# AspectScope によるアスペクトとクラスのつながりの視覚化

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**

本論文では、アスペクトとクラスの間の繋がりを視覚化するツール AspectScope を提案する。既存のツールの AJDT と異なり、AspectScope はクラスやメソッドのカプセル化を保持したままアスペクトの繋がりを表現する。このため、アスペクトはクラスを拡張するものと AspectScope ではとらえ、各クラスのアウトラインを表示する。また、AspectScope はメソッドやフィールドの javadoc コメントに加えて、それを拡張するアドバイスやポイントカットの javadoc コメントも表示する。

## 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 [2] 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 でないプログラムの視覚化方法と oblivious なソースコードの間で一貫性を確保する必要がある。これは、ツールによって自動的に行われるべきである。

この obliviousness の問題を解決するツールとして AJDT（AspectJ Development Tools）がある。AJDT では、クラス側のジョインポイントの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アスペクトとクラスの繋がりを理解するのが困難になる。開発者はプログラムの振る舞いを知るためにエディタなどを用いてメソッドの実装を見なければならない。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、我々はカプセル化を保持したままアスペクトとクラスの関係を視覚化する AspectScope を提案する。AspectScope ではクラスをモジュールとして拡張するものとアスペクトを捉える。そのため、AspectScope ではクラスのアウトラインをモジュールのインターフェースとして表示する。それにより、メソッドのカプセル化が保たれる。また、AspectScope は AspectJ 用の Eclipse プラグインとして実装されている。

以下、2 章では AJDT の提供するツールの持つ問題点について説明する。3 章ではその問題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、アスペクト指向プログラミングの具体的な例を用いて AJDT と AspectScope でどのような表示の違いが出るかを比較する。5 章では、関連研究について取り上げ、6 章で本論文をまとめる。

## 2　従来の視覚化の問題点

AJDT の提供するツールには AJDT エディタや Visualiser ビューなどのアスペクトとクラスのつながりを視覚化するためのツールがある。これらのツールを通してアスペクトとクラスの関係が視覚化される。例えば、図 1 のように AJDT エディタはポ

イントカットによって選択されるジョインポイントの位置をエディタのルーラー上にマークを付けることでアスペクトの織り込みを知らせる。Visualiser も同様にジョインポイントの位置にマークを付けることによってアスペクトの存在を知らせる。

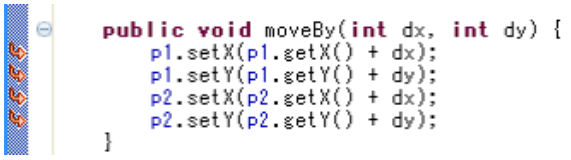


図 1: AJDT エディタにおけるマーキング

AJDT はアスペクト指向を event-driven なプログラミング技術と捉えている。event-driven とは、メソッド呼び出しやフィールドアクセスといったプログラムの実行をイベントの列と考えるということである。アドバイスはそのイベントに影響を与えるものである。AJDT のツールはジョインポイントの位置にマークを付けることでどのイベントでアドバイスが実行されるかを示す。

しかし、これはクラスやメソッドのカプセル化を壊している。AJDT のツールではエディタなどを用いてプログラムの内部実装を見なければアスペクトのつながりを理解できないからである。

# 3 AspectScope: An Aspect-Aware Outline Viewer

AJDT によるカプセル化の破壊に対する解決策として、我々は AspectScope を開発した。AspectScope はアスペクトが織り込まれたプログラムを視覚化するにあたり、プログラムの詳細を隠蔽し、カプセル化を保つことで AJDT のもつ問題点を克服した。これにより、より高い抽象度でアスペクトとクラスのつながりを理解することができるようになった。

## 3.1 Extension としてのアスペクト

AspectScope は、アスペクトが織り込まれるときも、クラスやメソッドをカプセル化されたモジュールとして考えられるようにするため、アスペクトをそれらのモジュール間のインタフェースを拡張するものととらえる。ここでいうインタフェースとは、そのクラスがどのようなメソッドやフィールドを外部に公開し、それぞれがどのように動作するかの仕様である。アスペクトは、そのような外部に公開されたインタフェースの仕様を拡張するもの、と考える。

こう考えることで、各クラスに対するアスペクトの織り込みの影響を、 外部に公開されたインタフェースの範囲内で説明することが可能になる。メソッドの内部実装に言及してカプセル化を破壊することもない。一方、織り込みの影響の詳細は抽象化されて見えなくなる。詳細が知りたい場合にはこれは欠点だが、視覚化された表現を簡潔にする意味では利点である。具体的には例えば execution ポイントカットと call ポイントカットの差異は AspectScope では無視される。

## 3.2 AspectScope

AspectScope はアスペクトが織り込まれた後に各々のモジュールがどうつながるかを表示するためのツールである。アスペクト指向言語のひとつである AspectJ 用の Eclipse プラグインとして実装した。AspectScope は 2 つのパネルから構成されるビューアである （図 2）。左側のパネルはアウトラインビューである。アスペクトによって織り込まれたクラスのメンバをインターフェースとして列挙する。右側のパネルは javadoc ビューである。各メンバがアドバイスによってどのように拡張されるかを示す。AJDT と異なり、アスペクトにより拡張されたメソッドの挙動を把握するためにメソッドの内部実装を確認する必要はない。

AspectScope は extension としてアスペクトをとらえたツールである。以下ではアスペクトによるクラスの拡張を、本ツールが具体的にどう表現するか述べる。

**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ばれたとき」をジョインポイントとして選択する。例えば

```
execution(void Point.setX(int))
```

は Point クラスの setX メソッドが呼ばれたときをジョインポイントとして選択する。このジョインポ

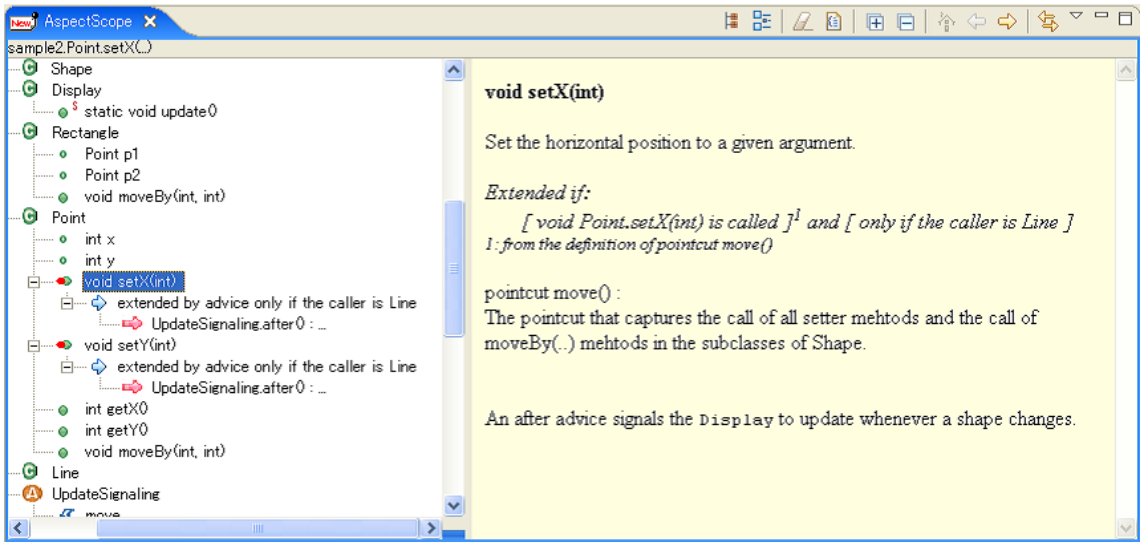


図 2: AspectScope

イントに結びつけられたアドバイスは、setX メソッドが実行されるときに一緒に実行される。したがって継承によるメソッドの上書きと同様、そのアドバイスは setX メソッドの動作を拡張することになる。

そのようなアドバイスは、AspectScope では対象となるメソッドを拡張するものとしてアウトラインビューに表示される。setX の場合の例を図 3 に示す。setX メソッドの動作は、UpdateSignaling アスペクトの after アドバイスによって拡張されることがわかる。

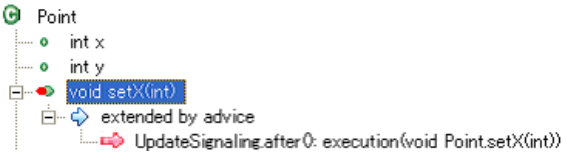


図 3: setX メソッドが拡張されることを示す表示

**call ポイントカット**　call ポイントカットは「指定のメソッドを呼び出したとき」をジョインポイントとして選択する。例えば

call(void Point.setX(int))

は Point クラスの setX メソッドを呼んだときをジョインポイントとして選択する。このジョインポイントに結びつけられたアドバイスは、setX メソッドの呼び出し側で、setX メソッドの呼び出しの前後に実行される。execution が呼ばれた側であるのに対し、call は呼び出し側をジョインポイントとして選択する。

call ポイントカットの視覚化は AJDT と AspectScope で異なる。例えば、Line クラスの moveBy メソッドの中で Point クラスの setX メソッドを呼び出すとする。このとき、AJDT では図 4 の表示になる。event-driven な解釈に基き setX メソッドの呼び出し箇所の前後でアドバイスが実行されると考えるため、Line クラスの moveBy メソッド中の下線の位置にマークが付く。

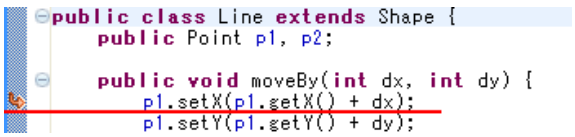


図 4: AJDT による表示

一方、AspectScope では図 5 の表示になる。アスペクトを extension ととらえる考えに基づき、呼ばれた側のメソッドが拡張されたと考えるため、Point クラスの setX メソッドが拡張されたという表示になる。これは、execution ポイントカットにおける表示方法と同一である。setX メソッドの呼び出し側でアドバイスが実行されたと考えても、呼ばれたときに拡張されたと考えても moveBy メソッドの動作は変わらない。しかし、呼び出し側を考慮する

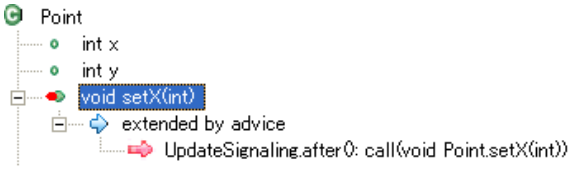


図 5: AspectScope による表示

とカプセル化を壊してしまうため、AspectScope では呼ばれた側の setX メソッドの動作が拡張されたと考える。

get ポイントカットや set ポイントカットの場合も call ポイントカットと同様に呼ばれた側が拡張されたという表示になる。get ポイントカットはフィールドの値を参照するときをジョインポイントとして選択する。set ポイントカットはフィールドへ値を代入するときをジョインポイントとして選択する。

**within、cflow ポイントカット**　call ポイントカット等は within や cflow ポイントカットと一緒に使われることがある。 例えば

```
call(void Point.setX(int)) && within(Line)
```

に結びつけられたアドバイスは、setX メソッドが Line クラス内で呼ばれたときだけ実行される。アスペクトはクラスを拡張するものであるが、サブクラスと異なり、様々な条件付きで拡張できるのである。上の例では、setX メソッドはアドバイスによって動作が拡張されるが、その拡張は呼び出し側が Line クラスであるときだけ有効となる。

AspectScope では、このような拡張の条件は省略せずに表示される。 setX メソッドの例の場合、図 4 のように ”only if the caller is Line” のように表示される。

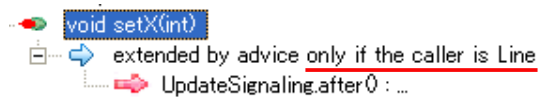


図 6: within を含む場合の AspectScope の表示

## 3.3　javadoc ビュー

AspectScope はクラスがアドバイスによりどう拡張されるかを説明するため、javadoc コメントを右側のパネルに表示する。今、図 7 のようなアスペクトがあったとする。このとき、Point クラスの setX メソッドの javadoc ビューは図 8 のようになる。

```
pointcut move() : call(void Shape+.set*(int))
                  || call(void Shape+.moveBy(..));
after() : move() && within(Line) {
        Display.update();
}
```

図 7: アドバイスとポイントカットの定義

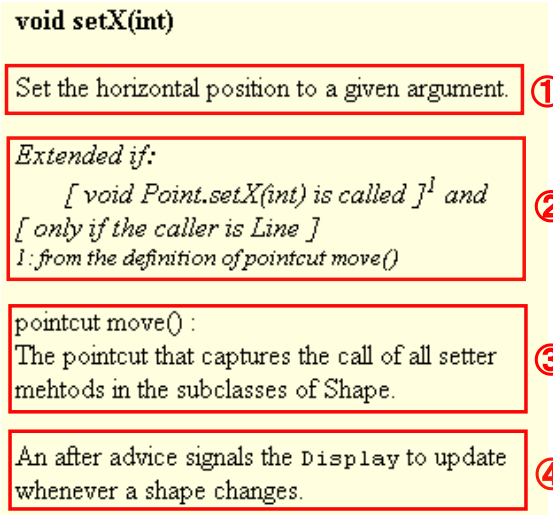


図 8: AspectScope の javadoc ビュー

表示されるコメントはプログラムのあちこちから集められたものである。①の囲みは setX メソッドからとった javadoc コメントで、setX メソッドの基本動作を示すものとして表示される。②の囲みは図 7 のポイントカットの定義を英文で表現したものである。ジョインポイントに現れる特殊文字の「*」や「+」はそれぞれ正しく書き換える。また、setX メソッドと関連のないポイントカット「call(void Shape+.moveBy(..))」は表示の際に無視する。これにより、setX メソッドがいつ拡張されるかが、ポイントカットの定義を直接見るよりも理解しやすくなっている。③は move ポイントカットからとった javadoc コメントで、アドバイスによる拡張が有効となるときを示す。Shape クラスのサブク

ラス内のすべての setter メソッドと moveBy メソッドの呼び出し時を意味する。④は after アドバイスからとった javadoc コメントで、setX メソッドがどのように拡張されるのかを示すものとして表示される。Shape クラスが変化するとき Display クラスの update メソッドが実行されることを意味する。

# 4 典型的なアスペクト指向プログラミングの例

この章では、AspectScope がカプセル化を保ちながら、アスペクトとクラスの関係を AJDT よりも詳しく表示することを示す。そのために、2つの具体的な例を用いて AJDT と AspectScope を比較する。

## 4.1 Observer アスペクト

最初の例は Observer アスペクトである（図 9）。これは Observser パターンをアスペクトを用いて書き換えたものである。今、図形エディタを作っているとし、図形を表す抽象クラスを Shape クラス、ユーザーが操作する画面を表すクラスを Display クラスとする。点や線を表す Point や Line クラスは Shape のサブクラスである。これら点や線の形が変わったときは、Display クラスの update メソッドを実行して画面を再描画しなければならない。この処理は横断的関心事であるためアスペクトとして記述するとよく、図 10 のようになる。ただし cflowbelow によって setter メソッドが他の setter メソッドを呼ぶ場合、アドバイスは最初の一度だけしか実行されない。

AspectScope は、Observer アスペクトを Shape

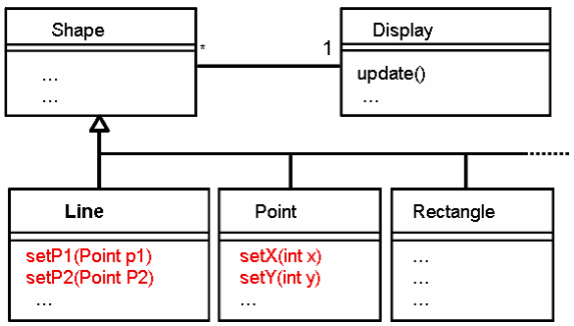


図 9: Observer アスペクト

```
pointcut change() : call(void Shape+.set*());
after() : change() && !cflowbelow(change()) {
    Display.update();
}
```

図 10: アドバイスとポイントカットの定義

とそのサブクラスの動作を拡張するものとして視覚化する。例えば、Line クラスの setP1 メソッドはアスペクトによって拡張されていることが表示され、javadoc ビューには呼ばれたときには同時に update メソッドが呼ばれることが表示される。また update が呼ばれるのは最初の 1 回だけであることも表示される。一方、AJDT では setP1 メソッドを呼び出している呼び出し側の行にジョインポイントのシャドウを示すマークが表示されるだけであり、cflowbelow の情報は表示されない（図 12）。

*Extended if:*

*[ void Point.setX(int) is called ][1] and [ not below the control flow of the call to Point.setX(int) ]*

*1 : from the definition of pointcut change()*

pointcut change() :
The pointcut that captures the call of all setter mehtods in the subclasses of Shape.

An after advice signals the `Display` to update whenever a shape changes.

図 11: AspectScope による表示

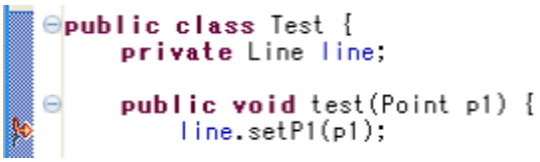


図 12: AJDT エディタによる表示

## 4.2 Logging アスペクト

次は、Logging アスペクトの例である。ログ出力アスペクトとして、Canvas クラス内で Graphics クラスの draw メソッドを呼び出すときにログを出力することを考える（図 13）。AJDT は Canvas クラスの中で draw メソッドが呼び出されている箇

```
aspect LoggingAspect {
  before() : call(* Graphics.draw(..))
                  && within(Canvas) {
      System.out.println("log");
}
```

図 13: Logging アスペクト

所、つまりメソッドの呼び出し側にマークを表示する（図 14）。一方、AspectScope では呼ばれる側の Graphics クラスの draw メソッドがアスペクトによって拡張されているという表示になる（図 15）。

Logging アスペクトの場合、AJDT の表示方法の方がより望ましい情報を与えているといえる。開発者が知りたいのは、どの箇所でログ出力が行われるかだからである。一方、AspectScope ではカプセル化が保たれている。図 15 の下線部分のように、Canvas からの呼び出し時のみログ出力するよう、アスペクトによって拡張されていることが分かるので、draw を呼ぶ側のメソッド（例えば paint メソッド）の中を見る必要はない。AspectScope の表示だけで、アスペクトの効果を理解できる。

```
public class Canvas {
    Image image;

    public void paint(Graphics g) {
        if (image != null)
            g.draw(image);
        else {
```

図 14: AJDT エディタによる表示

*Extended if:*
*[ void Graphics.draw(Image) is called ] and*
*[ only if the caller is Canvas ]*

図 15: AspectScope による表示

# 5 関連研究

AspectScope は Kiczales らの論文で発表された Aspect-Aware Interface [3] と基本的な考え方は同じである。しかし、この論文では call、get、set ポイントカットに関しては何の言及もしていない。また、javadoc コメントに関する考察もない。それに対して AspectScope はアスペクトによって拡張されるモジュールに対しより具体的な表現方法を提案したツールである。

また、Classbox/J [1] はある条件下でのみ有効なクラスの拡張を定義できるという考えが AspectScope と類似している。しかし、Classbox/J はアスペクト指向言語ではない。

# 6 まとめ・今後の課題

## 6.1 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 は呼ばれる側のクラスがどのように拡張されたかをアウトライン・ビューの形で表示する。また、拡張の詳細を関連する javadoc コメントを集めるて表示することで開発者に知らせる。それにより、プログラムのカプセル化を保ったまま、アスペクトとクラスの関係を視覚化できる。

## 6.2 今後の課題

Logging アスペクトの例（4.2 節）では呼ぶ側のクラスが拡張されたと示す表示の方が利用者の利便性が高かった。しかし、このようにするとカプセル化が損なわれる。今後は 呼び出し側を拡張したと見せる利便性とカプセル化による保守性との両方を満足する視覚化方法を検討していく必要がある。

# 参考文献


[1] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[2] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[3] 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.