を検査から非検査に変換することができます。

イントロダクションの例

ここで、永続データのキャッシュを表す1つのオブジェクトがあるとしましょう。データの「新鮮さ」を計測するために、オブジェクトにタイム・スタンプ・フィールドを追加して、そのオブジェクトと元の記憶装置との同期が保たれているかどうかを簡単に検出できるようにします。しかし、オブジェクトはビジネス・データを表しているので、この詳細な機能をオブジェクトそのものから切り離すのが得策です。AspectJを使えば、リスト4に示されているような構文を使用して、既存のクラスにタイム・スタンプ機能を追加することができます。

リスト4. 既存のクラスに変数とメソッドを追加する

```
public aspect Timestamp {

  private long ValueObject.timestamp;

  public long ValueObject.getTimestamp(){
      return timestamp;
  }

  public void ValueObject.timestamp(){
      //"this" refers to ValueObject class not Timestamp aspect
      this.timestamp = System.currentTimeMillis();
  }
}
```

イントロダクション (導入) されるメソッドとメンバー変数は、通常のクラス・メンバーの場合とほとんど同様に宣言できます。唯一の違いは、どのクラスでそれらを宣言するか明示しなければならない点です (この場合は`ValueObject.timestamp`)。

ミックスイン・スタイルの継承

AspectJでは、クラスだけでなくインターフェースにもメンバーを追加でき、C++ 風のミックスイン・スタイルの継承が可能です。タイム・スタンプ・コードをさまざまなオブジェクトで再利用するためにリスト4のアスペクトを一般化したい場合には、`TimestampedObject` というインターフェースを定義し、イントロダクションを使用して、同じメンバーや変数を (具象クラスにではなく) インターフェースに追加することができます。リスト5をご覧ください。

リスト5. インターフェースに動作を追加する

```
public interface TimestampedObject {
    long getTimestamp();

    void timestamp();
}
//and
public aspect Timestamp {

    private long TimestampedObject.timestamp;

    public long TimestampedObject.getTimestamp(){
        return timestamp;
    }

    public void TimestampedObject.timestamp(){
        this.timestamp = System.currentTimeMillis();
    }
}
```

次に、`ValueObject` に新しいインターフェースをインプリメントさせるために、`declare parents` 構文を使用することができます。他のAspectJ型式と同じように、`declare parents` は一度に複数のタイプに適用できます。

```
declare parents: ValueObject|| BigValueObject implements TimestampedObject;
```

こうして`TimestampedObject` がサポートする操作を定義したら、pointcutとadviceを使って、適切な状況が発生したときにタイム・スタンプを自動的に更新させることができます。この機能は事

前にコンテキストにアクセスする方法を示しているので、次のセクションで説明することにします。

AspectJのその他の特徴

pointcutを使用すれば、タイム・スタンプされるオブジェクトがどんな状況で自らのタイム・スタンプを更新すべきか、簡単に定義できます。リスト6のようなコードを`Timestamp`に追加することによって、設定(setter)メソッドを呼び出すたびにオブジェクトのタイム・スタンプが更新されます。

リスト6. 事前にコンテキストにアクセスする

```
pointcut objectChanged(TimestampedObject object) :
             execution(public void TimestampedObject+.set*(..)) &&
             this(object);
/*TimestampedObject+ means any subclass of TimestampedObject*/

after(TimestampedObject object) :  objectChanged(object){
        object.timestamp();
}
```

このpointcutは`after()` adviceが使用する引数を定義しています。ここでは、設定メソッドを呼び出される`TimestampedObject`です。`this()` pointcutは、現在実行中のオブジェクトが括弧内に定義されたものと同じ型になるような、すべてのjoin pointを識別します。このほかにも、メソッド引数、メソッドがthrowした例外、メソッド呼び出しのターゲットなど、いくつかの種類の値をadviceの引数にすることができます。

カスタム・コンパイル・エラー

カスタム・コンパイル・エラーは、AspectJの最も便利な機能の1つです。たとえば、あるサブシステムを分離して、クライアント・コードが中間処理を通り抜けた後に処理(worker)オブジェクトに達するよう設計するとします (これは、Facadeデザイン・パターンで発生し得る状況です)。`declare error`または`declare warning`構文を使用すれば、リスト7のように、join pointが出現した場合のajcコンパイラーの応答をカスタマイズできます。

リスト7. カスタム・エラーを定義する

```
public aspect FacadeEnforcement {

  pointcut notThruFacade() : within(Client) && call(public * Worker.*(..));

  declare error : notThruFacade():
    "Clients may not use Worker objects directly.";
}
```

`within` pointcutは`this()`と似ていますが、唯一の違いはajcがコンパイル時にこれを完全に検出することです (ほとんどのpointcutは、ランタイム情報に基づいて区別されます)。

エラー処理

私はJava言語における検査例外の価値を認識しています。しかし、何らかの改善、たとえば「この例外をランタイム例外に変更せよ」といった簡単なコマンドがあればいいと感じたことが、何度もあります。多くの場合、私の書くメソッドには例外に対する意味のある応答が存在しません - おそらく、私のメソッドを使ってくれるユーザーにとっても同様でしょう。私は例外を捨て去り

たくないのですが、かと言って、すべての呼び出し側を追跡して例外の存在を確認したくもありません。try/catchブロックを使って巧みにこれを行う方法もありますが、AspectJの`declare soft`のエレガントさにはとても及びません。リスト8のクラスは、あるSQL操作を試みます。

リスト8. 検査例外を使用するクラス

```
public class SqlAccess {

    private Connection conn;
    private Statement stmt;


    public void doUpdate(){
        conn = DriverManager.getConnection("url for testing purposes");
        stmt = conn.createStatement();
        stmt.execute("UPDATE ACCOUNTS SET BALANCE = 0");
    }

    public static void main(String[] args)throws Exception{
        new SqlAccess().doUpdate();
    }
}
```

もしAspectJを使わなかったり、各メソッド・シグニチャーで例外を宣言しない場合には、(JDBC APIのほとんどのメソッドからthrowされる) 検査例外`SQLException`を扱うために、try/catchブロックを挿入しなければならないでしょう。AspectJでは、以下のような内部アスペクトを使って、これを`org.aspectj.lang.SoftException`として自動的に再スローできます。

リスト9. 例外をsoftにする

```
private static aspect exceptionHandling{
  declare soft : SQLException : within(SqlAccess);

  pointcut methodCall(SqlAccess accessor) : this(accessor)
    && call(* * SqlAccess.*(..));

  after(SqlAccess accessor) : methodCall  (accessor){
    System.out.println("Closing connections.");
    if(accessor.stmt != null){
        accessor.stmt.close();
    }
    if(accessor.conn != null){
        accessor.conn.close();
    }
  }
}
```

このpointcutとadviceは、`SQLAccess`クラス内の各メソッドの後で接続とステートメントを閉じます (例外をthrowしたか、正常に値を戻したかにかかわらず)。ただ1つのメソッドのためにエラー処理アスペクトを使うのはやり過ぎでしょうが、接続およびステートメントを使用するメソッドを他にも追加する場合には、このエラー処理ポリシーをそれらにも適用できます。このようにアスペクトを新しいコードに自動的に適用できることは、AOPの持つ強みの1つです。新しいコードの作成者たちは、アスペクトに参加する際に横断動作を考慮する必要がありません。

結論

AspectJを利用する価値はあるのでしょうか。Grady Booch氏によると、アスペクト指向プログラミングは、ソフトウェアの設計および作成の方法を根本的に変えつつある3つの大きな動きの一角

です。(参考文献の "Through the Looking Glass" を参照してください。)私も同感です。AOPは、オブジェクト指向その他のプロシージャー型言語がこれまで解決できなかった種類の問題に取り組んでいます。私自身、AspectJを調べ始めて2、3週間のうちに、プログラミング上の基本的な制限 (と私が思っていたもの) がエレガントかつ再使用可能な形で解決される様子を見て取りました。私がオブジェクトを使い始めて以来知った抽象化うちで、AOPは最もパワフルなものだと言っても過言ではありません。

もちろん、AspectJには学習曲線が付き物です。他の言語や言語拡張と同様に、AspectJを使いこなすために理解しなければならない難解な点がいくつかあります。しかし私自身、開発者向けガイドを読んでサンプルをいくつか検討した後、役立つアスペクトを自分で構築できるようになりました。AspectJは自然です。私たちのプログラミング知識を新たな方向に拡張するというよりも、まるでプログラミング知識のすき間を埋めてくれるかのようです。AspectJツールの中にはとっつきにくそうなものもありますが、私としては、大きな問題に出くわしたことはありません。

モジュール化されざるをモジュール化するAspectJのパワーは、ただちに使い始めるに値すると私は思います。読者のプロジェクト (または企業) が実生産にAspectJを採用する準備ができていない場合には、デバッグや契約施行の段階で簡単にAspectJを使い始めることができます。読者自身のためにも、この拡張言語をぜひチェックしてください。

関連トピック

- asod.net は、AOP一般に関する主要な情報を提供しています。このサイトには他の先駆的AOPサイトへのリンク、他の言語でのアスペクト実装、アスペクト風のソース修正、アスペクト理論に関する記事があります。
- JavaWorld はAOPを紹介する短いシリーズ "I want my AOP" (僕もAOPが欲しい) を提供しています。
- Eric Allenによるシリーズ記事「Javaコードの診断」は、整合性のない方法で横断的関心を処理することに由来する多くの「バグ・パターン」について解説しています。Allenは、OO技法を使ってこの問題を最小限に食い止める方法について述べています。"バグ・パターン" では、OOPの世界では避けられないコードの繰り返しが原因で発生する、よく見られるタイプのバグについて説明しています。"「ヌル・フラグ」バグ・パターン" では、例外に関連してAspectJに言及しています。
- この記事で取り上げたロギングの例は、JakartaのCactus プロジェクトのものです。
- developerWorks Javaテクノロジー・ゾーン には、Javaに関するその他の参考文献があります。