アスペクト指向言語のための独立性の高いパッケージシステム

今吉 竜之介　　柳澤 佳里　　千葉 滋

東京工業大学 情報理工学研究科 数理・計算科学専攻

概 要

本稿では、アスペクトの織り込みに制限を設けたシステムを提案する。このシステムでは、原則としてアスペクトは同じパッケージ内のジョインポイントしか選択することができない。この制限により、パッケージの独立性は向上する。その一方で、本システムではアスペクトの記述力を保つために、選択可能なジョインポイントの範囲を明示的な記述によって拡大することもできる。abc を拡張して本システムを実装し、AJHotDraw に適用して考察する。

1　はじめに

AspectJ [5] にはアスペクトの織り込みに対する制限が存在しない。そのため誤った方法でアスペクトを使用すると、システム内のまったく関係の無い部分に予期せぬ影響を与えてしまう恐れがある。このようなシステムは独立性が低い。なぜならアスペクトは任意のクラスの挙動を変更することが可能なため、どんな局所的なプログラムもその挙動を保障することができなくなってしまうからである。

そこで我々はシステムの独立性を向上させるために織り込み制限を提案する。この制限によってアスペクトは関係の無いモジュールに影響を及ぼすことがなくなり、プログラムの挙動は保障される。また、この制限の元ではアスペクトと影響を受けるクラスは常に同じコンポーネントに含まれるため、コンポーネントの再利用性が向上する。なぜなら再利用の際にクラスとアスペクトが切り離されることが無くなるからである。ただし制限には例外ルールが存在し、明示的に記述することでアスペクトの影響範囲を拡大させることもできる。

我々は abc(AspectBench Compiler)[4] を改造することで織り込み制限を持つシステムを実装した。また、その実装を使って実験をおこない、織り込み制限の有無によるソースコードの変化を調べ、変化の前後で生じた利点や副作用について考察した。

以下では、２章でアスペクト指向プログラミング (AOP) におけるコンポーネント間の問題点を具体例を挙げて説明し、３章ではその問題を解決するための織り込み制限を設けたシステムを提案する。４章ではそのシステムの実装について述べ、５章では実験について説明する。６章では関連研究を挙げ、７章でこれらをまとめる。

2　コンポーネントに含まれるアスペクト

アスペクトを用いることで横断的関心事をまとめて記述することができる。しかしアスペクトは不注意な使われ方をすると、プログラマの予期せぬ織り込みを引き起こし、それに伴ってバグの発見や修正が必要になることもある。例として描画ソフトを複数人で共同開発する状況を考える。必要な機能はコンポーネントとして分離され、それらの作成は全て異なる開発者が担当する。

図 1 は FigureComponent 内のクラスを示している。FigureComponent は図形の状態を管理するコンポーネントであり、図形の状態が変化した時には再描画の通知をおこなう。各図形には色があり、Color でそれぞれの色を表す。setColor は色を変更するためのメソッドである。ここで開発者が色の変更直後に再描画を通知するために、図 2 のようなアスペクトを用いたとする。(1) のポイントカットが示すのは『setColor メソッドが実行されたとき』である。このアスペクトにより FigureComponent は、色が変更された直後に Display.update() を呼んで画面を再描画するようになる。FigureComponent の実装は図 1 と図 2 のようなプログラムとなり、FigureComponent を単体テストしても再描画処理に問題が発見される

```
package FigureComponent;

class Point {
  Color c;
  public void setColor(Color c) {
    this.c = c;
  }
}

class Circle {
  Color inside_c, line_c;
  public void setColor(Color c1,
                       Color c2) {
    this.inside_c = c1;
    this.line_c = c2;
  }
}

class Panel {
  Color frame_c;
  public void setColor(Color c) {
    this.frame_c = c;
  }
}
```

図 1: FigureComponent 内のクラス

```
package FigureComponent;
import WindowComponent.*;

public aspect UpdateDisplay {
  pointcut pc_changeColor() :
  execution(void *.setColor(..)); // (1)

  after() : pc_changeColor() {
    Display.update();
  }
}
```

図 2: FigureComponent 内のアスペクト

ことはない。

一方で、FigureComponent を作成した人物とは別の開発者が、描画ウィンドウを管理する WindowComponent を図 3 のように記述していたとする。Display は、setColor メソッドで自身のフレームカラーを変更する。WindowComponent では、Display のフレームカラーが変更された場合に再描画をさせるために setColor メソッドの中で update を呼んでいる。

WindowComponent を単体テストした場合、フレームカラーの変更とそれに伴う再描画は正しく実行される。FigureComponent も WindowComponent も正しく動作することが個別に確認されるため、開発は次の段階であるコンポーネントの結合へと進む。

FigureComponent と WindowComponent を組み合わせると、Display のフレームカラーが変わった時に

```
package WindowComponent;

public class Display {
  Color frame_c;
  public void setColor(Color c) {
    this.frame_c = c;
    update();
  }

  public static void update() {
    // 再描画
  }
}
```

図 3: WindowComponent 内のクラス

画面がちらつくという現象が起こる。なぜなら先程 FigureComponent の開発者が書いた図 2 のアスペクトにより、WindowComponent の Display クラス内にある setColor メソッド実行直後にもう一度 update が呼び出されるからである。アスペクトが (1) で指定したのは『setColor メソッドが実行されたとき』であり、本来なら FigureComponent とまったく関係の無いモジュールであった Display の setColor メソッドも偶然名前が一致してしまうためにこのような問題が発生してしまう。

アスペクトを含むコンポーネントを作成した場合、各々の単体テストでは正常に動作しても、それらを組み合わせた途端に個々のコンポーネントの挙動が変わってしまう恐れがある。これはアスペクトが複数のコンポーネントをまたいで影響を与えてしまうために起こる問題である。このような予期しない織り込みに関する問題は Fragile Pointcut [6] と呼ばれている。影響を受けたコンポーネント側にはその原因となるアスペクトが内部に存在しないため、プログラムの修正がより困難になる。既存の AspectJ のような言語には、このような複数のコンポーネントを対象とした織り込みの制約が存在しない。

3 織り込み制約の提案

2 章で述べた問題点を解決するため、織り込みに対して制限を設けたシステムを提案する。本システムのルールは、アスペクトは同じパッケージに存在するジョインポイントだけを織り込みの対象として選択可能にするというものである。ここではパッケージをコンポーネントの単位と見なしている。この制限下では、アスペクトがパッケージ外のジョイ

ンポイントを選択することは許可されない。

２章のソースコードについて考える。図２の (1) のポイントカットが選択するジョインポイントは AspectJ では『setColor メソッドが実行されたとき』である。本システムでは、これに『そのアスペクトが含まれるパッケージ内に存在』という条件が追加される。これはポイントカットの宣言文に『&& within(FigureComponent.*)』が付加されるようなものであり、この制約によりパッケージの異なる Display クラスにはアドバイスが織り込まれなくなる。

3.1 本システムの利点

本システムでは名前の条件が一致していてもパッケージが異なればアドバイスは織り込まれない。２章で述べた２つのコンポーネント間に起こる問題はこの制約によって解決される。外部アスペクトからの影響が無いため、WindowComponent の開発者は自分のプログラムを保守しやすい。また、ワイルドカードの使用による誤ったジョインポイントの選択も防ぐので、FigureComponent の開発者は外部クラスのメソッドやフィールドの名前を考慮することなくアスペクトを記述できる。

本システムの制約により各パッケージの独立性は向上する。本システムでは、全てのアスペクトは関心のあるプログラムと同一のパッケージに属している。そのためパッケージの再利用時に予期せず挙動が変わってしまうことが無い。なぜならそのパッケージに影響を与えるアスペクトがパッケージ外部には存在しないため、開発者が気付かずにアスペクト抜きでパッケージを再利用することが無いからである。また、本システムではアスペクトがパッケージ外部に予期せぬ影響を与えることが無い。そのためアスペクトを含むパッケージを再利用しても、そのアスペクトが移動後のシステムに存在する関係の無いクラスに作用したりすることは無い。

3.2 例外ルール

原則として本システムでは、アスペクトとジョインポイントが共に同一パッケージ内になければアドバイスは織り込まれないが、プログラムで明示すればジョインポイントで選択できる範囲をパッケージの外部まで拡大できる。本システムでは、その方法として『インターフェイスに対するアドバイス』と『公開ポイントカット』の２種類の機能を提供する。

```
package FigureComponent;

aspect AssertAspect {
  pointcut pc_callSetColor() :
  call(void *.setColor(..))
    && !within(FigureComponent); // (2)

  before() : pc_callSetColor() {
    // assertion の実行
  }
}
```

図 4: AssertAspect

3.2.1 インターフェイスに対するアドバイス

call,set,get ポイントカットは呼び出し側の式をジョインポイントとして選択するので、本システムではパッケージ外部からの呼び出しをアスペクトで選択できない。そこで、『自パッケージ内のメンバへの call,set,get』を表すポイントカット記述に追加条件で『&& !within(自パッケージ)』が記述された場合は、指定したメンバへの外部からのアクセス全てに対してアドバイスが織り込まれることにした。このルールを『インターフェイスに対するアドバイス』と呼ぶ。これは、パッケージ外部からのアクセスを対象としたルールで、内部からのアクセスには何も影響しない。また、このルールでは外部からの特定のアクセスだけを選択することは許可しない。

具体的な記述は図 4 のようになる。これは、パッケージ外部から setColor を呼び出す直前に必ず assertion を実行させるためのアスペクトである。(2) のポイントカットは、『setColor を呼び出すとき』かつ『FigureComponent の外部から』という条件を示している。AssertAspect は FigureComponent の内部に存在するアスペクトなので、このポイントカットには例外ルールが適用される。よって図 4 のアスペクトは、『パッケージ外部から setColor が呼び出される直前』にアドバイスを織り込むようになる。

この例外ルールにより、同じアスペクトが複数個所に重複して散在するのを防ぐことができる。本システムでは、１つのアスペクトが複数のパッケージに同時に作用することは許可されていなかった。そのため同じアドバイスを複数のパッケージに織り込みたい場合には、アスペクトをそれぞれのパッケー

ジ毎に作成する必要があった。本例外ルールを用いることで、そのようなアスペクトを１箇所にまとめて記述できる。

この例外ルールは、パッケージの独立性を低下させるようなものではない。パッケージの外部から見た場合、このようなアスペクトはパッケージインターフェイスの処理の一部と見なすことができる。それは外部から内部メンバへのアクセス全てに同一に作用するため、外から見た挙動が一定となるからである。

もし、特定のアクセスだけを選択可能にすると、パッケージの再利用性を損なう危険性が生じる。なぜなら一部のクラスからのアクセスに対する挙動だけをアスペクトで変化させることが可能になるため、外部からのアクセス全てに対する挙動が一定だという保障が無くなるからである。そのパッケージの開発に関わっていないプログラマは、アスペクトで拡張された特定のアクセスに対する挙動だけを見て、それが全てのアクセスに対しても同様に行われるものだと誤解するかもしれない。そのプログラマがパッケージを別のシステムに再利用した場合、移動前に期待していた処理と移動後にそのパッケージが実際に行う処理との間に食い違いが生じる恐れがある。このような問題を防ぐ必要があるため、本例外ルールでは特定のアクセスだけを選択することを禁止し、本システムの利点を損なわないようにしている。また、この例外を適用するには明示的な記述が必要なため、予期しない織り込みが行われることは無い。

3.2.2 公開ポイントカット

ポイントカットを定義している抽象アスペクトを継承することで、子アスペクトからそのポイントカットが選択するジョインポイントにアドバイスを織り込むことができる。このルールを『公開ポイントカット』と呼ぶ。この例外ルールを用いることで、アスペクトはパッケージ外部の特定のジョインポイントを選択できるようになる。本システムでは、引き継いだポイントカットを子アスペクトが使用する場合には、そのアスペクトは親アスペクトと同一のパッケージに所属していると見なす。

図 5 はポイントカットを定義している抽象アスペクトで、図 6 はそれを継承している子アスペクトである。UndoAspect は WindowComponent の外部に

```
package WindowComponent;

public abstract aspect UpdateAspect {
  protected pointcut pc_update() :
  execution(void Display.update()); // (3)
  }
}
```

図 5: UpdateAspect

```
import WindowComponent.UpdateAspect;

aspect UndoAspect extends UpdateAspect {
  before() : pc_update() {
    // Undo のための履歴の保存
  }
}
```

図 6: UndoAspect

存在しているが、この例外ルールにより、(3) のポイントカットが選択するジョインポイントにアドバイスを織り込むことができる。なぜなら UndoAspect が定義したアドバイスは、(3) のポイントカットを利用しているため、親アスペクトである UpdateAspect と同一のパッケージ（WindowComponent）に属していると見なされるからである。

本例外ルールを適用すれば、プログラムのイベントハンドラを、そのプログラムが属するパッケージの外にあるアスペクトからも扱うことが可能になる。例えば、あるパッケージ内部で行われる処理と同期させて外部の関数を呼び出したい場合は、抽象アスペクトに同期のタイミングを定義し、外部の子アスペクトにその処理を実装することでこれを実現できる。公開ポイントカットは、アスペクト指向を用いたオブザーバーパターンの実装などに役立つ。

この例外ルールは予期せぬ織り込みを引き起こさない。なぜなら、この例外を適用するには両方のパッケージに利用を明示する必要があるからである。つまり、公開元では抽象アスペクトを明示的に定義し、公開先ではアスペクトを明示的に継承する必要がある。

3.3 議論

本システムのルールと同等のことは、AspectJ でポイントカット記述に常に within を明記するコード規約を行うことで可能に見える。しかし、条件指定

を間違えた場合にそれを発見する機構が AspectJ には存在しない。また、ポイントカットを定義する毎に within を記述するのは手間が掛かる上に、記述し忘れる問題がある。我々は AspectJ の言語仕様を変更し、デフォルトで条件文を付加するようなシステムを考案した。本システムでは暗黙のうちに正しい条件文が付加されるため、このような問題は発生しない。

２つの例外ルールを AspectJ へのトランスレータとして実現するのは難しい。それらが正しく使用されているならば AspectJ で再現することは可能だが、中には例外ルールが間違った方法で使用される場合もあり得る。本システム下では間違った使用法を検出した場合はそれをエラーとして処理することが可能である。しかし AspectJ ではエラー検出機構までは再現できない。

4　実装

我々は、３章で述べたようなルールを実装するために abc を改造した。abc は **AspectJ**[5] の仕様を完全にサポートするコンパイラで、polyglot と soot という二種類のフレームワークを利用して作成されている。AspectJ との違いは拡張や最適化に優れていることと、ジョインポイントとして選択できるポイントの種類が増えたことである。

4.1　abc の構造

アスペクトを含むプログラムをコンパイルしたとき、abc は『Java 言語部分のプログラムのコンパイルとアスペクトの解析』を行う。プログラム中に文法に沿っていないようなコードが含まれていた場合、コンパイラはここでエラーを出して終了する。アスペクトはここでコードを解析され、abc はポイントカットやアドバイスの情報を取得する。

次に abc は全てのクラスの全てのジョインポイントを一つずつ調べ、その部分に一致するポイントカット記述を持ったアドバイスが存在するかをチェックする。アドバイスが見つかった場合、そこに織り込む予定のリストにそれを追加するだけで織り込み自体はすぐには行わない。このように abc はポイントカットを見てから一致するジョインポイントを検索するのではなく、ジョインポイントを見てから一致するポイントカットを検索する。アドバイスを実際に織り込む場合は、同様に全てのクラスの全てのジョインポイントを調べながら、そこに対応するリストから織り込みを行う。

4.2　織り込み制限

本システムのルールは、全てパッケージ名が基準となっている。abc 内部では、アドバイスは自分を記述しているアスペクトの情報を持ち、アスペクトは自身が属するパッケージの情報を持っている。本システムでは、織り込みを行う直前にアドバイスのリストを全てチェックして制限を加えている。『アドバイスが記述されているアスペクトが属するパッケージ名』と『ジョインポイントが属するパッケージ名』を比較し、それらが一致すればリストに残したまま（織り込み可）にする。abc は call,get,set ポイントカットを特殊な形式と見なしている。これらのポイントカットから得られる情報は他のポイントカットの場合と多少異なり、『呼ばれるクラス』と『呼び出し元』の２種類が含まれる。これらのポイントカットが選択するジョインポイントは呼び出し先ではなく呼び出し元のほうである。また、パッケージ名が一致しなくとも、例外ルールを満たしている場合はそのアドバイスはリストに残しておかなければならないことに注意する。

5　実験

この実験では、AspectJ で書かれたソースコードと、本システムでのソースコードの差異を計測した。実験対象には **AJHotDraw**[1](ver 0.3) を用いた。AJHotDraw は AspectJ の文法で書かれたオープンソースの描画アプリケーションである。

我々は、まず元のソースコードを AspectJ でコンパイルした場合と同じ挙動を示すように AJHotDraw を本システム用に修正した。この修正が必要な理由は、本システムの制限下では元のソースコードにはアスペクトが織り込まれない箇所が存在するため、異なる挙動を示すからである。

ここではクラス側のソースコードには一切触れず、アスペクトだけを修正した。アスペクトに関しても、アドバイスの内部などは一切変化させていない。そして、コンパイルした時に織り込まれたアド

表 1: AJHotDraw のソースコードの比較

	修正前	修正後
アスペクトを含むパッケージ数	3	7
アスペクトのファイル数	10	17
記述されたアドバイスの数	5	5
親クラス変更アスペクトの数	10	10
メンバ追加アスペクトの数	21	21
名前付きポイントカットの数	3	3
織り込まれたアドバイスの数	44	44

バイスの数や適用されたインタータイプ宣言の数が修正前後で変化しないことで挙動が同じであると判断した。

修正前後のソースコードを比較した結果を表 1 に示す。実験の結果、アスペクトを含むパッケージの数とアスペクトのファイル数が修正後にはそれぞれ増加していることが分かった。修正前の状態では、アスペクトは 3 つのパッケージにまとめて記述されていた。それらのパッケージはアスペクトのみを含み、本システムでは外部のプログラムに対して一切の織り込みが許可されない。それらのアスペクトをプログラムに作用させるために各ファイルを移動させた結果、それらの数が増加した。複数のパッケージに影響を与えるアスペクトが存在する場合など、修正後にポイントカットやアドバイスの数が増加することも考えられるが、今回のケースでは増加したのはアスペクトの数だけとなった。本実験により、本システムの制限下で書けなくなるアスペクトは AJHotDraw には無く、問題なくアスペクトが利用できることがわかった。また、本実験では例外ルールを使わなくては書けないところも存在しなかった。

6 関連研究

プログラムに予期せぬ影響を与えないようにする技術はすでにいくつか提案されている。**OpenModule**[2][7][3] や **ccJava**[8] などはプログラムをアスペクトの影響から守り、その挙動を保障するための研究である。一方、本システムはアスペクトが選択可能なジョインポイントの範囲を制限するものであるが、挙動の保障だけでなく、パッケージ単位での独立性も考慮している。

Open Module

Open Module は、『module』という単位でクラスをまとめ、外部のアスペクトからのアクセスを制御する技術である。module は外部に任意のジョインポイントを公開することができる。しかし公開されたジョインポイントは織り込みに対して一切の制限がなくなるため、関係の無いメソッドと名前が偶然一致した場合に誤ってそのジョインポイントが選択されてしまう恐れがある。

本システムではパッケージ外部に公開するのは名前付きポイントカットであり、公開側も継承側も明示的な記述が必要なため、誤った選択が行われることは無い。また、OpenModule ではプログラムと関連のあるアスペクトが同じパッケージに含まれる必要が無いため、アスペクトを含むパッケージの再利用性については考慮していない。

ccJava

ccJava はアスペクトの織り込みに対応した『織り込みインターフェイス』を導入している。全てのアスペクトはインターフェイスを介してプログラムに織り込まれ、インターフェイスで定義されていない箇所にはアドバイスを織り込むことができない。

ccJava は、クラスやアスペクトの修正の際に、織り込みインターフェイスの分だけ扱うコード量が増加するため、管理が困難になる。また、パッケージの再利用時には、織り込みインターフェイスの修正が必要になる。

7 まとめ

我々はパッケージの独立性を高めるため、アスペクトの織り込みに制限を設けたシステムを提案した。AspectJ では、アスペクトが影響を及ぼす範囲に制限は課せられていなかった。しかし本システムでは、アスペクトが関係の無いプログラムに予期せぬ影響を与えることは無い。本システムが提案する制限のルールは、アスペクトとジョインポイントが同じパッケージ内に存在する場合のみアドバイスの織り込みを許可するというものである。

本システムは、AspectJ を拡張した abc を改造することで実装した。また、実験では AspectJ で書かれた AJHotDraw を本システムのルールに対応する

ように修正し、修正前後でのソースコードを比較した。その結果、アスペクトのファイル数は増加するが、パッケージ再利用時にアスペクトを付加させる手間が無くなったことを確認した。

今後の課題として、様々なアスペクトに織り込み制限を課した際の副作用の検証と、細かいルールの徹底などが挙げられる。また、今回の実験ではアドバイスやポイントカットの数が修正前後で不変だったため、これらの数が変動するような場合についても考察する必要がある。

参考文献

[1] AJHotDraw. http://sourceforge.net/projects/ajhotdraw/.

[2] J. Aldrich. Open modules: Reconciling extensibility and information hiding. In *In AOSD workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT '04)*, March 2004.

[3] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *In Proceedings of the European Conference on Object-Oriented Programming, volume 3586 of LNCS*, pages 144–168, 2005.

[4] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: The AspectBench Compiler for AspectJ. In Robert Glück and Michael Lowry, editors, *Generative Programming and Componenent Engineering (GPCE)*, Lecture Notes in Computer Science. Springer, 2005.

[5] AspectJ. http://www.eclipse.org/aspectj/.

[6] C. Koppen and M. Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In *in European Interactive Workshop on Aspects in Software*, 2004.

[7] 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 Press.

[8] 玉井 哲雄 境 顕宏, 鵜林 尚靖. Aop 言語への織り込みインターフェイスの導入. In **日本ソフトウェア科学会第** *23* **回大会**, 2006.