クラスのインターフェースやその振る舞いに及ぼすアスペクトの影響の解析と可視化

堀江 倫大　　千葉 滋

東京工業大学大学院
情報理工学研究科 数理・計算科学専攻
{horie, chiba}@csg.is.titech.ac.jp

要 旨

アスペクト指向プログラミングでは、アスペクトの定義を見なければクラスの振る舞いを正確に理解することはできない。これは obliviousness と呼ばれるアスペクト指向の性質である。この obliviouness の問題を解決するために、アスペクトに影響を及ぼされたクラスのインターフェースやその振る舞いを解析し可視化するツール AspectScope を本論文は提案する。 AspectScope はアスペクト指向を用いて開発された既存プログラムの修正に主に使用する。既存のツール AJDT とは異なり、AspectScope はアスペクトの影響を間接的に受けるクラスにもその影響を表示する。これにより、カプセル化を保持したままアスペクトの影響をクラスに反映させることができる。AspectScope は outline ビューと javadoc 閲覧機能を提供する。また、 AspectScope を通して開発者は影響表示の細かな制御を行うことができる。

1　はじめに

アスペクト指向プログラミングでは、アスペクトの定義を見なければクラスの振る舞いを正確に理解することはできない。例えば、アスペクト A が選択するジョインポントが foo 中に含まれるとき、foo メソッドの挙動を把握するにはアスペクト A の定義を見る必要がある。foo メソッドにはアスペクトが織り込まれることを示す記述はないからである。これは obliviousness [3] と呼ばれるアスペクト指向の性質である。

obliviousness に対する解決策は、必要に応じてクラスとアスペクトの関係を視覚化して、ひと目でこれらの関係を理解できるようにすることである。コード中の obliviousness の性質を保つことは重要である。obliviousness はアスペクト指向の重要な性質のひとつだからである。したがって、oblivious を解消したプログラムの構造を可視化する必要がある。これは、ツールによって自動的に行われるべきである。

obliviousness の問題を改善するツールとして AJDT（AspectJ Development Tools）[14] がある。AJDT は AspectJ [9] 用の Eclipse プラグインである。AJDT では、ジョインポイントが発生する位置にマークを表示することによりプログラム中のどの箇所でアドバイスが実行されるかを示す。しかし、このようにしてアスペクトの織り込みを可視化する方法では obliviousness を完全に解決できない。AJDT を利用する開発者はプログラムの振る舞いを知るためにメソッドやアスペクトの実装（ソースコード）を確認する必要がある。これはメソッドのカプセル化を壊していることになる。

AJDT の持つ問題を解決するために、カプセル化を維持したままアスペクトの影響をクラス側のビューに可視化するためのツール AspectScope を提案する。AspectScope は、アスペクトが織り込まれたときに、メソッドの仕様がどう変わったかを解析し表示する。AJDT とは異なり、AspectScope はアスペクトの影響を間接的に受けるクラスにもその影響を表示する。アスペクトの影響を受けたメソッドの仕様の詳細を知るために、わざわざ実装を確認する必要はない。これにより、カプセル化を保持したままアスペクトの影響をクラスに反映させることができる。AspectScope は、アペクト指向言語 AspectJ を用いて開発された既存プログラムの修正に主に使用することを想定しており、outline ビューと javadoc 閲覧機能を提供する。また、 AspectScope が提供する機能 comment aspect を通して開発者はアスペクトの影響の表示方法の細かな制御を行うことがで

きる。

以下、2 章では obliviousness が引き起こす問題点について説明する。3 章ではその問題点を解決するために我々が提案する AspectScope について述べる。4 章では AspectScope の実装方法について簡単に説明し、5 章で既存のアスペクト指向プログラムに AspectScope を適用するときの評価を行う。6 章では、関連研究について取り上げ、7 章で本論文をまとめる。

2 問題点

アスペクト指向言語ではメソッドがアスペクトの影響を受けている可能性があるため元のプログラムの仕様がそのまま正しいとは限らない。そのため、本来ならば、モジュールプログラミングができるように、アスペクトが織り込まれたときにメソッドの仕様がどう変わったか分からなくてはならない。ここで、モジュールプログラミングが可能とは、プログラムを書くときに、コードを見ずに呼び出すメソッドの仕様だけを見て呼ぶ側を実装できることである。また、メソッドの仕様とは、メソッドのシグネチャとメソッドの振る舞いのことである。

例えば、AspectJ で記述された図 1 のような簡単な図形エディタプログラムを考える。図形を表す Figure インターフェースがある。Figure インターフェースの moveBy メソッドはエディタ上に描いた図形を移動させるときに呼び出される。点を表す Line クラスや直線を表す Point クラスは、 Figure インターフェースを実装している。Contract アスペクトは、 Point クラスの setX、setY メソッドの引数の値に事前条件を加える。execution ポイントカットの指定により before アドバイスは setX、setY メソッドの内部実装の直前に制約条件を織り込む。x と y の値それぞれがエディタのウィンドウサイズを越えると before アドバイスが例外を投げる。

モジュールプログラミングを可能にするには、Contract アスペクトの織り込み前後でメソッドの仕様が変わることが開発者に見えるようにしなければならない。しかし、これを支援するようなツールはない。そのため、Contract アスペクトを定義した後で、setX メソッドの javadoc コメントに開発者が以下のように拡張の詳細を追加することが考えられる。

Sets the horizontal position to a given argument. Advised by the <code>Contract</code> aspect. The set value should be no fewer than 0, nor more than 100.

これにより、Contract アスペクトによって setX メソッドに事前条件が加えられたことが開発者に分かる。しかし、この方法ではコメント内で関心事が完全に分離できていない。これでは、アスペクト指向言語の利点が損なわれてしまう。例えば、x に対する制約条件を変更したり Contract アスペクトを削除したときには、コメントをいちいち訂正する必要がある。

setX のコメントに事前条件を追加しない場合、リファクタリング時に、エディタ上に描かれた直線の移動範囲を制限するため、開発者が Line クラスの moveBy メソッドの実装を変更するとする。moveBy メソッドの中では、setX メソッドが呼ばれているので、開発者は setX メソッドの仕様も確認するはずである。moveBy メソッドを変更する前に、まずは moveBy メソッド内で呼んでいるメソッドの仕様を確認すると考えられるからである。しかし、setX メソッドの仕様は元のままなので、Contract アスペクトによって x の値が 0 以上 100 以下に制限されていることに開発者は気づけない。そのため、moveBy メソッドの引数 dx、dy に以下のような条件を開発者が新たに追加する可能性が考えられる。

```
public void moveBy(int dx, int dy) {
   if (dx < -50 || 50 < dx
       || dy < -25 || 25 < dy)
      throw new IllegalArgumentException();
   p1.setX(p1.getX() + dx);
   p1.setY(p1.getY() + dy);
   p2.setX(p2.getX() + dx);
   p2.setY(p2.getY() + dy);
}
```

dx と dy の値に対する新たな制約が原因となり、図形エディタは開発者の意図したとおりに動作しない。p1 の x の値が 0 で、dx の値が -30 であるような場合に、 moveBy メソッドからなぜ例外が発生するか開発者は分からない。実際には

```
p1.getX() + dx
```

の値が -30 になるため、Contract アスペクトの before アドバイス内で例外が発生する。

```
public interface Figure {
  /**
    * Moves the figure by dx along the x axis
    * and dy along the y axis
    */
  void moveBy(int dx, int dy);
}

class Line implements Figure {
   private Point p1, p2;

   /** Sets the starting point of this line */
   public void setP1(Point np1) {p1 = np1;}

   /** Sets the end point of this line */
   public void setP2(Point np2) {p2 = np2;}

   /** Returns the starting point of this line */
   public Point getP1() {return p1;}

   /** Returns the end point of this line */
   public Point getP2() {return p2;}

   /**
    * Moves this line by dx along the x axis
    * and dy along the y axis
    */
   public void moveBy(int dx, int dy) {
       p1.setX(p1.getX() + dx);
       p1.setY(p1.getY() + dy);
       p2.setX(p2.getX() + dx);
       p2.setY(p2.getY() + dy);
   }
}
```

```
class Point implements Figure {
  private int x, y;

 /** Sets the horizontal position to a given argument */
  public void setX(int nx) {x = nx;}

   /** Sets the vertical position to a given argument */
  public void setY(int ny) {y = ny;}

   /** Returns the horizontal position */
  public int getX() {return x;}

   /** Returns the vertical position */
  public int getY() {return y;}

   /**
    * Moves this point by dx along the x axis
    * and dy along the y axis
    */
  public void moveBy(int dx, int dy) {
      x += dx; y += dy;
  }
}

aspect Contract {
   /**
    * the set value <code>x</code> should be
    * no fewer than 0, nor more than 100
    * @throws IllegalArgumentException
    */
    before(int x) : execution(void Point.setX(int)) && args(x) {
      if (x < 0 || 100 < x)
         throw new IllegalArgumentException();
    }
    /**
     * the set value <code>y</code> should be
     * no fewer than 0, nor more than 50
     * @throws IllegalArgumentException
     */
     before(int y) : execution(void Point.setY(int)) && args(y) {
       if (y < 0 || 50 < y)
          throw new IllegalArgumentException();
    } }
```

図 1: AspectJ で実装された図形エディタ

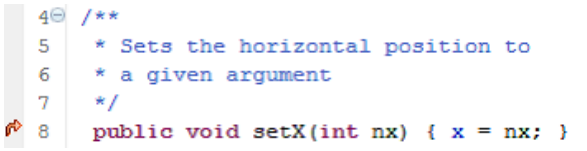

図 2: AJDT エディタにおける setX メソッドの表示

2.1 AJDT

AspectJ の開発支援ツールに AJDT (AspectJ Development Tools) がある。AJDT は Eclipse IDE 上で動くプラグインである。AJDT では、ジョインポイント（ジョインポイントシャドウ [11]）が発生する位置にマークを表示することによりプログラム中のどの箇所でアドバイスが実行されるかを示す。しかし、このようにしてアスペクトの織り込みを可視化する方法ではモジュールプログラミングができない。 AJDT を利用する開発者はプログラムの振る舞いを知るためにメソッドやアドバイスの実装の内部を確認する必要がある。これはメソッドのカプセル化を壊していることになる。以下では、AJDT が行う表示方法の詳細について述べる。

2.1.1 execution ポイントカット

execution ポイントカットは指定のメソッドが呼び出されたときをジョインポイントとして選択する。そのため、図 2 のように呼ばれる側の Point クラスに AJDT はマークを付ける。setX メソッドが呼び出されたときにアドバイスが実行されることを矢印型のマークが示している。

しかし、このように視覚化する方法では、アスペクトの影響を受けた setX メソッドの仕様がどう変化したのか一目で分からない。例えば、setX メソッドがどのように拡張されるかの詳細を調べるためには Contract アスペクトの実装を確認しなければならない。また、図 3 のように、Line クラスの moveBy メソッドには setX メソッドの呼び出し箇所にアスペクトによる拡張を示す矢印型のマークはない。ポップアップ（図 3、灰色の枠）による表示もアスペクトの織り込み前後で変わらない。そのため setX メソッドの定義を見なければ Constract アスペクトによって拡張されることに開発者は気づけない。

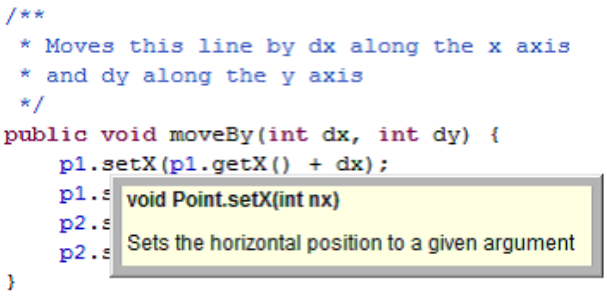

図 3: execution ポイントカットを使用したときの moveBy メソッドの表示

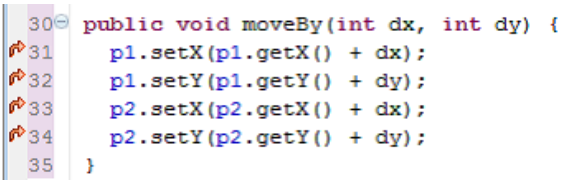

図 4: call ポイントカットを使用したときの moveBy メソッドの表示

2.1.2 call ポイントカット

call ポイントカットは指定のメソッドが呼び出されるときをジョインポイントとして選択する。Contract アスペクトの before アドバイスでは execution ポイントカットを使用している。以下のように、execution ポイントカットの代わりに call ポイントカットを使用してもよい。setX、setY メソッドの降る舞いは execution ポイントカットを使用したときと変わらない。

```
aspect Contract {
  /** ... */
  before(int x) :
         call(void Point.setX(int))
         && args(x) {
    if (x < 0 || 100 < x)
       throw new IllegalArgumentException();
   }
  /** ... */
  before(int y) :
         call(void Point.setY(int))
         && args(y) {
    if (y < 0 || 50 < y)
       throw new IlleaglArgumentException();
   }
}
```

このとき、呼ぶ側である Line クラスの moveBy メソッドに AJDT はマークを付ける（図 4）。しかし、この場合にも setX メソッドがどのように拡張

されるかの詳細は Contract アスペクトの実装を見なければ知ることはできない。AJDT は、ジョインポイントの発生箇所を示すだけである。また、図 4 の moveBy メソッドを呼ぶ他のクラスからは setX メソッドが拡張されていることを示すマークすらない。例えば、以下のように Line クラスの moveBy メソッドを呼び出す MultiLines クラスがあるとする。一筆書きで描いた複数の直線からなる任意の図形をこのクラスは表す。

```
public class MultiLines implements Figure {
   private List lines;
        :
   public void moveBy(int dx, int dy) {
      for (Iterator it =
           lines.iterator(); it.hasNext();)
        ((Line) it.next()).
                    moveBy(dx, dy);
    }
}
```

このとき、MultiLines クラス内には Line クラスの moveBy メソッドの仕様が拡張されていることを示す表示は何もない。Point クラスの setX、setY メソッドが拡張されていることを、MultiLines クラスだけを見て知ることはできない。Line クラスの moveBy メソッドの実装を確認する必要がある。

2.2 モジュールプログラミング

アスペクトを織り込むと、ポイントカットで選択したメソッドだけでなく、それを呼ぶ側や呼ばれる側のメソッドの仕様が次々に変わっていく。例えば、Contract アスペクトによって事前条件が setX メソッドに加えられた。そのため、setX メソッドを呼び出す Line クラスの moveBy メソッドの仕様もこの事前条件を考慮したものに変わる。さらに、Line クラスの moveBy メソッドを呼び出す MultiLines クラスの moveBy メソッドの仕様も変わる。そのため、アスペクトが織り込まれる度に、どのメソッドの仕様が拡張されるかをその都度解析し可視化するツールが必要である。これを見ながらなら、現在のメソッドの仕様が見て分かるためモジュールプログラミングが可能になる。

一般に、オブジェクト指向言語の場合には、メソッドの仕様だけを見てプログラムを書ける。API（Application Program Interface） を使ったプログラミングはその典型的な例である。アスペクト指向言語とは異なりプログラムのコンパイルや実行の前後で、メソッドの仕様が変化することはない。そのため、オブジェクト指向言語ではモジュールプログラミングが可能である。

3 AspectScope

AJDT の持つ問題を解決するために、我々はカプセル化を保持したままアスペクトの影響をクラス側のビューに可視化するためのツール AspectScope を提案する。AspectScope は、アスペクトが織り込まれたときに、メソッドの仕様と振る舞いがどう変わったかを解析し可視化する。アスペクトの影響を間接的に受けるメソッドに対しても解析を行う。そのため、アスペクトの影響を受けたメソッドの詳細を知るために、わざわざ実装を確認する必要はない。これにより、メソッドのカプセル化が保たれ、モジュールプログラミングを助ける。AspectScope は、アスペクト指向言語 AspectJ を用いて開発された既存プログラムの修正に主に使用することを想定している。

3.1 AspectScope エディタ

AspectScope は、アスペクトによる拡張をメソッドの仕様に表示できる。例えば、Point クラスの setX メソッドの仕様は図 5 のようになる。Contract アスペクトの before アドバイスによって仕様が拡張されていることを、ルーラー上の矢印型のマークとポップアップ内の before アドバイスに関する javadoc コメントによって表現している。

アスペクトが間接的に影響を及ぼすメソッドの仕様にも AspectScope はその影響を表示できる。つまり、ポイントカットによって指定されたメソッドとそのメソッドのコールグラフツリー上にある他のメソッドに AspectScope は影響を反映できる。ひとつのアドバイスが、メソッドのコールグラフツリー

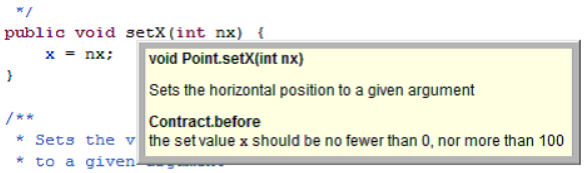

図 5: AspectScope による表示

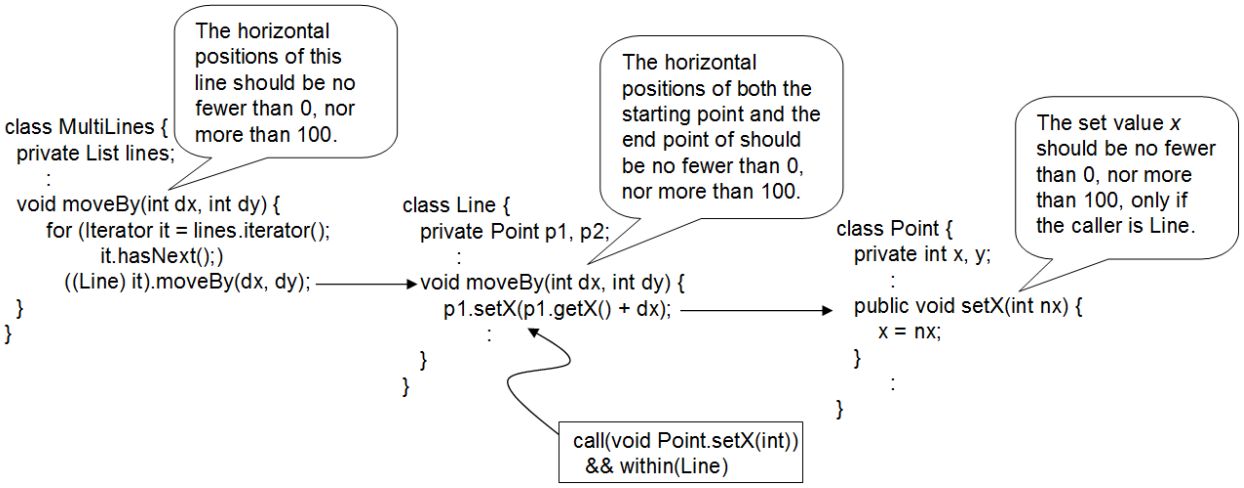

図 6: setX メソッドのコールグラフ

のいたるところに影響を与える可能性があるためである。例えば、call ポイントカットの場合には、呼ぶ側だけでなく呼ばれる側のメソッドの仕様も拡張される [8]。例えば、以下のようにアドバイスが call ポイントカットを指定しているとする。

call(void Point.setX(int)) && wihtin(Line)

これにより、setX メソッドを呼ぶ Line クラスの moveBy メソッドだけでなく、呼ばれる側の Point クラスの setX の仕様も拡張される。また、moveBy メソッドを呼ぶ MultiLines クラスの moveBy メソッドの仕様も拡張される（図 6）。また、ひとつのアドバイスによる拡張の仕様は各メソッドの抽象化に沿っていなければならない（図 6、吹き出し）。例えば、図 5 の javadoc コメント中にあるアスペクトの仕様部分

The set value <code>x</code> should be no fewer than 0, nor more than 100, only if the caller is Line

は setX メソッドの抽象化に沿ったものである。しかし、呼び出し側の Line クラスの moveBy メソッドの仕様としてはそぐわない。コメント中の x が何を表す変数かは moveBy メソッドからは分からない。また、Line から呼ばれたときのみ制約条件がかかるという内容 *"only if ..."* も、呼ぶ側の Line クラスに表示する仕様としては不適切である。Line クラスは、Point 型の 2 つの変数を使って直線の始点・終点を表現しているので、moveBy メソッドに載せべきアスペクトの仕様は以下のような内容が適切である。

The horizontal position of both the starting point and the end point should be no fewer than 0, nor more than 100

x の代わりに始点と終点の水平方向の座標という表現を使用しているため、カプセル化の原則を維持している。MultiLines クラスの moveBy メソッドの仕様については、*"The horizontal positions of this line"* の部分のコメントが Line のものとは異なる。メソッドの抽象化に適したこれらのアスペクトの仕様は、アスペクトを定義するときに開発者が記述すべきである。アドバイスの実装を定義することと、そのアドバイスのクラスへの振る舞いを考慮することは開発者が一貫性を持って行う作業だからである。

3.2 comment aspect

AspectScope は、メソッドの抽象化に沿った仕様を提供するための機能 comment aspect を提供する。comment aspect を通して、開発者はひとつのアドバイスに複数の javadoc コメントを付加し、指定したメソッドにそれらを反映させることができる。矢印型のマークも comment aspect の制御に従って表示・非表示が決まる。

ひとつのアドバイスに関する仕様をメソッドコールグラフツリー上のメソッドに無制限に反映させて

```
aspect Contract {
    /** @comment
     *    The set value <code>x</code> should be no fewer than 0, nor more than 100, only if the caller
     *    Line class
     *    @throws IllegalArgumentException
     *
     * @comment (execution(void Line.moveBy(int, int)))
     *    The horizontal positions of both the starting point and the end point should be no fewer than 0, nor
     *    more than 100
     *    @throws IllegalArgumentException
     *
     * @comment (execution(void MultiLines.moveBy(int, int)))
     *    The horizontal positions of the lines should be no fewer than 0, no more than 100
     *    @throws IllegalArgumentException
     */
     before(int x) : call(void Point.setX(int)) && args(x) && within(Line) {
        if (x < 0 || 100 < x)
            throw new IllegalArgumentException();
     }
         :
}
```

図 7: @comment 注釈を用いた javadoc コメントの記述とその制御

いいわけではない。アドバイスの仕様をコールグラフツリーのどこまで反映させるかはプログラムに依存する。そのため、アドバイスの仕様の適用範囲は開発者が決定する必要がある。例えば、Contract アスペクトの before アドバイスには図 7 のように記述する。アドバイスのコメント内に @comment 注釈が 3 つある。ひとつ目の注釈はポイントカットが指定するメソッドに反映されるコメントである。この場合には、setX メソッドに *"The set value ..."* のコメントを反映させる。@comment の引数には制御文からなる論理式を書くことができる。 2 つ目と 3 つ目の注釈には制御文がある。

execution(*method pattern*)

は、*method pattern* に適合するメソッドに @comment 以下に書かれた javadoc コメントを関連付ける。この場合には、Line クラスの moveBy メソッドや MultiLines クラスの moveBy メソッドにそれぞれのコメントを関連付ける。

制御文には execution の他に within や caller がある。また、メソッドパターンやクラスパターンにはワイルドカード *、+、.. を使用できる。

within(*class pattern* || *method pattern*)

caller(*int*)

within は、引数で指定しているクラスやメソッドの内部で呼んでいるメソッドにコメントを関連付ける。例えば、within(* csg.ppl.*(..)) は csg.ppl パッケージ内の全てのメソッドのうち、ポイントカットで指定されたメソッドのコールグラフ上にあるものにコメントを関連付ける。caller は int 型の整数を引数に取る。これは、ポイントカットで指定されたメソッドの何階層上にある呼び出し側にコメントを表示させるかを指定できる。caller は、指定された階層にだけコメントを表示し、その階層以上やそれ以下といった指定はできない。caller は、ポイントカットで指定されたメソッドを呼ぶ複数のメソッドに対し、同一のコメントを反映させたいときに使用する。

3.3 comment アドバイス

メソッドにコメントがない場合には、AspectScope 独自のアドバイス comment を用いてコメントをメソッドに織り込むことができる。API のように、外部に公開するメソッドに対しては javadoc コメントが付けられているのが一般的だが、private なメソッドにまで javadoc コメントが付けられているとは限らない。comment アドバイスは、例えば以下

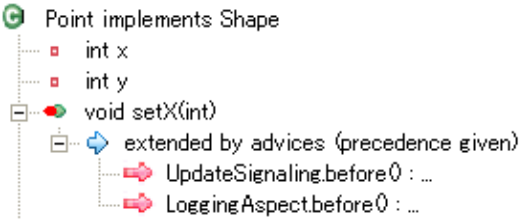

図 8: outline ビューア

のように記述する。

```
/**
 * @comment
 *  Returns the distance between this line
 *  and the point.
 *
 */
comment() :
  execution(int Line.getDistance(Point)) {}
```

comment アドバイスが指定できるポイントカットは execution だけである。織り込んだコメントは AspectScope エディタのポップアップ機能を通して見ることができる。comment アドバイスは unweave の機構を備える。

3.4 アウトラインビューア

仕様の変化が一目で分かるように、AspectScope はアウトラインビューアを提供している。これにより、実装を見なくても、どのメソッドの仕様が変化したかを知ることができる[1]。AspectScope は、アスペクトが行う 2 つの操作に対してそれぞれ可視化を行う。以下では、その詳細について述べる。

3.4.1 ポイントカット＆アドバイス

ポイントカットによって選択されたメソッドに対し、AspectScope はその仕様の変更を表示する。例えば、Point クラスの setX メソッドの仕様が拡張されたことを図 8 のように表示する。ここでは、2 つのアドバイスが setX メソッドを拡張している。また、declare precedence 句によってアスペクトの優先順位が定められているときは優先度の高い順に上から列挙する。

[1]現時点では、アスペクトが間接的に影響を与えるメソッドに対しては表示が不十分な点がある。今後、機能の充実化を行う予定である。

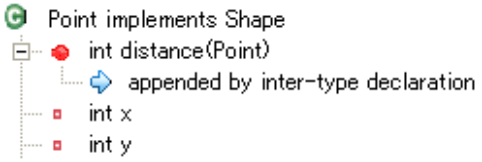

図 9: インタータイプメソッドの表示

AJDT では、アウトラインの setX メソッドに拡張を示すマークが付く。ただし、setX メソッドが execution ポイントカットで指定されていた場合だけである。 call ポイントカットによって指定されていた場合には、アウトラインに何の表示も現れない。また、どのアスペクトがどのように拡張しているかを知るためには setX メソッドやアスペクトの実装を確認する必要がある。

3.4.2 インタータイプ宣言

インタータイプ宣言によるクラスへのメンバの追加が行われると、AspectScope は追加されたメンバをクラスのアウトラインに表示を行う。図 9 は、Point クラスに distance メソッドを付加していることを示している。

AJDT では、インタータイプ宣言で追加されるメンバの表示は行わない。Point クラスに拡張を示すマークが付加されるだけである。そのため、どのようなメンバが追加されているかを知るにはアスペクトの実装を確認する必要がある。

4 その他の例

これまで、いくつかの例を通して AspectScope の機能を見てきた。この章では、その他の例をいくつか取り上げる。

4.1 図形エディタ

図 6 では、MultiLines クラスまでの呼び出し階層について述べた。MultiLines クラスやその他の図形クラスは、さらに、Figure インターフェースの moveBy メソッドを通して以下のように呼び出される。

```
class DrawApplication implements
          MouseListener, MouseMotionListener {
   privae int x0, y0;
```

```
   private Figure f;
      :
   public void mouseDragged(MouseEvent e) {
       if (f == null)
            return;
       f.moveBy(e.getX() - x0, e.getY() - y0);
            :
   }
      :
}
```

DrawApplication クラスの mouseDragged メソッドの中で、Figure インターフェースに対して moveBy メソッドを呼び出している。

そのため、Contract アスペクトにはそれぞれのメソッドの抽象化に沿ったコメントを用意する必要がある。Figure インターフェースの moveBy メソッドに対しては、水平方向の移動範囲が 0 以上 100 以下に制約されるということを書く。また、mouseDragged メソッドには、マウスドラッグの終点がエディタのウィンドウサイズ内になければならないことを書く。また、例外処理は mouseDragged メソッド内で処理するため、例外を投げることをコメントに明記する必要はない。

```
aspect Contract {
 /**
  * @comment (execution(
  *          void Figure.moveBy(int, int)))
  *   The horizontal positions of the edges
  *   of this figure should be no fewer 0,
  *   nor more than 100
  *   @throws IllegalArgumentException
  *
  * @comment (execution(
  *       void DrawApplication.mouseDragged(
  *                              MouseEvent)))
  *   The released position of the mouse
  *   dragging should be inside the window.
  */
 before() :
       call(void Figure.moveBy(int, int)) {}
}
```

図 7 のアドバイスとは別に、ここでは、Figure インターフェースの moveBy メソッドの呼び出し時を指定したアドバイスを開発者は定義する必要がある。MultiLines クラスまでの呼び出し階層は、mouseDragged メソッドまでの呼び出し階層とは異なるからである。

4.2 デバッグツールとしての役割

メソッドに反映されるアドバイスのコメントにより、拡張に誤りがあることに気づくことができる。例えば、以下のようにエディタ上の図形の再描画を通知する DisplayUpdate アスペクトがあるとする。これにより、図形を移動させたときには必ず図形の再描画が行われるようになる。また、図形クラスを集めた figures パッケージ内で、このアドバイスの仕様が反映されるようにするとする。制御には wihtin を用いる。

```
aspect DisplayUpdate {
  /**
   * @comment (within(* figures.*(..))
   *   Signals the <code>Display</code>
   *   to update a shape changes.
   */
  after() :
       call(void Figure+.moveBy(int,int)); {
     Display.update();
  }
```

リファクタリング時に、矢印を表す Arrow クラスを他の開発者が以下のように新たに定義するとする。Arrow クラスは矢印の先端の三角形を表す Triangle 型のフィールド tri を持つ。moveBy メソッド内では tri に対する moveBy メソッドの呼び出しを行っている。

```
public class Arrow implements Figure {
   private Point p1, p2;
   private Triangle tri;
       :
   public void moveBy(int dx, int dy) {
      tri.moveBy(dx, dy);
      p1.setX(p1.getX() + dx);
      p1.setY(p1.getY() + dy);
      p2.setX(p1.getX() + dx);
      p2.setY(p2.getY() + dy);
  }
}
```

このプログラムを追加するとバグが発生する。Arrow クラスが表す矢印をエディタ上で移動させると画面がちらついてしまう。Arrow クラスの moveBy メソッドの中で Triangle クラスの moveBy メソッドを呼んでいるため、update メソッドが 2 回呼ばれることが原因である。AspectScope を使用すると、Arrow クラスの moveBy メソッドの仕様を見ただけでバグの原因が分かる（図 10）。ポップアップには、DisplayUpdate アスペクトの after アドバイスのコ

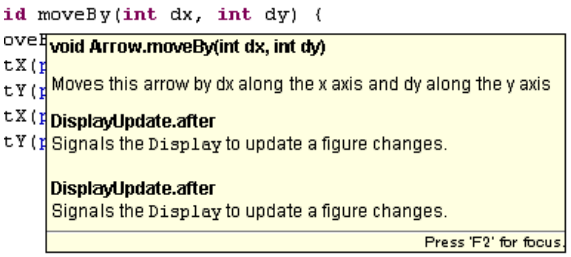

図 10: DisplayUpdate アスペクトによる拡張

メントが 2 つ表示されており、upadte メソッドが 2 度呼ばれることを示している。

AJDT を使用する開発者は、このバグにはなかなか気づけない。Triangle クラスの moveBy メソッドには表示が出るが、Arrow クラスの moveBy メソッドには何も表示もない。そのため、この moveBy メソッドが拡張されていることには気づけない。拡張されていることを確認するには、Arrow クラスの moveBy メソッドを呼び出しているクラスの実装を見なければならない。

5 予備的な評価

アスペクトが間接的に影響を与えるメソッドに対し、仕様の拡張を実際に行わなければならないかの調査を行った。そのために、アスペクト指向言語を使用したアプリケーションサーバープログラムの Health Watcher [5] を利用した。 Health Watcher には CaesarJ [12] と AspectJ による実装があるが、我々が評価に用いたのは AspectJ のプログラムである。クラスの数は 692（LOC 9591）であり、アスペクトの数は 25（LOC 1989）である。これらのアスペクトの使用用途は大別して 6 つに分類できた。デザインパターン、永続化、分散化、トランザクション、例外ハンドリング、ロギングである。以下では、特筆すべき点に焦点を当てて説明する。

デザインパターン

Health Watcher プログラムでは、アスペクト指向言語を用いて 4 種類のデザインパターン [4] が実装されている [7]。Observer パターン、Command パターン、Factory パターン、State パターンである。

Observer パターンでは、オブザーバーの登録（addObserver）と更新通知（updateObserver）をアドバイスで実装している。例えば、以下のアドバイスは 17 個のメソッドの振る舞いを拡張する。

```
after(Subject subject):
                    call(* Subject+.set*(..))
                    && this(CommandServlet+)
                    && target(subject) {
 Iterator iter =
           getObservers(subject).iterator();
 while ( iter.hasNext() ) {
   updateObserver(subject,
                  ((Observer)iter.next()));
 }
}
```

そのうちのひとつが UpdateEmployeeData クラスの setPassword メソッドである。このメソッドの呼び出し時に更新通知のためのメソッド updateObserver が実行される。また、UpdateEmployeeData クラスの executeCommand メソッド内で setPassword メソッドが以下のように呼ばれている。変数 employee は executeCommand メソッド内でのみ使用されている。また、executeCommand メソッドを呼ぶメソッドは、Command インターフェースの executeCommand メソッドを通して実行時に呼ばれる。そのため、コールグラフの構造は、executeCommand メソッドが setPassword メソッドを呼ぶ 1 階層だけである。

```
public class UpdateEmployeeData
               extends CommandServlet {
  public void executeCommand(CommandReceiver
                                    receiver) {
    Employee employee = null;
    String newPassword = ...;
        :
    //選択されるジョインポイント
    employee.setPassword(newPassword);
        :
  }
}
```

このとき、呼ばれる側の UpdateEmployeeData クラスの setPassword メソッドの javadoc コメントには以下を追加する必要がある。

Calls <code>updateObserver</code> after setting the new password of the employee to update the Observer.

また、executeCommand メソッドには以下を追加する必要がある。newPassword は executeCommand メソッドのローカル変数であるため、パスワードが更新されることに言及するのは executeCommand メ

パターン	拡張したメソッド数	呼び出し1階層上	呼び出し2階層上	呼び出し3階層上	呼び出し6階層上	呼び出し9階層上	呼ばれる側
Observer	17	17	0	0	0	0	17
Command	3	3	0	0	0	0	0
Factory	0	NA	NA	NA	NA	NA	NA
State	15	14	13	10	5	2	0

表 1: デザインパターン

ソッドの抽象化にそぐわない。

Updates the information of the employees by calling <code>updateObserver</code>.

他の 3 種類のデザインパターンについては、表 1 のようになった。Command パターンは、Observer パターンと同様にインターフェースによるメソッド呼び出しによってコールグラフの構造が 1 階層になった。State パターンは、最大でコールグラフを 9 階層分だけ上にまで仕様を反映させる必要があった。表 1 には 4、5 階層についてのデータが省略してあるが、ひとつ下の階層のデータと同じであることを意味している。7、8 階層についても同様である。また、Factory パターンには、アドバイスが存在しなかった。

ロギング

ロギングコードはプログラムの振る舞いに影響を与えるわけではないため、呼び出し側のメソッドにログに関するコメントを反映させる必然性はないと考えられる。ロギングのためのアスペクト HWLogging は、HealthWatcherFacade のコンストラクタの呼び出し時にログ出力を行う。このコンストラクタを呼び出しているメソッドは以下の HealthWatcherFacade クラスの getInstance メソッドである。

```
public synchronized static
           HealthWatcherFacade getInstance() {
  if (singleton == null)
     singleton = new HealthWatcherFacade();
  return singleton;
}
```

ここで、getInstance メソッドの仕様にロギングに関する情報を追加する場合には、以下のような記述を行うのが一般的だと考えられる。

Configures the logging before returning the instance

ロギングアスペクトについてまとめた表を表 2 に示す。表の括弧付きの数字はコールグラフ上のメソッドにアドバイスのコメントを必ずしも反映させる必要がないことを表す。

例外ハンドリング

アスペクト内で例外処理を行っているものが 4 つある。ExceptionHandlingPrecedence アスペクトはこれら 4 つのアスペクトの優先順位を定めたアスペクトであり、例外処理は扱っていない。HWPersistenceExcpetionHandler アスペクトを除いて 3 つのアスペクトではアドバイスの仕様を呼び出し側のメソッドに追加する必要がなかった。これは、開発者が定義するメソッド呼び出しがこれ以上存在しなかったためである。例えば、HWDistributionExceptionHandler アスペクトの around アドバイスは以下のポイントカットを指定している。

```
execution(* HWServlet+.do*(
      HTTPServletRequest, HTTPServletResponse))
      && args(.., response)
```

このポイントカットを使って HttpServlet クラスのサブクラスの doGet メソッドなどを拡張する。doGet メソッドは HTTP の GET リクエストに対して呼び出されるメソッドであるため、開発者が仕様拡張を行えるのは doGet メソッドまでである。

表 1 から 表 3 を見て分かるように、拡張したメソッドの呼び出し側のメソッドに、アドバイスによ

アスペクト	拡張したメソッド数	呼び出し1階層上	呼び出し2階層上	呼ばれる側
HWLogging	2	(2)	(1)	1

表 2: ロギング

アスペクト	拡張したメソッド数	呼び出し1階層上	呼び出し4階層上	呼び出し6階層上	呼び出し7階層上	呼び出し9階層上	呼ばれる側
HWDistributionExceptionHandler	4	0	0	0	0	0	0
HWPeRsistenceExceptionHandler	9	5	5	3	2	1	2
HWTransactionExceptionHandler	4	0	0	0	0	0	0
HWUpdateObserverExceptionHandler	5	0	0	0	0	0	0
ExceptioHandlingPrecedence	0	NA	NA	NA	NA	NA	NA

表 3: 例外ハンドリング

る拡張の仕様を反映させれば充分であることが多い。コールグラフ上をさかのぼって何階層も仕様を作成することはあまりなく、多くの場合は 3、4 階層で済む。これは、インターフェースによるメソッド呼び出しを行っているプログラムが多いことなどが上げられる。そのため、ひとつのアドバイスに複数のコメントを記述したとしても、その数は現実的な個数に 収まることが期待できる。

6 関連研究

AspectScope は、Aspect-Aware Interface [10] と基本的な考えは同じである。これは、obliviousness を解決するアスペクト指向言語用の新しいインターフェースである。このインターフェースを通して、アスペクトが織り込まれる度にそのクラスへの影響を表示するべきであると提案している。これにより、モジュールプログラミングが可能になる。しかし、Aspect-Aware Interface は基本的な概念を提示しただけである。各ポイントカットの表示方法などに明確な指針はない。execution ポイントカットの場合には呼ばれる側のメソッドの仕様を変更するが、call や get、set ポイントカットなどは未解決の問題としている。そのため、どのポイントカットでも、影響を与えるすべてのメソッドに仕様の拡張を表示するべきという考えは AspectScope 独自のものである。また、各メソッドに合った仕様の詳細を、javadoc コメントを利用して表示するというのも AspectScope の新しい提案である。AspectScope は、Aspect-Aware Interface を実現した AspectJ 用のツールであるといえる。

OpenModule [1, 13] や XPI（crosscut programming interface）[6] は、obliviousness を解決する別の手法を提案している。これらの手法では、アスペクトが拡張可能なジョインポイントをモジュールのインターフェースに公開する。このインターフェースで公開されているジョインポイントしかアスペクトは拡張しないため、クラスの obliviouesness が保たれる。しかし、この手法では、アスペクトによって拡張されるジョインポイントを予め開発者が予測しなければならない。拡張される可能性のあるポイントカットを予測することは困難である。予測できない場合には、拡張するジョインポイントに合わせて毎回モジュールインターフェースを更新する必要がある。AspectScope はアスペクトがクラスをどう拡張しているかを表現するためのツールである。そのため、 OpenModule や XPI を補完する役割があると考えられる。

active model [2] は、横断的な構造を表現するための別の手法を提案している。active model では、クラス図を用いてアスペクトが選択するジョインポイントを可視化する。しかし、アスペクトが間接的に影響を及ぼす構造をこのクラス図から判断するのは難しい。

7 まとめ・今後の課題

7.1 まとめ

本稿において我々は AspectJ のためのプログラミングツールである AspectScope を提案した。AspectScope は、アスペクトが織り込まれたときに、

メソッドの仕様と振る舞いがどう変化したかを解析し可視化する。また、アスペクトの影響を間接的に受けるメソッドに対しても解析を行う。そのため、アスペクトの影響を受けたメソッドの詳細を知るために、実装をいちいち確認する必要はない。これにより、メソッドのカプセル化が維持され、モジュールプログラミングを助ける。

7.2 今後の課題

State パターンを実装したアスペクトや HWPersistenceExceptionHandler アスペクトでは、最大で 9 階層もメソッドコールグラフをさかのぼったために、多くのコメントを開発者が用意する必要があった。comment aspect は、現段階で、このような深い呼び出しの構造に対し充分に対応できているとはいえない。今後は、comment aspect の記述力強化や利便性向上のための機能を検討していく必要がある。

参考文献

[1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, LNCS 3586*, pages 144–168. Springer-Verlag, 2005.

[2] Wesley Coelho and Gail C. Murphy. Presenting crosscutting structure with active models. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 158–168, New York, NY, USA, 2006. ACM.

[3] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John M Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[5] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Claudio Sant Anna, Sergio Soares, Paulo Borba, Uira Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *ECOOP 2007 - Object-Oriented Programming, LNCS 4609*, pages 176–200. Springer-Verlag, 2007.

[6] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design With Crosscutting Interfaces. In *IEEE Software, vol.23*, pages 51–60, 2006.

[7] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM.

[8] Michihiro Horie and Shigeru Chiba. An Outline Viewer for AspectJ Programs. TOOLS EUROPE 2007, 2007.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, LNCS 2072*, pages 327–353. Springer-Verlag, 2001.

[10] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[11] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs, 2002.

[12] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[13] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2006. ACM.

[14] The Eclipse Foundation. *AspectJ Development Tools(AJDT)*. http://www.eclipse.org/ajdt.