2004－SE－146　(8)
2004／11／26

アスペクト指向プログラミングにおけるアスペクトの単体テスト

山崎雄大†，櫻井孝平‡，橋浦弘明‡，古宮誠一‡

ソフトウェア開発の現場では，ソフトウェアの信頼性を確保するためにテストを行う．アスペクト指向プログラミングを用いたソフトウェア開発でも，アスペクトをテストする必要がある．しかしアスペクトをテストしようとすると，エラーの発生箇所が特定できないといった問題が発生する．アスペクトのテストに関するいくつかの問題は，アスペクトの単体テストを行うことで解決できる．しかし，単体で実行することが想定されていないアスペクトをどう単体テストすればいいかという問題がある．

本研究ではアスペクトの単体テストの問題を明確にし，アスペクトの単体テストの問題を解決する手法と，そのテストを支援するツールを提案する．

A Method for Unit Testing of Aspect-Oriented Programs

Yudai Yamazaki，Kouhei Sakurai,
Hiroaki Hashiura and Seiichi Komiya

Software testing is required the programs implemented by using aspect-oriented programming technology to guarantee reliability of them. In this case, however, there are problems such that it's not able to find where some errors in the program exist, when programmers test the program which uses aspects. It's able to resolve some problems by performing unit testing of them. However, there is no guide to perform unit testing of an aspect, because an aspect isn't intended to execute itself alone. The authors clarify some problems in case of unit testing of an aspect module, and a method for solving the problems, and propose a tool to support unit testing of the module.

1. はじめに

アスペクト 指向プログラミング (AOP [1]) を用いたソフトウェア 開発では，複数のモジュールを横断する関心事をアスペクトというモジュール にまとめることができる ．AspectJ に代表されるアスペクト指向プログラミング言語では，アスペクトは新たな言語機構として提供されており，他の複数のモジュールと結合(weaving)して動作する．

ソフトウェア開発のテスト工程の初期には各モジュール に対して単体テスト を行う．AOP を利用した場合，アスペクト が他のモジュールと結合しなければ動作しないために，アスペクト の単体テストを行う際に問題が発生することがある．また，そもそもアスペクトをどのような 方法でテストすべきかという 手法が確立していない．

本研究では，アスペクトのテストを行う際の問題を明らかにし，その解決のためにアスペクトを単体テストする機構を提案する．また，AspectJ のアスペクトはポイントカットとアドバイスといった言語構造で構成されていることから，これらを独立にテストする環境を提案し，ツールとして実装する．

以下，第 2 節ではアスペクトをテストする際の問題，第 3 節で本研究の提案，第 4 節で本研究の提案をサポートするためのツールの解説，第 5 節で今後のための議論をし，おわりにまとめを述べる．

2. アスペクトをテストする際の問題

本節では AspectJ を利用したプログラムの例とそのテストプログラムを紹介し，AOP におけるテストの問題を明らかにする．

† 芝浦工業大学
‡ 芝浦工業大学大学院
Shibaura-Institute of Technology

2.1 例題：MoveTracking アスペクト

図 1 の MoveTracking アスペクトは，点や線分といった図形要素の移動を横断的に監視するアスペクトである[1]．このアスペクトは図形要素が移動したときだけに再描画を促す，といったことに利用できる．

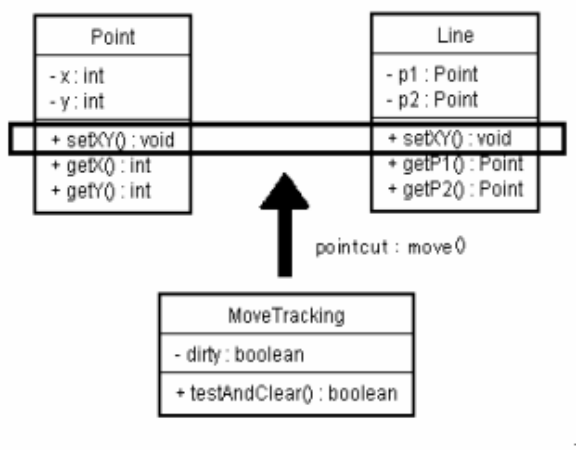

図 1. MoveTracking アスペクト

MoveTracking アスペクトは以下のような仕様になっている．

- static 変数として dirty フラグを持つ．
- Point クラスか Line クラスの setXY メソッドが呼び出された直後に dirty フラグを true にする．
- testAndClear メソッドが呼ばれると現在の dirty フラグの状態を返し，dirty フラグを false にする．

```
aspect MoveTracking{
    private static boolean dirty = false;
    public static boolean testAndClear(){
        boolean result = dirty;
        dirty = false;
        return result;
    }
    pointcut move() :
        call(void *.setXY(int, int));
    after() returning : move(){
        dirty = true;
    }
}
```

図 2. MoveTracking アスペクトのソース

[1] このアスペクトは aspectj project が配布する Programming Guide の 1．Getting Started with AspectJ，Production Aspects，Change Monitoring に書かれている例を少し書き換えたものである．

2.2 横断するテストケース

MoveTracking アスペクトは Point クラスと Line クラスの setXY メソッドを書き換える．このことを考慮してテストケースを書くと次のようになる．

```
1.  class PointTest extends TestCase{
2.      void testSetXY(){
3.          Point p = new Point(0, 0);
4.          p.setXY(1, 2);
5.          assertTrue(p.getX() == 1);
6.          assertTrue(p.getY() == 2);
7.          //.....
8.          // 以下，アスペクトのテスト
9.          assertTrue(MoveTracking
10.             .testAndClear());
11.     }
12.     // 以下 getX，getY のテストが続く
13. }
14. class LineTest extends TestCase{
15.     void testSetXY(){
16.         Line l = new Line(
17.             new Point(1, 2), new Point(3, 4));
18.         l.setXY(5, 6);
19.         assertTrue(l.getP1().getX() == 6);
20.         assertTrue(l.getP1().getY() == 8);
21.         assertTrue(l.getP2().getX() == 8);
22.         assertTrue(l.getP2().getY() == 10);
23.         //.....
24.         // 以下，アスペクトのテスト
25.         assertTrue(MoveTracking
26.             .testAndClear());
27.     }
28.     // 以下 getP1，getP2 のテストが続く
29. }
```

このテストを行うため，MoveTracking アスペクトを Point クラスと Line クラスに結合している．

このテストの問題は，9 行目や 25 行目のように Point クラスと Line クラスのテストケースに，MoveTracking アスペクトのテストケースが横断していることである．この場合，アスペクトのテストケースを変更しようとすると，横断したテストケースを追わなければならず，テストケースの変更が困難になるなど，横断的関心事の問題がテストケースにも現れる．

2.3 結合を前提としたテスト

前出 の 問題 を 解決 するために ，MoveTracking アスペクトの仕様を図 3 のようなテストケースでテストすることを考える[2]．このテストケースでは MoveTracking アスペクトの仕様を以下の項目で検証する．

[2] このテストケースは JUnit フレームワークを利用して記述されている．

・ Point クラスのコンストラクタ，getX()および getY() の呼び出し後は，testAndClear()の結果が true にならない．
・ Point クラスの setXY()の呼び出し後は，testAndClear()の結果が true になる．
・ testAndClear()の呼び出し後に，続けて testAndClear()を呼ぶと false が返る．

```
1.  class MoveTrackingTest extends TestCase{
2.    void testPointMoveTracking(){
3.      Point p = new Point();
4.        assertFalse(MoveTracking.testAndClear());
5.        p.getX();
6.        assertFalse(MoveTracking.testAndClear());
7.        p.getY();
8.        assertFalse(MoveTracking.testAndClear());
9.        p.setXY(1, 2);
10.       assertTrue(MoveTracking.testAndClear());
11.       assertFalse(MoveTracking.testAndClear());
12.   }
13.   // 以下 Line を使ったテストが続く
14.   // ...
15.}
```

図 3. MoveTracking アスペクトのテスト例

2.4 結合を前提としたテストの問題

図 3 のようにアスペクトの結合を前提としでテストケースを書くと，以下のような問題が発生する場合がある．

・エラーの発生箇所が明確でない．

例えば図 3 の 6 行目でエラーを発見したとする．この場合，getX メソッドを呼び出したときに dirty を true にしてしまっていると考えられる．しかしもしかすると，Point クラスの getX メソッドが内部で setXY メソッドを使う仕様になっていることで 発生したエラーかもしれない．この問題は Point クラスの仕様を知らなければ解決しない．

つまり，このテストでは実際にどこでエラーが発生したかを特定できず，デバッグの際にエラーの発生箇所を特定する手間が発生する．

・アドバイスを実行させるために手間がかかる．

図 3 では，MoveTracking アスペクトのアドバイスを呼び出すために， Point クラスのオブジェクトを生成して getX，getY，setXY といったメソッドを呼び出すコードを書いている．

もし，アドバイスを呼び出すために，生成が困難なオブジェクトが必要になったり，呼び出すために多くのコードを書かなければならないメソッドが必要になったりすると，より多くの手間がかかることになる．

2.5 アスペクトの単体テスト

今までのような問題はアスペクトを単体テストによって解決できる．単体テストを行う動機には，テストの組み合わせの問題を回避する，エラーの発生箇所を特定する，テストを平行処理するなどが挙げられる [2]．図 3 のテストケースは 2.4 節で挙げたようにエラーの発生箇所を特定できていないので，アスペクトの単体テストがうまくできているとは言えない．これはアスペクトを横断するモジュールに結合した状態でテストしているために 発生する問題である．

アスペクトの単体テストについての研究に Zhao の[3]がある．この研究ではアスペクトのデータフローを解析してテストすべき箇所を特定し，プログラムとしての正当性のチェックを支援することを目的としている．

本研究ではこのアプローチとは別に，2.1 節で示したようなアスペクトの仕様をチェックするためのアプローチを提案する．

3. 本研究の提案

本研究では AspectJ のアスペクトを横断するモジュールに結合させず，アスペクトの構成要素をそれぞれテストすることをアスペクトの単体テストと定義する．アスペクトの構成要素は，図 4 のような分類となる[3]．

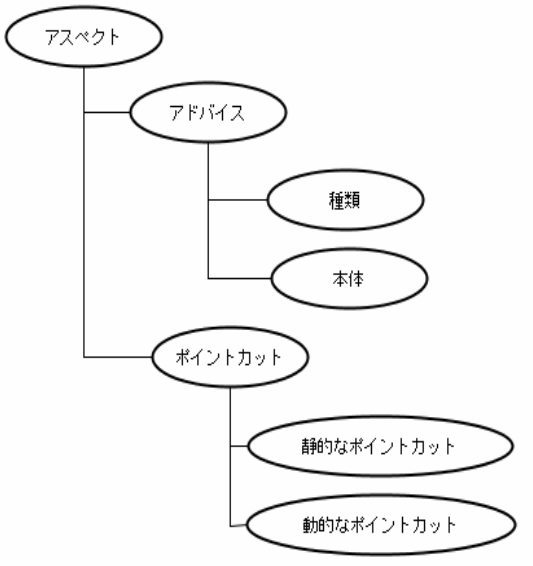

図 4. アスペクトの構成要素

[3] アスペクトの構成要素にはインタータイプ定義というものもあるが，そのテストについては本稿の提案には含めていないので，ここでは説明を省略する．

アドバイスには before，after，around の 3 種類があり，アドバイスの種類によってアドバイス本体が実行されるタイミングなどが違う．

```
// アドバイスの種類：after
after() returning：move(){
    // 以下，アドバイス本体
    dirty = true;
}
```

AspectJ はアドバイスの実行タイミングを動的なジョインポイントモデル により決定している．このモデルでは，プログラム実行時における特定の点（ジョインポイント）でポイントカット式の評価を行い，アドバイスを実行するかを決定する [4].

ジョインポイントには，対応するソースコード上，またはバイトコード上の位置が存在し，ジョインポイントシャドウ などと呼ばれている [5].

AspectJ が提供する原始ポイントカットには，ジョインポイントシャドウをコンパイル時に決定する静的なポイントカットと，実行時のジョインポイント に対して条件を付加する動的なポイントカットに分類することができる．

・静的なポイントカット

call，execution，get，set，initialization，preinitialization，staticinitialization，handler, within, withincode, adviceexecution

・動的なポイントカット[4]

cflow，cflowbelow，if，this，target，args

このような分類は，コンパイラの実装における最適化手法として利用される．本研究ではこの分類を，ポイントカットのテストを行うために利用する．

本稿では，アドバイス本体と静的なポイントカットのテストについて提案する．

3.1 アドバイス本体のテスト

AspectJ はアドバイス本体を明示的に呼び出し，実行する機構を提供していない．そのため，アドバイス本体をテストするには，アドバイス本体が実行される状況を作り出さなければならなくなり，第 2 節で紹介したような問題が発生する．そこで，本研究ではアドバイス本体を明示的に呼び出す機構を提案する．

例えば以下のようなコードでアドバイス本体をテストする．

```
class MoveTrackingTester{
    static void afterReturningMove(){
        // MoveTracking のアドバイス本体を
        // 呼び出す処理.
    }
}
void testAdvice(){
    assertFalse(MoveTracking
        .testAndClear());
    MoveTrackingTester
        .afterReturningMove();
    assertTrue(MoveTracking
        .testAndClear());
}
```

まず，アドバイスを static なメソッドに置き換えたテスタクラスを用意する．これにより，アドバイス本体をメソッドとして扱えるようになり，アドバイス本体のテストをメソッドのテストと同じように行うことができるようになる．

3.2 静的なポイントカットのテスト

本研究では，静的なポイントカットによって表現されるジョインポイントシャドウ の集合をテストするために，特定のジョインポイントシャドウとポイントカット 式を比較するテスト方法を提案する．

例えば以下のようなコードでポイントカットをテストする．

```
class Shadow{
    // ジョインポイントシャドウを
    // 表現するためのクラス.
}
class MoveTrackingTester{
    boolean testMovePointcut(Shadow s){
        // move ポイントカットと
        // ジョインポイントシャドウを
        // 比較する処理.
    }
}
void testPointcut(){
    // ジョインポイントシャドウを生成
    Shadow s1 = new Shadow(
        “call(void Point.setXY(int, int))”);
    Shadow s2 =new Shadow(
        “call(int Point.getX())”);

    // ポイントカットのテスト
    assertTrue(testMovePointcut(s1));
    assertFalse(testMovePointcut(s2));
}
```

[4] this，target，args は静的に決定する場合もある．ここでは動的な判定が発生する可能性があるものを動的なポイントカットとして分類している．

まず，ジョインポイントシャドウを表現するためのクラスを用意する．次に，以下のような仕様でテスト用のメソッドを，テストするポイントカット式ごとに用意する．

- テストケース となる ジョインポイントシャドウを引数として取る．
- 入力されたジョインポイントシャドウ がテスト対象となる ポイントカット に含まれるなら true を返す．
- 入力されたジョインポイントシャドウ がテスト対象となる ポイントカット に含まれないなら false を返す．

このメソッドを利用し，対象のポイントカットに含まれなければいけない 代表点を入力したときには true が，含まれてはいけない代表点を入力したときには false が返ってくることを確認することでテストを行う．

4. ツール

第 3 節で提案したテストを実施するには，アドバイス本体を呼び出すためのメソッドや，ジョインポイントシャドウ とポイントカット式を比較するメソッドを，独自に用意しなければならない．本研究では，これらのメソッドを持つテスト用クラスを自動生成するツールを開発している[5]．

4.1 アドバイス本体のテスト支援ツール

AspectJ はコンパイルによってアスペクトをクラスとして生成する[6]．このクラスにはアスペクトのフィールドとメソッドの他に，アドバイス本体がメソッドとして含まれている．このアドバイス本体に相当するメソッドが，アドバイス本体のテストに必要なメソッドである．

そこで本研究では，アドバイス本体に相当するメソッドを簡単に呼び出すためのテスト用クラスを，アスペクトから自動生成するツールを開発中である．このツールにアスペクトを入力すると，以下のようなメソッドを持つクラスを生成する．

```
public static void afterReturningMove(){
  MoveTracking.aspectOf()
    .ajc$afterReturning$MoveTracking$1$c0539092();
}
```

図 5. アドバイス本体のテスト用クラス

以下でツールの仕組みを説明する．

・シグネチャの取得

アドバイス 本体 に相当 する メソッド は AspectJ のコンパイラがアスペクトから生成し，指定されているジョインポイントから暗黙に呼び出されるものである．そのため，アドバイス本体に相当するメソッドを明示的に呼び出すことが想定されていない．

そこでまず，アスペクトから生成されるクラスを解析し，アドバイス本体に相当するメソッドのシグネチャ を取得する．図 5 の ajc$afterReturning$MoveTracking1c0539092 がアスペクトから生成されたクラスに含まれているアドバイス本体に相当するメソッドの名前である[7]．この場合，戻り値は void，引数は無しであるが，アドバイス本体の仕様によっては戻り値や引数が付く場合がある．

・アドバイスの名前付け

AspectJ では，プログラマがアドバイスに名前を付けることができない ．また，実際に AspectJ のコンパイラによって生成されるアドバイス 本体 に相当 する メソッド の名前は ajc$afterReturning$MoveTracking1c0539092 のようなわかりにくいものになっている．

そこでこのツールでは，アドバイス本体に相当するメソッドの名前を，わかりやすい名前で呼び出せるようにしている．例えば図 2 のアスペクトでは，アドバイスが after returning で指定され，ポイントカットに move が指定されているので，afterReturningMove というメソッド名を自動生成する．

アドバイスに指定されるポイントカット 式は，原始ポイントカットの記述がそのまま出てくる可能性もある．例えば以下のような場合がある．

```
after() returning :
    call(void *.setXY(int, int)){...}
```

この 場合 は， setXY という 記述 から，afterReturningSetXY などのメソッド名を自動生成することを考えている．

この他にも，アドバイス本体に相当するメソッドの名前を明示的に指定したいことがあるかもしれない．このような場合のために，アスペクト のソースコード 中にアドバイス本体に相当するメソッドの名前を記述することでアドバイス本体に相当するメソッドの名前を決定する機構も用意してある．

---

[5] 付録 A1 にこのツールが出力する予定のテスト用クラスのソースを載せている．

[6] 付録 A2 に MoveTracking アスペクトから生成された MoveTracking クラスの Java コードを載せている．

[7] これは AspectJ version1.2 の例である．

4.2 静的なポイントカットのテスト支援ツール

本研究が提案する静的なポイントカットのテストを行うには，ジョインポイントシャドウとポイントカット式を比較するメソッドをポイントカット ごとに 作成しなければならなかった．そこで本研究ではこのようなメソッドをアスペクトから自動生成するツールを開発中である．このツールにアスペクトを入力すると，以下のようなメソッドを持つテスト用クラスを生成する．

```
public static void testMovePointcut(Shadow s){
  TestPointcut pc = new PointcutParser(
    "call(void *.setXY(int,int))").parsePointcut();
  return pc.match(s);
}
```

ポイントカット式は AspectJ がアスペクトから生成するクラスには含まれないため，このツールではアスペクトのソースファイルからポイントカット式を取得する．取得したポイントカット式から，各ポイントカット専用の比較メソッドを生成する．

このツールが出力する比較メソッドは，引数に Shadow クラスのオブジェクトを入力する．Shadow クラスはジョインポイントシャドウを表現するためのクラスであり，本研究で提供する．

5. 議論

本節ではアスペクトの単体テストに関連するいくつかの話題や将来的な課題を議論する．

5.1 AOP フレームワークでの単体テスト

JBoss AOP[6]や AspectWerks[7]や AOP alliance[8]に準拠したフレームワークでは，オブジェクト指向言語自体を拡張することなくフレームワークとして AOP を実現している．これらのフレームワーク ではアスペクトもしくはアドバイス定義は 1 つのクラスとして定義される．従ってアドバイス定義に特定の名前がついており，pure Java のコードにより呼び出すことで，アドバイスの本体がテスト可能である．

本研究では AspectJ のようなアスペクト指向言語における アドバイス定義の意味に対して，テストを行う手法とツールによるサポートを提案している．またこれらのフレームワークではポイントカットが XML や annotation で記述されるため，ポイントカット式のテストを行うことが一般に困難であり，本研究の提案するポイントカット 式のテスト手法が有効であると考えられる．

5.2 他のテスト用フレームワークとの連携

本研究の提案するツールを利用してアスペクト の単体 テスト を行う上で， JUnit[9]や mock object[10]といったテスト用のフレームワークを利用することができる．本研究が提案するツールの典型的な利用方法は，JUnit が提供する TestCase クラスを利用してテストケースを記述し，そのテストケースでは第 4 節で示したツールによって出力されたテストのためのクラス(例えば MoveTrackingTester クラス)を介して，アスペクトをテストするコードが記述される．

また，アドバイスの本体のテストを行う際には，アドバイス定義の引数や文脈において他のクラスを利用する場合は mock object を利用することが 考えられる ． 例えば ，MoveTracking のアドバイスが Point クラスを利用するような場合，実際の Point オブジェクトを渡すかわりに mock object を渡すことで，他のクラスを利用しないアスペクト の完全な単体テストが可能になると考えられる．

本研究の提案するテストのための インターフェースは pure Java のクラスとして提供されるため，このような既存のフレームワークとの連携が可能である．

将来の展望として，アスペクトのテストの全貌を考えるとなると，アスペクトの結合テストについても考察する必要がある．例えば複数のアスペクトが一つのジョインポイント に作用する場合，どちらのアスペクトが先に反映されるかによって挙動が変わる場合がある．アスペクトの中で例外を投げるといった 例がこれにあたる．結合テストを考える際に，本稿が提案したアスペクト の単体テストが貢献するかどうかも議論の対象となる．

他にも，テスト・ファーストによるアスペクト の開発 とその 利点について 議論 したいと思っている．

6. おわりに

本稿では，アスペクトのテストの問題を解決するためにアスペクト の単体テストが有効であることを示し，アスペクトの単体テストの手法としてアドバイス本体のテストと静的なポイントカットのテストを提案した．また，本研究が提案するアスペクト の単体テストを支援するツールを開発していることを示した．しかし，アスペクトの単体テストの問題がすべて解決したわけではない．

本稿の提案では，アドバイスの種類と動的なポイントカットがテストされていない．これらのテストについても考察する必要がある．また，AspectJ にはアドバイスとポイントカットの他にもインタータイプ定義といった機能がある．このインタータイプ定義のテストについても考察する必要がある．

参考文献

[1] Gregor Kiczales，John Lamping，Anurag Mendhekar，Chris Maeda，Cristina Videira Lopes，Jean-Marc Loingtier，John Irwin，”Aspect-Oriented Programming,” Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP)，Finland. Springer-Verlag LNCS 1241．June 1997.

[2] Glenford J. Myers，長尾 真 監訳，松雄 正信 訳，“ソフトウェア・テストの技法,” 近代科学社，1980，192p，ISBN 4-7649-0059-9

[3] J．Zhao，“Data-Flow-Based Unit Testing of Aspect-Oriented Programs,” Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC’2003)，Dallas，Texas，USA，November 3-6，2003.

[4] Gregor Kiczales , Erik Hilsdale , Jim Hugunin , Mik Kersten , Jeffrey Palm , William G. Griswold, “An Overview of AspectJ,” Proceedings of the 15th European Conference on Object-Oriented Programming, p.327-353, June 18-22, 2001.

[5] Erik Hilsdale, Jim Hugunin, “Advice weaving in AspectJ,” Proceedings of the 3rd international conference on Aspect-oriented software development 2004, Lancaster, UK March 22 - 24, 2004.

[6] JBoss AOP：http://www.jboss.org

[7] AspectWerks：
http://aspectwerkz.codehaus.org

[8] AOP Alliance：
http://aopalliance.sourceforge.net

[9] JUnit, Testing Resources for Extreme Programming：http://junit.org

[10] Tim Mackinnon, Steve Freeman, Philip Craig, “Endo-Testing: Unit Testing with Mock Objects,” eXtreme Programming and Flexible Processes in Software Engineering - XP2000, Cagliari, Sardinia, Italy, June 21-23, 2000.

付録

A1. MoveTrackingTester クラスの Java コード

以下は第 4 節で述べた本研究が提供するツールにより出力されるテストのためのクラスの想定される出力例である．

```
public class MoveTrackingTester{
// アドバイス本体のテストのためのメソッド
    public static void afterReturningMove(){
      MoveTracking.aspectOf()
       .ajc$afterReturning$MoveTracking$1$c0539092();
    }
// ポイントカット式のテストのためのメソッド
    public static boolean testAfterReturningMovePointcut(
        Shadow s){
// テストのためのポイントカット式の構文木を構築
      TestPointcut pc = new PointcutParser(
        "call(void *.setXY(int,int))").parsePointcut();
// ポイントカット式の評価を s に対して行う
      return pc.match(s);
    }
}
```

A2. MoveTracking クラスの Java コード

以下は図 2 の MoveTracking アスペクトを AspectJ version1.2 のコンパイラによってコンパイルした結果，出力されるクラスをソースコードで表現したものである．

```
class MoveTracking {
   private static boolean dirty;
   public static boolean testAndClear() {
      boolean result = MoveTracking.dirty;
      MoveTracking.dirty = false;
      return result;
   }
// アドバイス本体の実装
   public final void
   ajc$afterReturning$MoveTracking$1$c0539092(){
      MoveTracking.dirty = true;
   }
// 以降は主にインスタンス管理のためのコード
   MoveTracking() { super(); }
   public static MoveTracking aspectInstance;
   public static MoveTracking aspectOf() {
      return MoveTracking.aspectInstance;
   }
   public static boolean hasAspect() {
      return MoveTracking.aspectInstance != null;
   }
   static {
      MoveTracking.aspectInstance =
         new MoveTracking();
      MoveTracking.dirty = false;
   }
}
```