平成19年度 学士論文

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

東京工業大学 理学部 情報科学科
学籍番号 03-0315-0
今吉 竜之介

指導教員
千葉 滋 助教授

平成19年2月16日

概要

本研究は、アスペクト指向プログラミングで書かれたプログラムにおいて、アスペクトに適度な制限を設けたパッケージ機構を提案する。アスペクト指向プログラミングとは既存のモジュールを拡張するための技術で、ロギングや同期処理、イベント通知などのような複数のモジュールにまたがって散在する横断的関心事をまとめて扱うことができる。本研究では、Java パッケージによるアクセス制御機構をアスペクトの織り込みに対して適用させた。この制約によって、関係の無いパッケージへのアスペクトの予期せぬ影響を回避できる。アスペクトをプログラム中に織り込む処理のことをウィーブと呼び、プログラム中でウィーブされるポイントのことをジョインポイントと呼ぶ。本研究が提案するシステムがアスペクトの織り込みに対して行う制限とは、アスペクトが属しているパッケージとジョインポイントが属しているパッケージが一致しない限りウィーブは行われないというものである。本システムはこの制約を基本ルールとしているが、さらに２つの例外ルールが存在する。それは、パッケージ外部に対して公開されているジョインポイントに対してはパッケージ外のアスペクトであってもウィーブが可能というルールと、自パッケージ内のクラスメンバが任意の外部パッケージからアクセスされるときならばそのプログラムに自パッケージ内のアスペクトも作用可能というルールである。これらの例外を適用させる場合、プログラマは明示的にコードを記述しなければならない。そのため、これらのアスペクトがプログラムに予期せぬ影響を与えたりすることは無い。

本研究が提案するシステムの制限下では、アスペクトは限られた部分にしかアクセスできない。そのためプログラムの挙動の予測が容易になり、パッケージの再利用時の副作用が軽減される。なぜなら、関心のあるプログラムに対してのみアスペクトが作用するため、再利用されたパッケージ内のアスペクトがシステムの別の部分に影響を与えたりすることが無くなるからである。また、アスペクトと、関係のあるプログラムは同一パッケージに含まれていなければならないため、再利用するときにそれらが切り離されることが無くなる。そのため再利用の前後で一部のアスペクトが誤って抜け落ち、プログラムの挙動が変化してしまうようなことは無い。

従来のアスペクト指向プログラミングでは、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ため、プログラムの挙動の予測は難しかった。これらの問題を解決する技術としては、例えばOpen ModulesやccJavaなどがすでに存在する。これらの技術は、アドバイスの無制限なウィーブに何らかの制約を課すことで、プログラムの挙動を保障するものである。しかしこれらは、アスペクトを含むようなシステム内のモジュールの再利用については考慮していない。なぜならアスペクトと、それによって影響を受けるプログラムが同じモジュール内に存在している必要が無いからである。このような場合、モジュールを別のシステムに移行させた際に、影響を与えていたアスペクトが切り離されて、移行前と移行後でモジュールの挙動が変わってしまう可能性がある。

本研究で提案するシステムは、AspectJを拡張したコンパイラであるabc(AspectBench Compiler)を改造することで実装した。これを用い、アスペクト指向で設計されたアプリケーションであるAJHotDrawをパッケージ制限が付加されたルールに対応するように修正し、その修正前後でのアスペクトの情報を比較した。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その部分に影響を及ぼすアスペクトを付加させる手間が無くなることを確認した。さらに実験により、abcの改造の前後でコンパイル時間がほとんど変化しないことを確認した。

謝辞

本研究を進めるにあたり、研究の方向性や論文の構成などの多くの点にわたり指導して頂いた千葉滋助教授に心より感謝致します。

また、システムの設計や、論文の書き方、関連研究などの紹介を親身になって行って下さった東京工業大学の柳沢佳理氏、システムの実装にあたりその解析に助言を頂いた西沢無我氏、本論文のスタイルファイルを作成していただいた光来健一氏に感謝致します。

最後に、本研究を行う上で様々な指導、協力、激励を頂いた同研究室の皆様に感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

アスペクト指向プログラミングは、既存のモジュールを拡張するための技術である。オブジェクト指向プログラミングは様々な関心事をオブジェクトという単位でモジュール化する技術だが、多様化するアプリケーションの中には分離しにくい関心事が存在することが少なくない。例えばロギングや同期処理、イベント通知などがあり、これらの処理は複数のモジュールにまたがって散在してしまうため、モジュールとしての性質が損なわれてしまう。このように複数のモジュールにまたがって散在する処理のことを横断的関心事と呼ぶ。アスペクト指向プログラミングは横断的関心事をモジュールにまとめて扱うことができる技術である。アスペクト指向プログラミングでは、横断的関心事をアスペクトという単位でモジュール化する。横断的関心事の処理内容をアドバイスと呼び、アドバイスをプログラム中に埋め込む作業をウィーブと呼ぶ。

アスペクトからアドバイスがウィーブされた場合、クラス側のソースを読むだけではそれを知ることはできない。クラス側のプログラマは、その挙動がアスペクトによって変更されたことに気付かない場合がある。アスペクト指向プログラミングではアドバイスのウィーブに対して制限が存在しないため、アスペクトは任意のプログラムの挙動を変更できてしまう。これは逆に言えば、どんな局所的なプログラムであってもその挙動を保障できないということになる。

アスペクト指向の問題点として、上記のようなカプセル化破壊が挙げられる。この問題を解決する技術としては、例えばOpen Modulesやccなどがすでに存在する。前者はmoduleという新たな概念でクラスやアスペクトをまとめ、外部のアスペクトに対してはプログラムのどの部分を公開するか選択できる。そして非公開の部分へのウィーブを禁止することで、プログラムの挙動を保障する。後者はアスペクトとプログラムの間に「織り込みインターフェイス」という概念を導入し、そこで記述されたルールに従ってウィーブを行うというものである。これらの技術は、アドバイスの無制限なウィーブに何らかの制約を課すことで、プログラムの挙動を保障するものである。

しかしこれらの技術は、アスペクトを含むようなシステムでのモジュールの再利用については考慮していない。なぜならアスペクトと、それに

よって影響を受けるプログラムが同じモジュール内に存在している必要が無いからである。このような場合、モジュールを別のシステムに移行させた際に、影響を与えていたアスペクトが切り離されて、移行前と移行後でモジュールの挙動が変わってしまう可能性がある。

本研究では、アスペクトを含むようなシステムに、Java パッケージシステムを対応させることを提案する。ここではパッケージを再利用の単位と見なし、パッケージ外のプログラムに対してはウィーブが許可されないというルールを基点とするウィーブ制限を考える。これにより、システム内の関係の無い部分のプログラムにアスペクトが影響を与えたりしなくなる。また、プログラムとアスペクトが同一のパッケージに含められていることで、再利用時にアスペクトが切り離されることが無い。本システムは、以上のようなカプセル化と再利用性に優れたパッケージ機構を提供する。

本稿では、第 2 章でアスペクト指向プログラミングの問題点とそれに対する従来の研究について述べる。第 3 章で本研究が提案するパッケージ制限について記述し、第 4 章ではそのシステムを既存のアスペクト指向言語で設計されたアプリケーションに適用させ、本システムの有用性を示す。第 5 章では、本システムの副作用と今後の課題を述べ、第 6 章で本研究についてまとめる。

第2章　既存のアスペクト指向向けパッケージシステムの問題

2.1　アスペクト指向について

アスペクト指向プログラミング（AOP）とは、横断的関心事をモジュール化する技術である。ソフトウェアを設計する場合、開発すべきシステムをまず小さな機能（モジュール）ごとに分解し、それらを組み合わせることで一連の処理を実現する。関心事をモジュール化する技術としては、オブジェクト指向プログラミング（OOP）が有名である。OOP はソフトウェアの関心事をオブジェクトという単位で分離するが、アプリケーションの中には分離しにくい関心事がいくつか存在する。その例としては、ロギングや同期処理、イベント通知などが挙げられる。これらの処理は通常、プログラム中のあらゆる場所に散らばってしまうため、横断的関心事と呼ばれる。横断的関心事は一つのオブジェクトとしてモジュール化することができない。また、そのようなコードを含んだプログラムは、モジュールとしての性質が欠けてしまう。他にも、横断的関心事のコードを修正したいときに、それらがプログラム中に散在しているために、その作業が困難になったり、予期せぬバグを生み出したりする危険がある。AOP は、このような OOP の欠点を補う技術である。

2.2　AspectJ

AspectJ[5] は汎用的な AOP 言語の一つで、OOP の一種である Java にアスペクト指向を取り入れて拡張したものである。数多く提案されている AOP 言語の中でも AspectJ は代表的なもので、現在もっとも幅広く使われている。

2.2.1　joinpoint

joinpoint はアスペクトを織り込むことが可能な、あるプログラムを実行する「とき」を指している。AspectJ におけるジョインポイントの種類としては、メソッドやコンストラクタの呼び出す時や実行する時、フィー

ルドの参照や書き込みをする時、例外ハンドラを実行する時など多数存在する。

2.2.2　pointcut

joinpoint を選出する作業のことを pointcut という。プログラム中にある様々な joinpoint のうち、どの joinpoint を指定したいかを具体的に記述する。指定した joinpoint でどのような処理を実行するかは、pointcut には記述できない。

例えば、「Point クラス内で宣言されていて、名前が getX、返り値が int 型の関数」を「呼び出すとき」という joinpoint を選出したい場合は、AspectJ では次のように記述する。

```
call(int Point.getX())
```

この call は pointcut 指定子と呼ばれているものの一つで、これはメソッドを呼び出すときの joinpoint を対象としている。そして続く括弧内に条件を記述して、プログラム内のどのメソッド呼び出しを選出するかを指定している。

AspectJ には以下のような pointcut 指定子が存在する。

call メソッドを呼び出すときを選択

set フィールドの値に書き込むときを選択

get フィールドの値を読み込むときを選択

initialization クラスのインスタンスを生成するときを選択

execution メソッドを実行するときを選択

handler 例外ハンドラを実行するときを選択

staticinitialization クラスの静的初期化子を実行するときを選択

これらの pointcut 指定子は、joinpoint の種類を選択としているが、joinpoint の位置を選択する指定子もある。例えば、「Line クラスの中」にある joinpoint を選出したい場合は、次のように記述する。

```
within(Line)
```

within は pointcut 指定子で、これは特定のクラスの中にある joinpoint を選択する。括弧内には、そのクラス名を記述する。joinpoint の位置を選択する pointcut 指定子には以下のようなものが存在する。

within 指定したクラスまたはパッケージ内にある joinpoint を選択

withincode 指定したメソッド内にある joinpoint を選択

pointcut には、プログラムの実行時の情報を基にしたものも存在する。以下の 3 種類の pointcut は joinpoint の情報を取得するために使われる。

target joinpoint が対象としているインスタンスの情報を取得

this 指定した joinpoint が実行されているインスタンスの情報を取得

args joinpoint において変数を取り込む場合に、その変数の情報を取得

さらにプログラムのフローを利用する pointcut も存在する。

cflow 指定した制御フローの中にある joinpoint を選択

cflowbelow 指定した制御フローの中にある joinpoint を選択。ただし、指定された演算自体は含まない

if 指定した条件を満たしている状態の joinpoint を選択

pointcut 指定子には、一度に複数の joinpoint を選択できるように、ワイルドカードを使用できるものがある。例えば以下のような使用方法がある。

```
call(public void Point.*(..))
```

これは、「Point クラス内で宣言された public で、名前や引数は任意、返り値が void のメソッド」を「呼び出すとき」に一致する joinpoint を選択する。ここで [*] は.（ドット）を含まないような任意の文字列を表している。引数部分の [..] は、「引数は何でもよい（引数が無くてもよい）」という意味である。

また、AspectJ では複数の pointcut を論理演算子で組み合わせることで、より細かく joinpoint を選出することができる。演算子には、&&（論理積）、——（論理和）、!（否定）の 3 種類がある。例えば、以下のように記述する。

```
call(public void Point.*(..)) && !within(Point)
```

これは「Point クラスの外」からメソッドが呼ばれたときを選択する。

2.2.3　advice

pointcut 記述で選択された joinpoint において、新たに実行させたいプログラムのことを advice という。advice には、主に以下のような種類がある。

before joinpoint の命令が処理される直前に織り込む

after joinpoint の命令が処理された直後に織り込む

around joinpoint の命令に置き換えて織り込む

AspectJ で、joinpoint に advice を織り込みたい場合は、例えば以下のように記述する。

```
before() : call(int Point.getX()) {
  System.out.println("before call Point.getX()");
}
```

このアドバイスは、Point クラスの getX メソッドを呼び出す直前に文字列を出力させる。一行目の before() の部分がタイミングの指定、コロンの後が pointcut 記述である。括弧の中には織り込みたい命令（advice body）を Java のコードで記述する。

2.2.4　named pointcut

pointcut には名前が付けられる。これはメソッドに名前が付けられるのと同じで、複数の advice に同じ pointcut 記述を書かなくて済むようになる。その記述は例えば以下の通りである。

```
pointcut PgetX() : call(int Point.getX());
```

このように記述すると、[call(int Point.getX())] の代わりに [PgetX()] と書くだけで同じ joinpoint を選択できるようになる。

2.2.5　インタータイプ宣言

また、AspectJ には、advice 以外にも、インタータイプ宣言というものが存在する。例えば、以下のような記述をする。

```
declare parents : A extends B;
declare parents : C* implements D;
```

これらは、あるクラスの親クラスやインターフェイスを変更する命令である。最初の命令はクラス A の親クラスをクラス B にする記述である。二行目の記述は、名前が C から始まるクラスのインターフェイスを D に変更する命令である。インタータイプ宣言で、インターフェイスを変更ではなく追加したい場合は、[implements] の後に元々のインターフェイスも記述しなければならない。インタータイプ宣言で記述されなかったインターフェイスは、たとえクラス側に宣言されていたとしても無視される。

親クラスの変更には条件がある。変更したいクラス X の直接の親クラスを Y、新たに親クラスとして指定したいクラスを Z とする。このとき、Z は Y を継承している必要があり、この条件を満たしていなければこのインタータイプ宣言はコンパイルエラーとなる。

また、アスペクト側からクラスに新たなメソッドやフィールドを追加する命令も記述できる。抽象メソッドを含むようなクラスを継承する場合、子クラスはそのメソッドを実装しなければコンパイルエラーとなる。インタータイプ宣言で親クラスが変更された場合も同様であるが、例えば以下のような宣言をすることでそのようなエラーを回避できる。

```
private int Point.num;
public void Point.func() { ... }
```

一行目は Point クラスに num という名前の int 型の変数を追加する命令である。その下の命令は、Point クラスに func というメソッドを追加し、さらにその処理を記述したものである。

インタータイプ宣言されたフィールドやメソッドは、可視性なども含めて、本来のクラスに宣言されている場合とまったく同等に扱われる。AspectJ では、コンパイル時にクラス側の情報が書き換えられるので、クラス側のコードから追加されたフィールドやメソッドを参照することもできる。

2.2.6　weave

アスペクト側からプログラム内のどこにどんな処理を織り込みたいかは、advice と pointcut、そして織り込みたいコードで記述する。アドバイスをプログラムに織り込む処理のことを weave という。weave のタイミングはコンパイル時、ロード時、実行時のいずれかだが、AspectJ ではコンパイル時に行う。

2.2.7　aspect

横断的関心事をモジュール化したもの。AspectJ における aspect は、pointcut、advice、インタータイプ宣言の記述が可能であること以外は、Java の class と同じように扱うことができる。aspect はフィールドやメソッドを持つことが可能で、class を継承することもできる。aspect の記述は例えば以下のようになる。

```
aspect SampleAspect {
  public static int Point.count = 0;

  private void print() {
    System.out.println(Point.count);
  }

  pointcut pc() : call(public * Point.*(..)) && !within(Point);
  before() : pc() {
    Point.count++;
    print();
  }
}
```

最初の [aspect] は、Java の予約語である [class] と同じ意味合いで使われる。aspect はメソッドやフィールドを持つことができ、アドバイスの中ではそれらを参照できる。また、別の aspect や class 側から参照することも可能である。

このコードは、Point クラスの public メソッドが Point クラスの外から呼ばれた回数を出力する。Point クラスに count という static フィールドを追加し、pointcut の pc() に一致する joinpoint を実行する直前にアドバイスの処理する。

この例では、Point クラス以外の全てのクラスの joinpoint が選択される。そのような場合でも、織り込みたい処理の内容やタイミングを変更したい場合は、アスペクト側を修正するだけよく、複数箇所に散らばったコードを修正したりすることはない。

2.2.8　aspect の継承

aspect は aspect を継承できる。しかし、これはクラスを継承する場合と条件が若干異なる。親として指定される aspect は、abstract aspect でなければならない。また、クラスと同様に、親として指定できる aspect

```
aspect DrawAspect {
  pointcut pointMovePc() : call(public void Point.move(..);

  after() : pointMovePc() {
    Draw.update();
  }
}
```

図 2.1: DrawAspect アスペクト

は 1 つだけである。aspect が aspect を継承すると、親アスペクト側の名前付き pointcut を子アスペクトも利用できるようになる。これはメソッドなどが継承されることと同じで、pointcut をオーバーライドすることも可能である。親アスペクトの advice は継承されない。

2.3　AOP の問題点

以下では、上記のような AspectJ の設計や言語仕様に従って議論を進めていく。AOP では、呼び出される側（織り込みたい処理を記述した側）がプログラムのどこから呼び出されるかを記述できる。この仕様により AOP には OOP には無かったいくつかの問題が生じている。

2.3.1　予測の困難さ

アスペクトは、局所的なプログラムの挙動の予測を困難にする。例えば、図 2.1 のようなアスペクトを考える。

これは Point クラスの move メソッドを呼び出した直後に、アドバイスで再描画を行うように記述したアスペクトである。このアスペクトにより、Point の位置が更新された際に、必ず最描画も行われるようになるので、プログラマが再描画の呼び出しを気にせずにプログラムを記述できるようになる。また、再描画の方法や update メソッドの引数を変更する際も、アスペクト一箇所のみ修正しさえすれ良い。

これはアスペクト指向の利点である。なぜならオブジェクト指向では再描画メソッドの呼び出しが複数個所に分散することもあり、変更の際にはそれら全てを修正の対象にする必要が生じるからである。

しかし、アスペクトの存在を知らないプログラマが、図 2.2 のようなコードを書いた場合を考える。図 2.2 のプログラマにとっては、Point が移動した後の再描画の回数は一度きりである。しかし実際には、図 2.1 の

```
class Shape {
  private Point[] point;

  public void moveOnePoint(int i, int x, int y) {
    p[i].move(x, y);
    Draw.update();
  }
}
```

図 2.2: Shape クラス

```
class Point {
  private int x;
  private int y;

  public void move(int dx, int dy) {
    x += dx;
    y += dy;
  }
}
```

図 2.3: Point クラス

アスペクトからウィーブされたアドバイスも再描画を行うため、このプログラムでは続けて 2 回、再描画が行われることになる。しかしプログラムを見る限りは、再描画の回数は一度なので、このような現象がプログラマを混乱させる恐れがある。

このように、アスペクトはプログラムの挙動の予測を困難にする。AspectJ では、アドバイスをどこに織り込むかをアスペクト側が記述するため、プログラム側がその織り込まれたアドバイスの存在を知らない場合がある。どのアスペクトもプログラム中の様々な joinpoint を選択し、そこにアドバイスをウィーブすることができる。AspectJ にはアスペクトのウィーブに対する制限が存在しないため、このような問題が発生する。

2.3.2　再利用時の副作用

AOP のもう一つの問題は、モジュールを再利用する時に生じる。例えば、図 2.1 と図 2.3 のようなプログラムを含んでいるシステムを考える。

この Point クラスの move メソッドは、アスペクトにより拡張されている。move メソッドの呼び出しの直後にはアドバイスで再描画が行われることになっている。

このシステムから Point クラスを別のシステムに再利用する場合を考える。このとき、Point クラスを移動させたプログラマがアスペクトの存在を知らなかった時、移動後のシステムでは、DrawAspect が無い状態で Point クラスが置かれることになる。そして Point クラスの move メソッドは移動前とは異なる挙動を示す。「move メソッドの呼び出し直後に再描画をするという」処理が、移動後のシステムでは存在しなくなったために再描画が行われなくなるからである。

また、アスペクトを含むようなモジュールを再利用したい場合を考える。あるシステムに、別のシステムから再利用したいモジュールを移動させたとする。上の例でいうなら、図 2.1 と図 2.3 の両方を別のシステムに移行したとする。このアスペクトは Point クラスの挙動を満たすために用いられているものである。しかしポイントカット指定の記述で、プログラム中のまったく関係の無い箇所にあるジョインポイントが選択されることがある。ポイントカットにはワイルドカードによるジョインポイントの選択が可能なため、アドバイスがまったく異なるモジュールに影響を及ぼす危険がある。

ポイントカット記述と名前が一致してしまうと、モジュール外であってもウィーブが行われてしまう。これはウィーブに対する制限が無いために起こる。

2.4　求められる性質

以上のような問題は、アドバイスのウィーブに対する一切の制限が無いために起こりうるものである。これらの問題を回避するような AOP の性質としては、アスペクトの適用範囲に対して適度な制限がかけられているものが望ましい。

AOP に関する研究の中には、このような問題に取り組むものも幾つか存在する。制限のかけ方については、ウィーブに対する制限や、アドバイス内の処理に対する制限など様々である。

2.5　従来の技術

ここでは、上で述べた AOP の問題を解決するための研究をいくつか紹介する。以下の研究は、AOP の問題として指摘されたカプセル化破壊の

```
public class Figure {
  List elements;
  public Figure translate(int dx, int dy) {
    for(Iterator iter = elements.iterator(); iter.hasNext(); ) {
      Point elem = (Point)(iter.next());
      elem.translate(dx, dy);
    }
  return this;
  }
}
```

図 2.4: Figure クラス

原因の一つであった、アスペクトの無制限なウィーブに何らかの制約を設けるものである。

2.5.1　Open Modules

Aldrich の提唱する **Open Modules** [2][8][3] は上記のような、適用範囲が制限されていないアドバイスによって起きる問題を解決する。Open Modules は、簡単な関数型言語を定義して、アドバイスの適用範囲を制限することができる。

例えば図 2.4 のような簡単な Figure クラスを考える。このクラスは List で図上の点を表し、translate メソッドでそれら全ての位置を移動させる。

また、図 2.5 のようなアスペクトを考える。これは Figure.translate への呼び出し全てを横取りし、あとで再描画するためにその移動についての情報を保存するというものである。

このアドバイスは translate メソッドの呼び出しに強く関わっているので、Figure クラスの実装を変えて translate メソッドの呼び出し方を変更した場合に対処できなくなる。

例えば Figure クラスを図 2.6 のように変更したとする。これにより、elements の要素に Point オブジェクトだけでなく Figure オブジェクトも入れたい場合にも対応できるようになる。しかし、ReplayAspect のアドバイスは、Figure クラスの外から translate メソッドを呼び出す場合と、中から translate メソッドを呼び出す場合の両方に対してウィーブされるので、この修正によって再描画のためのリストに同一のものが保存されてしまう可能性がある。

そこで Open Modules は、図 2.7 のようなインターフェイスを提案して

```
Aspect ReplayAspect{
  pointcut translate(int dx, int dy):
    call(* Figure.translate(int, int)) && args(dx, dy);

  LinkedList moves = new LinkedList();

  before(int x, int y, Figure fig) :
    translate(x, y) && target(fig) {
      // Store fig, x and y in th moves list
  }
}
```

図 2.5: ReplayAspect アスペクト

いる。module の後ろに書いてある FigureModule はモジュール名を表し、class はそのモジュールに含まれるクラスを表す。なお、この例では、Figure クラスと Point クラスがその対象である。friend はアスペクト名をとり、ここで書かれたアスペクトはそのモジュールのメンバ内の全てのジョインポイントにアクセスすることが可能になる。expose はポイントカット記述で、外部に見せるジョインポイントを選択できる。モジュール内の、expose で指定されたポイントカットで選択されるジョインポイントには friend で指定されたアスペクト以外からも、任意のアドバイスのウィーブが許可される。advertise は class で指定されたクラスを除くクラスから呼び出された場合にアドバイスを適用するポイントカットを記述する。これは expose のシンタックスシュガーと考えることも出来る。実際、class 節に Figure と Point があるので、次の 2 つは同等である。

```
advertise : call(Figure Figure.translate(int, int));
expose : call(Figure Figure.translate(int, int)) &&
  !within(Figure) && !within(Point)
```

図 2.7 の場合では、friend で指定された DebuggingAspect は Figure クラスや Point クラス内の全てのジョインポイントへウィーブすることが可能になり、advertise によって ReplayAspect の再描画用のリストに、同一のデータが重複して入らないようにできる。また、expose によって、例えば図 2.8 のようなアスペクトを利用することも可能になる。

Open Modules は、モジュールという単位でクラスをまとめて、外部のアスペクトからのアクセスを制御している。これは公開していないジョイ

```
public class Figure {
  List elements;
  public Figure translate(int x, int y) {
    for(Iterator iter = elements.iterator(); iter.hasNext(); ) {
      Object elem = iter.next();
      if(elem instanceof Point)
        ((Point)elem).translate(x, y);
      else if(elem instanceof Figure)
        ((Figure)elem).translate(x, y);
    }
  return this;
  }
}
```

図 2.6: 変更した Figure クラス

```
module FigureModule {
  class Figure || Point;
  friend DebuggingAspect;
  advertise : call(Figure Figure.translate(int, int));
  expose : call(* Point.translate(int, int)) && within(Figure);
}
```

図 2.7: Figure モジュール

ンポイントを外部から選択できないようにしている。この技術は、外部のアスペクトからのウィーブを拒否することで、局所的なプログラムの挙動を保障する。

しかし、モジュールで指定していないクラスの挙動は保障されないため、新たなクラスを追加したときや既存のクラスの名前が変更したりした場合に、プログラマのミスが介入する恐れがある。また、Open Modules はアスペクトを含むようなモジュールの再利用については考慮していない。

2.5.2　ccJava

ccJava[11] では通常の Java に、アスペクトのウィーブに対応したインターフェイスの概念として「織り込みインターフェイス」と呼ばれるものを導入している。織り込みインターフェイスは公開した箇所にだけウィー

```
aspect TranslateCost {
  int total = 0;

  pointcut translate() : call(* Point.translate(int, int)) && within(Figure);
  before() : translate() { total++; }
}
```

図 2.8: Figure Translate Cost アスペクト

```
w_interface xShape {
  pointcut set() :
    execution(void set(int, int)) ||
    execution(void set(Point, Point));
 import after set();
}
```

図 2.9: xShape 織り込みインターフェイス

ブを限定するようなメカニズムになっている。織り込みインターフェイスを利用することで、クラス側のプログラマはアスペクトのウィーブを意識することなく実装することができる。織り込みインターフェイスによって公開されていない箇所にはウィーブされないようにすることで、ウィーブがどこに行われるのかを明確にし、同時に予期せぬウィーブを防止することも可能になる。

織り込みインターフェイスは通常の Java インターフェイスと異なり、ポイントカットやアドバイスといった関心事の合成に関わる情報を構成要素としてもつ。例として、図 2.9 のような織り込みインターフェイスを考える。織り込みインターフェイスであることを示すには [w_interface] という語を用いる。xShape は、set メソッド実行後に画面を再描画するという関心事を織り込ませる必要がある。したがって、この図では set メソッドの実行位置をポイントカットとして選択し、他の織り込みインターフェイスからの織り込みを可能にするためにインポートしている。

続いて画面クラスの織り込みインターフェイスを考える。図 2.10 のポイントカット記述は、update メソッドを選択している。そしてここここでは選択したメソッドを他のクラスに対して公開するためにエクスポートしている。

そして最後に、weave 節を記述する。この節では、どの織り込みインターフェイスをどのクラスが実装するのかを指定し、マッチングルールを

```
w_interface xDisplay {
  pointcut update() :
    execution(void update());
 export update();
}
```

図 2.10: xDisplay 織り込みインターフェイス

```
weave {
  class Point implements xShape;
  class Line implements xShape;
  class Display implements xDisplay;

  connect(xDisplay.update, xShape.set);
}
```

図 2.11: weave 節

記述する。

図 2.11 では、まずどのクラスにどの織り込みインターフェイスを実装させるかのリストを記述している。例えば、Point クラスは xShape 織り込みインターフェイスを実装させている。また、その後の connect 節は、xDisplay 内の update を、xShape 内の set へ織り込むようなジョインポイントを生成する。

ccJava では、織り込みたい処理の内容は、クラス側にメソッドとして記述され、織り込みインターフェイスにはポイントカット指定とアドバイスメソッドの呼び出し、そしてどれを公開するかという情報のみを記述する。このように、アスペクトとアドバイスの中身を分離することで横断的関心事をより汎用的にしている事も ccJava の利点である。

また、図 2.9 において、[execution(void set(int, int))] の set() が、AspectJ のように Point.set() となっていないのは、クラスが織り込みインターフェイスを実装するという機能のためである。Point クラスが xShape を実装することで、この「set メソッド」は「Point クラスの set メソッド」という扱いになる。

import 節はポイントカットと織り込みのタイミングを公開する。そして export 節はアドバイスの中身をクラスメソッドを呼び出すという形で記述する。connect 節がこれらを接続することで初めてウィーブが成される。ccJava では織り込みインターフェイスを継承したクラスは、織り込

みインターフェイス内に記述されているプログラムポイント（例では set メソッドや update メソッド）を必ず持っていなくてはならない。この考え方は Java インターフェイスを実装したクラスが、インターフェイス内にある全てのメソッドを実装しなくてはならない事と同様である。

Open Modules が「指定された箇所の挙動を保障する技術」であるのに対し、ccJava は「記述しなければ関心事を織り込まない技術」である。ccJava は、新たなアスペクトの追加や、クラスの変更・削除の際の手間が増える。特に、アスペクトを追加したい場合は、アドバイスの中身をメソッドとしてどこかに用意し、それをエクスポートした上で織り込みインターフェイスをそのメソッドを持つクラスに実装させなければならない。また、ポイントカット記述に一致するジョインポイントを持ち、且つそれを反映させたいクラス全部に織り込みインターフェイスを実装させる必要もある。ccJava はロギングのような単純な例でも AspectJ に比べると記述しにくいという欠点がある。

2.6　関連技術

AOP は OOP と比べて、不十分なカプセル化や可読性の低下が指摘されている言語である。以下では、そのような AOP の欠点を補うための様々な研究とそのアプローチを紹介する。

2.6.1　COW

COW（COntract Writing language）[12][10] は、制御フローと依存関係を用いてアドバイスの織り込み時の契約を記述する言語である。COW の契約は、織り込みによって生じるメソッドの振る舞いの変化をあらかじめ制限するものである。

COW における契約の考え方は、OOP の設計手法の一つである DbC（Design by Contract）に基づいている。DbC はプログラムを契約を用いて設計するという手法で、それは次の三種類の概念から構成される。

事前条件: メソッドが呼び出されるときに必要な条件

事後条件: 事前条件が満たされているときに、メソッド実行後に満たされるべき条件

不変条件: メソッド実行前後に関係なく満たされているべき条件

そして COW はこの概念を AOP におけるアドバイスのウィーブに対して適用させたものである。COW にも事前条件・事後条件・不変条件の概念があるが、DbC とは以下のような違いを持つ。

事前条件: 織り込み前のメソッドの振る舞い

事後条件: 織り込み後のメソッドの振る舞い

不変条件: 織り込み前後に関係ないメソッドの振る舞い

COW では、アドバイスのウィーブによって生じるメソッドの振る舞いへの影響をこれらのような条件を用いて表現する。ウィーブの影響を記述することで、そのウィーブがプログラマの予期しない作用を与えないように保障する。COW の契約は、アドバイスをウィーブされるクラスと、アドバイスを記述したアスペクトの間に結ばれる。

COW はアスペクトのウィーブ自体を操作するものではない。COW の契約は、ウィーブされたアドバイスがプログラマの意図に沿った処理を行うことを支援するものである。つまり、これはウィーブに制限を加えるのではなく、アドバイスの内容に制限を加えることでプログラムの挙動を保障する技術である。

2.6.2　XPI

K.Sullivan の提唱する **XPI(Crosscutting Programing Interface)**[6] は、アスペクトとクラスを含むようなシステムを設計する際にルールを設けることで予期しない織り込みを防ぐ技術である。

XPI で記述される設計ルールは全て自然言語で書かれている。このルールが守られているかどうかは、AspectJ の機構を用いることで検証可能ではあるが、XPI は設計ルールを提案するものであり、システムへの影響を制限するものではない。そのため、プログラマのミスによる予期せぬウィーブなどに対応できない。

2.6.3　Impact Analysis of Weaving

Impact Analysis of Weaving[9] はプログラムに対してそれぞれのアスペクトがどのような影響範囲を持っているかを調べるシステムである。

このシステムは、まずアスペクトを含むようなソースコードをコンパイルした際に、プログラムとアドバイスそれぞれのデータ依存グラフとコールグラフを生成する。そしてそれらのグラフを利用して、プログラムにウィーブが行われた場合のデータ依存グラフとコールグラフを生成する。

プログラムを実行させたときに、プログラマが予期せぬ挙動を発見したら、それらのグラフを用いることで、原因となったアスペクトを発見しやすくなる。この技術はアスペクトの影響を調べたり、バグを発見したり

することに利用できるが、ウィーブやアドバイスに制約を課すものではない。

2.6.4　AAIF

AAIF(Aspect Aware InterFace)[7] は、プログラムとアスペクトのウィーブに関する情報を、それぞれのモジュールのインターフェイスにまとめて記述する技術である。そのようなインターフェイスを見ることでプログラマは、どのメソッドにどのアドバイスがウィーブされているか理解できる。

AAIF はアスペクトの影響を一箇所に記述することでプログラムの Modular-Reasoning を向上させる手法である。AAIF にはウィーブに対する制限などは一切課されていない。

第3章　AOPのための独立性の高いパッケージシステム

すでに述べたように、AspectJ の仕様ではアスペクトの適用範囲に一切の制限が存在しないため、２章で述べたような問題が生じた。本稿では、このような問題を解決する手段として、アスペクトに適度な制限を設けたパッケージシステムを提案する。

適用範囲のルールは以下の二つである。

- パッケージ内のアスペクトがパッケージ外のプログラムに予期せぬ影響を与えない
- パッケージ外のジョインポイントで選択できるものは、公開されているものとパッケージ内のメソッドやフィールドへの call, set, get に限る

以下では、これらのルールについてより詳しく記述する。

3.1　本論におけるパッケージについて

Java には package という機構が存在する。本稿で説明するパッケージシステムは、Java の package の概念をそのまま利用している。パッケージ外からパッケージ内のクラスメンバへのアクセス制御が存在するように、本システムでもパッケージ内外へのアスペクトの織り込みに制限を設けている。

また、本システムではパッケージについて以下のように捉えている。

- パッケージは、再利用の単位である
- １つのパッケージは、１人または情報を共有したプログラマが作成している

１つ目の「再利用の単位」とは、「パッケージは各モジュール毎に対応して作成されているものである」という意味である。再利用できるプログラムは一般的には１つ以上のモジュールとして形成されているので、ここ

では「モジュールの再利用」と「パッケージの再利用」を同義で扱っている。また、Java のパッケージ機構はモジュールをまとめやすくするための設計であるので、このような解釈は不適切では無い。

２つ目の解釈は、パッケージとは何らかのモジュールであり、モジュールは普通、同一のプログラマが作成するものである、という意味である。プログラマにとって、自分の作成したモジュール内で行われた修正やアスペクトの織り込みなどは全て把握できる。本稿で用いる「予期せぬ織り込み」とはプログラマが携わっていない外部モジュールからのアスペクトの織り込みを意味している。

以上のことを基点として、アスペクトの適用範囲について考察した。

3.2　アスペクトの適用範囲が制限されたパッケージシステム

２章で述べたような問題を解決するには、アスペクトが限られた部分にのみ織り込みを許可されるようなシステムを構築するのが望ましい。なぜなら影響する範囲を狭めることで、局所的なプログラムにのみ作用させたいアスペクトがプログラム全体に散らばったりすることが無くなるからである。また、そのように構成されたシステムではプログラムの挙動の予測が容易になっている。

3.2.1　パッケージ制限

パッケージ内のアスペクトは、パッケージ外のプログラムを変更することができない。本システムでは、まずこのルールを基本としている。ポイントカット記述の条件に、パッケージ外のプログラムに属するジョインポイントが一致しても、そのジョインポイントにアドバイスが織り込まれることは無い。例えば、次のようなポイントカット記述を考える。

```
execution(int *.*(..));
```

このポイントカットが選択するジョインポイントは、プログラム中の任意の int を返す関数の実行時である。従来の AOP システムでは、このような記述を含むようなアスペクトはプログラム中のあらゆる場所に影響を与えてきた。しかし本システムでは、このようにワイルドカードを用いたとしても、パッケージ内のジョインポイントのみが選択され、パッケージ外のジョインポイントは選択されない。

```
package P1;

public class C1 {
  public void run() {
    P2.C2.run();
  }
}
```

図 3.1: P1.C1 クラス

```
package P2;

public class C2 {
  public void run() {
    /* ... */
  }
}
```

図 3.2: P2.C2 クラス

ポイントカット指定子の中でも、call, set, get は特殊なパターンである。これらは他のポイントカット指定子と違い、「呼び出し元」と「呼び出し先」の 2 つが存在する上でジョインポイントが現れる。

例えば、図 3.1 と図 3.2 のようなクラスを考える。そして、このプログラムに図 3.3 アスペクトを含める。

本システムでは、このアスペクトの適用範囲は『パッケージ [P2] の内部のみ』である。ポイントカット [pc] は『C2 クラスの run メソッドを呼ぶとき』を選択する。しかしこのとき選択されるジョインポイントは、P2 クラスではなく P1 クラスに含まれている。call, set, get で選択されるジョインポイントは全て呼び出す直前の位置（呼び出し元）に存在することになっている。このような場合、図 3.3 のアドバイスはウィーブされない。同じアドバイスをパッケージ P1 の内部に用意した場合は、アスペクトとジョインポイントが同じパッケージ P1 に属することになるのでアドバイスが織り込まれる。

その他の execution 等のポイントカット指定子については『呼び出し元』や『呼び出し先』といった概念が無い。ポイントカットは「○○を実行するとき」という指定になるので、その一点がアスペクトと同じパッケージに属しているかどうかを判定すれば良い。

```
package P2;

public aspect A1 {
  pointcut pc() : call(public void C2.run());

  before() : pc() {
    /* ... */
  }
}
```

図 3.3: P2.A1 アスペクト

```
package P2;

public abstract aspect A2 {
  pointcut pc() : execution(public void C2.run());
}
```

図 3.4: P2.A2 抽象アスペクト

アスペクトと同じパッケージに属していないジョインポイントには、そのアスペクトの影響が及ばない。しかし本システムには幾つかの例外ルールが存在する。以下はそれらについて述べる。

3.2.2　ポイントカット公開

上のルールでは、パッケージの内部に存在するジョインポイントに対してパッケージ外のアスペクトからウィーブすることは出来ない。しかしプログラムによっては、内部のジョインポイントを外部のアスペクトに公開し、ウィーブを可能にしたほうが良い場合もある。そのような場合を考え、本システムでは「抽象アスペクト」と「名前付きポイントカット」を用いることでパッケージ内のジョインポイントを外部に公開できるようにした。

例えば、図 3.4 のような抽象アスペクトと、それを継承した、図 3.5 のようなアスペクトを考える。ポイントカット pc は親アスペクト A2 で宣言されているが、そのポイントカットを利用したアドバイスは子アスペクト A3 で実装されている。このルールでは、A3 はパッケージ P3 に属するが、継承したポイントカット pc をそのまま使用する場合に限り、そのポ

```
package P3;

public aspect A3 extends P2.A2 {
  before() : pc() {
    /* ... */
  }
}
```

図 3.5: P3.A3 アスペクト

イントカットを宣言したアスペクト A2 が属しているパッケージ P2 の内部にあると見なされる。

アスペクトが抽象アスペクトを継承するときは、名前付きポイントカットのみが引き継がれ、アドバイスは引き継がれない。（メソッドやフィールドは Java のクラスの継承と同様に引き継ぐ。）ポイントカットはオーバーライド可能である。しかし本システムではそれをした場合には、そのポイントカットは親アスペクトではなく子アスペクトに宣言されたものとして見なす。

子アスペクトが干渉できるジョインポイントは、自パッケージ内のジョインポイントと、親アスペクトがポイントカット記述で選択したジョインポイントである。無論、親アスペクトが干渉できないようなジョインポイントを選択していたなら、子アスペクトも継承したポイントカットを利用してそこに影響を与えることは出来ない。

Open Modules は、モジュールの外部に対してはジョインポイントを公開できる。対して本システムではポイントカットを公開可能にしている。Open Modules では一度公開したジョインポイントに対するウィーブに制限がかからず、予期せずポイントカット記述が偶然一致した場合などにその織り込みを防ぐことができない。本システムでは、公開されたポイントカットが存在していても、その記述を含んだアスペクトを継承し、引き継いだ名前付きポイントカットを利用して初めてアドバイスが織り込まれるため、プログラマのミスが介入しにくい。

3.2.3　インターフェイスと見なせるアドバイス

すでに述べたように、call, get, set のようなポイントカット指定子は特殊である。これらには『呼び出し元』と『呼び出し先』が存在し、ジョインポイントとして選択されるのは呼び出し元のほうである。異なるパッケージをまたがるようなメンバアクセスを想定すると、呼び出される側の

```
package P2;

public aspect A4 {
  pointcut pc() : call(public void C2.run()) && !within(P2);

  before() : pc() {
    /* ... */
  }
}
```

図 3.6: P2.A4 アスペクト

```
package P3;

public class C3 {
  public void f() {
    P2.C2 cl = new P2.C2();
    cl.run();
  }
}
```

図 3.7: P3.C3 クラス

パッケージに属するアスペクトは呼び出し元とは異なるパッケージに属しているため、本来ならばそのようなアスペクトは干渉されない。しかし本システムでは、公開されたポイントカットを用いずにパッケージ外のジョインポイントに干渉できるルールが存在する。

例えば、図 3.6 のようなアスペクトを考える。このアスペクトに記述されたポイントカットの [&&] 以降には、『パッケージ P2 の外部にあるジョインポイントを対象とする』という条件が付け加えられている。本論で述べてきたルールでは、属するパッケージが異なるジョインポイントには干渉できないため、パッケージ P2 の内部にあるアスペクトがこのようなポイントカット記述をしてもウィーブはされないように思える。しかしここでは、このようなポイントカットを特殊なケースとして扱っている。ポイントカットに「自パッケージに属するメンバへの call, set, get」と、**!within(自パッケージ名)** の両方を含めるように記述した場合のみ、そのポイントカットはパッケージ外部のジョインポイントも選択できる。例では、図 3.6 のアスペクトは図 3.7 のクラス内部にあるジョインポイント (cl.run() の直前) にアドバイスをウィーブできる。

```
package P1;

public class C4 {
  public void f() {
    /* ... */
  }
  public void g() {
    /* ... */
  }
}
```

図 3.8: P1.C4 クラス

```
package P2;

public class C5 extends P1.C4 {
  public void f() {
    /* ... */
  }
}
```

図 3.9: P2.C5 クラス

このようなウィーブは、外部のどのパッケージから自パッケージ内のメンバにアクセスしても全て同一の処理をするようなプログラムなら、それは自パッケージのインターフェイスと見なせるため許可されるようにした。したがって、特定のパッケージだけに影響を与えるようなアドバイスは、呼ばれる側のパッケージ内のアスペクトには記述できない。そのようなアスペクトは全て、呼び出す側のパッケージ内に記述しなければならない。

3.2.4　インタータイプ宣言

AspectJ には、あるクラスの親クラスを変更したり、メンバを追加したりするインタータイプ宣言がある。インタータイプ宣言もまた、プログラムに予期せぬ挙動を与えることがある。

例えば、図 3.8 のクラスと、それを継承した図 3.9 と図 3.10 の 2 つのクラスを考える。プログラム上で、クラス C5 のメソッド g を呼んだとき、

```
package P3;

public class C6 extends P1.C4 {
  public void g() {
    /* ... */
  }
}
```

図 3.10: P3.C6 クラス

```
package P4;

public aspect A5 {
  declare parents : P2.C5 extends P3.C6; // (1)

  public void P3.C6.f() { // (2)
    /* ... */
  }
}
```

図 3.11: P4.A5 アスペクト

親であるクラス C4 の g が代わりに呼ばれる。同様に、クラス C6 のメソッド f を呼んだときは、クラス C4 の f が呼ばれる。

このようなコードを含んだプログラムが、図 3.11 のようなアスペクトから影響を受けたとする。アスペクト A5 内の最初の命令 (1) は、クラス C5 の親を C6 に変更するインタータイプ宣言である。このとき、クラス C6 はクラス C5 の直接の親であるクラス C4 を継承しており、親クラスを変更する際の条件を満たしているので、この命令は問題無く実行される。しかしこのようなアスペクトが織り込まれると、クラス C5 のメソッド g を呼んだときにクラス C4 ではなく、クラス C6 のメソッド g が代わりに実行されてしまう。

同様にアスペクト A5 の次の命令 (2) は、クラス C6 にメソッド f を追加するインタータイプ宣言である。クラス C6 は、親であるクラス C4 からメソッド f を継承しているが、C6 内ではメソッド f を宣言（オーバーライド）していない。クラスメンバを追加するアスペクトは、オーバーライドをする場合に限り、名前が被ることをコンパイラが許可している。このときアスペクト A6 により、クラス C6 のメソッド f は親クラスのメソッド f ではなく、アスペクト側に記述された命令を実行するように変更されて

しまう。本研究では、以上のような状況を考慮し、インタータイプ宣言にもパッケージ制限を加えることが望ましいと判断した。

インタータイプ宣言の対象となるクラスは、アスペクトと同一パッケージ内にあるものでなければならない。親クラス変更の命令の場合は、子クラスとして指定される側がアスペクトと同一パッケージに属していることが条件である。新たな親クラスとして指定されるクラスに制限は存在しない。親クラスにとっては、子クラスが新たに追加されたせいで挙動が変化するようなことは起こりにくいからである。

3.3　実装

本研究では、上で述べたようなルールを **abc(AspectBench Compiler)**[4] を改造することで実装した。abc は AspectJ 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用いて作成されている。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。また、abc は Open Modules の仕様もサポートしている。

abc がアスペクトを含むような Java プログラムをコンパイルしたとき、その動作は次のようになる。

最初は『Java のコンパイルとアスペクトの解析』を行う。プログラム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、そのプログラム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、コンパイラはポイントカットやアドバイスの情報を取得する。

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、そこにウィーブされるアドバイスリストに追加し、ウィーブ自体はすぐには行わない。これはアスペクトの優先順位や、実行時に動的に条件をチェックするアスペクトが存在するからだと思われる。ポイントカットから一致するジョインポイントを検索するのではなく、ジョインポイントから一致するポイントカットを検索する。ウィーブは、上と同様に、全てのクラスを調べながら、そこにあるアドバイスリストからアドバイスをウィーブしていく。

3.3.1　パッケージ制限

このルールの制限は、パッケージ名が一致するかどうかが基準となっている。abc 内部では、アドバイスは自身を記述しているアスペクトの情報を持っていて、アスペクトは自身を含むパッケージの情報を持っている。

ここでは、ウィーブを行う直前にアドバイスリストを全てチェックすることで制限を加えている。『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、一致すればリストに残したままにする。call, get, set ポイントカットの場合は、ジョインポイントは呼び出し元にあると見なされるので、保持する情報は多少異なるが、アスペクトのパッケージ名とジョインポイントのパッケージ名を比較する部分は同様である。

3.3.2　ポイントカットの継承

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をリストから削除してはならない。

アスペクトが別パッケージの抽象アスペクトを継承して、名前付きポイントカットをオーバーライドすることなく利用するとパッケージが異なっていてもウィーブが行われる。アドバイスは、自身を実装しているアスペクトと、ポイントカット記述の情報を持っている。ポイントカットは、それが名前付きポイントカットであった場合は、自身が宣言されたアスペクトの情報を持っている。つまり、ここではアドバイス側のアスペクトと、ポイントカット宣言のアスペクトが異なるかどうかをチェックする。また、アスペクトにも親アスペクトが存在するかどうかをチェックし、親アスペクトとポイントカット宣言のアスペクトが一致していたなら、アドバイスは一時的に親アスペクト内に記述されているように振舞う。

親アスペクトに宣言された名前付きポイントカットを子アスペクトがオーバーライドした場合、そのポイントカットは宣言が子アスペクトにあるものと見なされるので、通常のルールが適用される。これは通常のアスペクトがポイントカットを宣言して利用する場合と同様に処理できる。

名前付きポイントカットがオーバーライドされた場合に注意しなければならないのが、親アスペクト内に記述されているアドバイスである。ポイントカットのオーバーライドは Java メソッドのオーバーライドと異なり、親側の挙動も変更してしまう。というのは、継承させたいアスペクトは abstract 指定子を付けた抽象アスペクトでなければならず、抽象アスペクトは実体が無いため、親アスペクトのアドバイスは実際は子アスペクトのアドバイスとなっているからである。抽象アスペクトは、それを継承するような非抽象アスペクトが存在しなければ、どんなアドバイスを記述していても何も起こらない。子アスペクトが存在して初めて親のアドバイスも実体化するため、ポイントカットのオーバーライドの影響を受けてしまうのである。AspectJ では、Java と同様に、オーバーライドを禁止する [final] 指定子をポイントカットに付けられる。ポイントカット記述がオー

バーライドされた場合、その挙動予測が困難になるため、本ルールではそのような親アドバイスは全てウィーブされないようにしている。

3.3.3　!within(自パッケージ名)

もう一つの例外ルールは、インターフェイスと見なせるアドバイスである。自パッケージ内のクラスメンバへの call, get, set をポイントカットで指定するとき、選択されるジョインポイントは呼び出し元に含まれているため、それが他のパッケージに含まれていた場合はアスペクトとジョインポイントが異なるパッケージに属していることになる。本システムでは、外部のどのパッケージから呼び出されても必ずアスペクトが織り込まれるという条件の下ならば、アスペクトとジョインポイントの所属パッケージが異なっていてもウィーブをすることは可能である。

このような条件は、call, set, get ポイントカットに [&& !within(自パッケージ名)] を付けているかどうかで判定する。自パッケージ内にしか影響を及ぼさないルールの下で、「自パッケージの外にあるジョインポイント」という条件を付け足すことにより、それがインターフェイス用のポイントカットであることを明示する。

ここで、[!within()] の中に自パッケージ以外の特定のパッケージを選択しても、ウィーブを許可されるのは、自パッケージ内のジョインポイントのみである。また、パッケージでなくクラスなどを記述したとしても、パッケージ内でその条件を満たすジョインポイントにウィーブが可能になるだけである。

[within(特定のパッケージ)] という条件を付けた場合は、例外ルールは適用されない。自パッケージの中で、且つ指定された特定のパッケージの中にあるジョインポイントが選択される。ここでの例外ルールに関しては、全てのポイントカット記述には常に [&& within(自パッケージ名)] という条件が暗黙の内に追加されていると考えるのが理解しやすいだろう。そして [&& !within(自パッケージ名)] という条件を明示した場合のみ、本来ならばジョインポイントは一切選択されないが、例外ルールとして外部パッケージにのみ影響を与えることのできるポイントカット記述と見なされる。

3.3.4　インタータイプ宣言

abc では、インタータイプ宣言の処理は、アドバイスのウィーブよりも前に行われる。本システムでは、異なるパッケージに属するクラスに影響を与えるようなインタータイプ宣言は実行されないようにしている。

メソッドをオーバーライドさせるようなインタータイプ宣言が無視されると、オーバーライドの記述が無い状態でコンパイルが進む。そのメソッドが抽象メソッドであったりした場合は、通常の Java と同様にコンパイルエラーを起こす。

無視されたインタータイプ宣言は、何も書かれていないのと同様に扱われるので、プログラム側に問題が無ければ、コンパイルは正常に終了する。つまり、新たに追加されたはずのメソッドがパッケージ制限により追加されずにコンパイルが進んだとしても、そのメソッドを呼び出すような命令がプログラム中に存在しなければエラーにはならない。

親クラスを変更するインタータイプ宣言も同様である。変更の命令を無視された場合でも、継承命令自体が記述されていないものとしてコンパイルが進むだけで、プログラム側に問題がなければエラーにはならない。エラーになるかどうかの条件は、通常の Java の設計と同様である。

3.4　再利用性の向上

本システムでは、パッケージをモジュールの単位として、そのカプセル化の方法を提案した。このルールを用いてアスペクトを制限した場合、従来の AOP に比べて局所的なプログラムの挙動を予測しやすいという利点がある。

３章ではカプセル化を推進するようなルールを実装を交えて説明し、従来の AOP のもう一つの問題点として指摘した再利用性については触れなかった。しかし本システムのルールは、その再利用性の低さも補っている。

まず第一に、アスペクトを含むようなプログラムの中から一部のモジュールを取り出して再利用したい場合を考える。従来の AOP では、プログラムはシステム内の関係の無い場所にあるアスペクトから影響を受けることがあった。再利用したいプログラムが、そのようなアスペクトからの影響を受けていたなら、移行した先のシステムにおいて、移行前と同じ挙動をするとは限らないだろう。しかし本システムではルール上、影響を与えるようなアスペクトは同一パッケージ内に存在しなければならない。つまり、そのようなモジュールを再利用する場合には必ずアスペクトが付随しているため、別システムに移行した後にモジュールの挙動が変化することが無い。

第二に、アスペクトを含むようなモジュールを他のシステムに移行した際に起こりうる不具合を考える。従来の AOP では、そのアスペクトが移行先のシステム内で関係の無い場所にあるプログラムに影響を与えてしまう可能性がある。本ルールでは、そのようなパッケージ外部への影響が許可されていないため、システム内の別の部分の挙動が変化することは

無い。

本研究で提案するルールは、パッケージ単位のカプセル化の破壊を防ぐものであった。しかし同時に、このようなパッケージ制限は、アスペクトを含むようなモジュールの再利用性も向上させている。

第4章　実験

本章では、サンプルプログラムを用いて AspectJ と本システムの性能を比較する。実験の対象には **AJHotDraw**[1] のソースコードを用いた。AJHotDraw はオープンソースのアプリケーションで、AspectJ の文法に沿ったアスペクト指向言語で作成されている。計測の内容は以下の通りである。

4.1　アスペクトの情報の比較

まず、AJHotDraw のソースファイルを一切修正していない状態のアスペクトの情報を計測する。情報とは、アスペクトのファイルがいくつのパッケージに含まれているか、ファイル数やアドバイスの数はいくつか、などを指す。これを、既存の AspectJ でコンパイルする場合の情報とする。なお、本研究では AspectJ を拡張した abc を 3 章で述べた実装の対象としているので、実験においても abc を用いてコンパイルを行う。

もう一つの計測内容として、本論で提案したパッケージ制限が掛けられている場合の AJHotDraw のアスペクト情報を調べる。パッケージ制限が存在する場合、修正されていない AJHotDraw はパッケージ制限が存在しない時とは異なる挙動を示す。ここでは、パッケージ制限が掛けられた上で、初期状態の AJHotDraw とまったく同じ挙動をするようにアスペクトを修正した。そしてその修正した AJHotDraw を、パッケージ制限付きの abc でコンパイルする場合のアスペクトの情報を計測した。

4.1.1　リファクタリングについて

AJHotDraw が修正前と修正後で同じ挙動であることは、以下のようにして確かめた。

まず、修正の対象がアスペクトだけであることから、クラス側の挙動は変化しないと言える。アスペクト側の挙動が変化しないことは、アドバイスが修正前とまったく同じジョインポイントにのみ影響していて、且つ修正後のインタータイプ宣言が全て適用されれば良い。なぜなら、アドバイスは内容が変わらなければソースコードのどの部分に記述されていて

表 4.1: AJHotDraw のアスペクトの比較

	abc(1)	パッケージ制限 (2)
パッケージ数	3	7
アスペクトのファイル数	10	17
アドバイスの数	5	5
親クラス変更アスペクトの数	10	10
クラス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
ウィーブの合計数	44	44

もウィーブされるジョインポイントは変わらず、インタータイプ宣言もまた、それが定義されたソースコードが別の部分に移動しても影響は変わらないからである。パッケージ制限を課した場合にそれらの数が減少することはあっても増加することは無い。

abc では、全てのジョインポイントに対してそこにウィーブされるアドバイスの情報をリストとして持っているため、それらの合計数を知ることは簡単である。AJHotDraw ではウィーブは全部で 44 回である。修正後の AJHotDraw でも 44 回であることを確認し、また、本システムではインタータイプ宣言がパッケージ制限で除去された場合には通知するようにしているため、これらの挙動は修正前後で変化していないと判断できた。

4.1.2　結果

修正前後でのアスペクト情報を比較した結果は、表 4.1 のようになった。

4.1.3　考察

以下、abc でコンパイルする場合の AJHotDraw の (修正前の) ソースを (1)、パッケージ制限付きの abc でコンパイルする場合の (修正後の) ソースを (2) と呼ぶ。

両者の差は、アスペクトが属するパッケージの数とアスペクトのファイル数が (1) から (2) にかけてそれぞれ増加した点である。(1) 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ファイルのみを含んでいたので、パッケージ制限が掛けられたコンパイルではアドバイスもインタータイプ宣言も一切許可されない。それらのアスペクトをプログラムに反映させるためにファイルを移動させた結果、(2) のようにそれらの数が増加した。

表 4.2: AJHotDraw のコンパイル速度の比較

	abc	パッケージ制限付き abc
修正前のソース	42.843 秒	コンパイルエラー
修正後のソース	42.375 秒	42.500 秒

しかし (2) では、全てのアスペクトは必ず、そのアスペクトが影響を与えるプログラムと同じパッケージに属している。これは (1) と異なり、パッケージ単位でのプログラムの再利用性に優れている。(1) では、アスペクトを含まない部分のパッケージの再利用時には、そこに影響を与えるアスペクトを発見し、同時に持っていく必要があった。あるアスペクトの一部分のみが影響を与えている場合などに、その部分だけを抽出するのは特に手間がかかる。(2) では影響を与える部分毎にアスペクトをすでに分割しているため、そのような煩雑さを解消している。

4.2　コンパイル速度の比較

abc と、パッケージ制限を設けるように改造した abc で、AJHotDraw をそれぞれコンパイルした場合の終了までの時間を計測する。

4.2.1　実験環境

実験環境には以下のものを使用した。

CPU : AMD Athlon(tm) 64 2.20GHz

メモリ : 2GB

OS : Windows XP SP2

JavaSDK : 1.5.0

abc : 1.2.1

4.2.2　結果

実験結果は以下のようになった。

4.2.3　考察

図 4.2 が示す通り、両者のコンパイル時間には極端な差は見られなかった。(0.3%のオーバーヘッド) したがって、パッケージ制限を加えたコンパイラであっても十分実用的な速度で動作することが確認された。

第5章　議論

実験では、本システムのパッケージ制限に AspectJ プロジェクトを対応させた。AJHotDraw はアスペクト指向で設計されたアプリケーションであったが、本システムのルール下では、パッケージ外へ影響を与えるようなアスペクトがウィーブを禁止されるため、そのままでは正常に動作しなかった。そのため、パッケージ制限を課せられた状態でも本来の動作を行うようにソースコード中のアスペクトを移動、分割する必要が生じた。

４章ではこの AJHotDraw の修正におけるアスペクトの移動、修正に伴う副作用として、アスペクトのファイル数と、アスペクトを含むパッケージ数が増加することを確認した。

5.1　パッケージ制限を設けることによる副作用

ここでは、４章の実験では確認することの出来なかった、パッケージ制限を加えることによる様々な副作用を考察する。パッケージ制限が課されたことで、１つのアスペクトが複数のパッケージに同時に影響することはなくなった。そのため、同じアスペクトを複数のパッケージに作用させたい場合にその数だけファイルを増やさなくてはならないという欠点が考えられる。以下では、修正の前後でアスペクトの情報の個数を増やすような場合について考察している。

5.1.1　アドバイス数の増加

AJHotDraw の修正時には、その前後でアドバイスの数は増加することはなかった。これは、ジョインポイントを選択する際に、異なるパッケージに属しているジョインポイントを同時に選び出しているものが無かったからである。

複数のジョインポイントを選ぶようなポイントカット記述というのは、そのパターンが限定されている。ひとつは、クラスやメソッドの名前の記述でワイルドカードを使用した場合である。条件に該当するような名前が複数あれば、その数だけ選択されるジョインポイントも多くなる。もうひとつは、call, set, get ポイントカットを用いた場合である。これらのポ

イントカットは、指定されたメソッドの呼び出し、またはフィールドの参照や上書きを実行しようとするようなプログラム全てに作用するものである。execution などのポイントカット指定子では、『実行するとき』が選択されるため、１つのポイントカットに対して、ジョインポイントは１つである。しかし、call などのポイントカット指定子では、選択されるジョインポイントは『呼び出し元』に属するため、そのメソッドの呼び出しが複数箇所に存在すればその数だけ選択されるジョインポイントも増えてしまう。

本システムでは、インターフェイスと見なせるアドバイスは、呼び出し先のパッケージに属していても呼び出し元のパッケージのジョインポイントに作用することができる。しかし、これは外部の特定のパッケージからの呼び出しだけを選択することができるわけではない。本システムでは、『パッケージ外からの全ての呼び出しに対して一様に作用するアドバイス』をインターフェイスと見なしている。特定のパッケージからの呼び出しだけに作用させたいアドバイスは、呼び出し元のパッケージ内に含めなければならない。

ここでもっとも副作用が大きくなるのは、例えば、名前が pA, pB, pC, pD, … という複数のパッケージが存在するようなシステムで、以下のようなアドバイスが記述されているときである。

```
before() : call(void pA.class1.run()) && !within(pB) { ... }
```

これは、パッケージ pA 内のメソッド run を、パッケージ pB 以外の場所から呼び出す直前に作用するアドバイスである。このアドバイスを、本稿のパッケージ制限付きのシステム内で実現させるのは困難である。

なぜなら、このアドバイスは本システムが提案する『インターフェイスと見なせるアドバイス』の定義に当てはまらないため、パッケージ pA 内にそのように記述できないからである。「pB 以外のパッケージ」という条件が特定のパッケージを指しているため、このアドバイスは全ての呼び出しに対して一様に処理されなくなっている。そのため、呼び出し元になる可能性を持つ全てのパッケージにこのアドバイスを記述していかなければならなくなる。この例では、パッケージ pA, pC, pD, … がアスペクト追加の対象であり、もしアスペクトを含んでいないようなパッケージが存在した場合は、アドバイスの追加に伴い、ファイル数が増加してしまう。

AJHotDraw では、call ポイントカット記述の within 節で、特定の一つのクラスしか指定していなかったため、アドバイスの数が増えることは無かった。また、このアプリケーションではワイルドカードを用いた名前の指定も使用されていなかった。

5.1.2　インタータイプ宣言の増加

AJHotDrawでは、インタータイプ宣言の数も修正の前後で変化しなかった。インタータイプ宣言が増加するようなパターンは、アドバイスの場合よりも少なく、修正前の名前指定でワイルドカードを使用していたときのみである。AJHotDraw のインタータイプ宣言で指定されたクラスの名前にはワイルドカードは一切使用されていなかった。

5.1.3　名前付きポイントカットの増加

名前付きポイントカットが増加するようなパターンは、アドバイスが増加する場合と同じである。AJHotDraw では、アドバイスのポイントカット指定に、ポイントカット記述をそのまま書いているものと、定義した名前付きポイントカットを書いているものと両方が存在した。しかしいずれも、修正後にその数が増えるような条件のポイントカットでは無かったため、これらの数は変化しなかった。

同じ名前付きポイントカットが 2 つ以上のアドバイスで使用されているものも存在したが、それらのアドバイスは同一パッケージ内にのみ影響するものであったため、名前付きポイントカットの総数が増加することは無かった。

5.1.4　ウィーブ回数の増加

パッケージ制限を加えた場合に、ウィーブの回数が増加することは無い。ソースコードの修正に伴ってアドバイスやアスペクトの数が増減しても、元のシステムを同じ挙動をするのならウィーブ回数は変化しない。

制限によりアドバイスがウィーブを禁止される場合があるので、修正後にウィーブ回数が減少することは十分に考えられる。しかしこれは本稿の指摘する、予期しないウィーブを防ぐための機構であるからで、ウィーブ回数の減少は本システムの欠点を示すものではない。

5.2　今後の課題

本研究では、パッケージ単位でアスペクトを制御することでカプセル化と再利用性を高められることを述べた。しかし AJHotDraw は本システムの有用性を示すケーススタディとしては十分ではなかったと考えられるため、様々なパターンのアスペクトに対してパッケージ制限を課した場合の変化を計測する必要がある。

　また、パッケージ制限の機構は abc を改造することで実現しているが、そのルールは細部まで完成していない。例えば、パッケージをまたぐようなクラスの継承関係が存在した場合に、親クラスが属するパッケージのインターフェイスと見なせるアドバイスは子クラスのメソッド呼び出し時に影響するかどうか、などの議論の余地が残されている。

第6章　まとめ

本稿は、アスペクト指向プログラミングで書かれたプログラムにおいて、アスペクトに適度な制限を設けたパッケージ機構を提案した。従来のアスペクト指向プログラミングでは、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、本研究が提案するシステムでは、アスペクトはプログラム中の限られた部分にのみアクセスを許可されている。これにより、プログラムの一部を想定して書かれたアスペクトがプログラム全体に影響を与えることがなくなり、プログラムの挙動の予測が容易になる。

本研究が提案するシステムにおいてアスペクトの織り込みに課せられる制約とは、アスペクトが属しているパッケージと、ジョインポイントが属しているパッケージが一致しない限りウィーブは行われないというものである。この制約によって、関係の無いパッケージへのアスペクトの予期せぬ影響を回避できる。また、例外として外部に対して公開されているジョインポイントになら、パッケージ外からのアスペクトでもウィーブを行うことができる。自パッケージ内のクラスメンバが任意の外部パッケージからアクセスされるとき、そのプログラムに自パッケージ内のアスペクトを作用させることも可能である。なぜならそのようなアスペクトは、自パッケージのインターフェイスと見なせるからである。これらの例外を適用させる場合、プログラマは明示的にコードを記述しなければならず、そのためこれらのアスペクトがプログラムに予期せぬ影響を与えたりすることは無い。

本研究で提案するシステムは、AspectJ を拡張したコンパイラである abc(AspectBench Compiler) を改造することで実装した。これを用い、アスペクト指向で設計されたアプリケーションをパッケージ制限が付加されたルールに対応するように修正し、修正前後でのアスペクトの情報を比較した結果、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その部分に影響を及ぼすアスペクトを付加させる手間が無くなることを確認した。

本システムの制限は、モジュールのカプセル化を向上させただけでなく、パッケージの再利用性も高めた。アスペクトの影響に制限を課す技術として、Open Modules や ccJava などが挙げられる。これらはアスペク

トに何らかの制限を加えることで、プログラムの挙動を保障する技術である。しかし、これらはアスペクトを含むようなモジュールの再利用性までは考慮していない。本研究の今後の課題としては、様々なアスペクトにパッケージ制限を課した際の副作用の検証と、パッケージをまたぐようなクラス継承に対する制限の方法の考察などがある。

参考文献

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

[2] Aldrich, J.: Open Modules: Reconciling Extensibility and Information Hiding, *In AOSD workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT '04)* (2004).

[3] Aldrich, J.: Open Modules: Modular Reasoning about Advice, *In Proceedings of the European Conference on Object-Oriented Programming, volume 3586 of LNCS*, pp. 144–168 (2005).

[4] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J.: *abc*: The AspectBench Compiler for AspectJ, *Generative Programming and Componenent Engineering (GPCE)* (Glück, R. and Lowry, M.(eds.)), Lecture Notes in Computer Science, Springer (2005).

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

[6] Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y. and Rajan, H.: Modular Software Design with Crosscutting Interfaces, *IEEE Softw.*, Vol. 23, No. 1, pp. 51–60 (2006).

[7] Kiczales, G. and Mezini, M.: Aspect-oriented programming and modular reasoning, *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 49–58 (2005).

[8] Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O. and Sittampalam, G.: Adding open modules to AspectJ, *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, New York, NY, USA, ACM Press, pp. 39–50 (2006).

[9] Shinomi, H. and Tamai, T.: Impact Analysis of Weaving in Aspect-Oriented Programming, *ICSM '05: Proceedings of the 21st IEEE*

International Conference on Software Maintenance (ICSM'05), Washington, DC, USA, IEEE Computer Society, pp. 657–660 (2005).

[10] Shinotsuka S, Ubayashi N, S. H. and T, T.: An Extensible Contract Verifier for AspectJ, *2nd Asian Workshop on Aspect-Oriented Software Development (AOAsia-2, Workshop at ASE 2006)*, pp. 35–40 (2006).

[11] 境顕宏, 鵜林尚靖, 玉井哲雄: AOP 言語への織り込みインターフェイスの導入, 日本ソフトウェア科学会第 23 回大会 (2006).

[12] 篠塚卓, 鵜林尚靖, 四野見秀明, 玉井哲雄: 契約によるクラスとアスペクト間の影響解析, 日本ソフトウェア科学会第 22 回大会 (2005).