4F-6

横断的関心事の特定に基づくアスペクトモジュールの記述支援

斉藤 瞳† 榊原 正天† 桜井 孝平‡ 山崎 雄大‡ 古宮 誠一‡
芝浦工業大学工学部情報工学科† 芝浦工業大学大学院電気電子情報専攻‡

1. はじめに

アスペクト指向プログラミング(AOP)[1]は、プログラム中に存在する横断的関心事を一つにまとめて、アスペクトというモジュール単位で扱うことにより、横断的な関心事の分離を実現するプログラミング手法である。

AspectJ は Java を拡張することによってアスペクト指向プログラミングを実現した処理系の一つである。

AspectJ では、横断的関心事として実行したい処理の内容をアドバイスにまとめ、アドバイスを挿入する位置を特定するための位置としてポイントカットを定義し、アスペクト内に記述する。

AspectJ で、AOP を実践するにはまず開発者がプログラム中の横断的関心事を識別する必要がある。しかし、この横断的関心事は定義が曖昧であるために識別が困難である。

たとい横断的関心事が識別できたとしても、アスペクトをコードとして実装すること、数あるポイントカットを使用して、アドバイスを挿入したい位置だけを指定することは難しい。

そこで本研究では、段階的にアスペクトを追加記述するための支援を提供する。具体的には、1) 段階的に追加される関心事を横断的関心事と仮定する観点[2]を提供し、2) 関心事が特定されたコードの記述からアスペクトモジュールを自動で生成することを目標とする。

2. 誤りのあるアスペクト実装の問題

段階的に機能追加されていくプログラムの例として、点や線といった図形の移動が行われるたびに、図形の再描画を行う図形表示プログラムを考え、横断的関心事とアスペクトの実装の問題について述べる。

段階的開発手法を想定すれば、図形表示プログラムは、以下のような段階が考えられる。

1. 初期の仕様として 図形を表わす Point や Line,及び Display などのモデル化されたクラスを定義。
2. ディスプレイ更新: 図形を再描画するための追加的な機能。
3. 図形追加: Rectangle などの図形要素の追加定義。

これらの段階を横断的関心事としてとらえる。本稿では 1,2 についてのみ扱う。

1 のモデルクラスの段階では図 1 のような図形が移動する機能を持っている。2 のディスプレイ更新の段階では図 2 のアスペクトのアドバイスを図 1 のプログラムに挿入して実行する。DisplayUpdate は、call ポイントカットによって各クラス内の move が呼び出された後に Display.update();を実行するアスペクトである。

このように段階的開発手法においてアスペクトを利用することで、既存のモジュール(Point や Display)を修正することなく横断的に関心事を追加定義できる。

しかし、うまく実装できるとしても、実際にはコードを理解し、複数クラスに存在すると考えられる横断的関心事を把握してアスペクトを実装するのは困難である。結果として、誤ったアスペクトモジュールの実装になる場合がある。

例えば図 2 のアスペクトを適用すると、Line の move を呼び出すと Point の move が p1 と p2 でそれぞれ 1 回ずつ呼ばれるために、Point の move 呼出し後に行われる合計 2 回の Display.update();が呼び出される。この多重更新は望ましい振舞とは言えない。解決のためには図 4 のような定義とする必要がある。図 4 では 2 行目にあるポイントカット式に、&& !cflow(call(void move(int, int)))という条件が追加されている。

これにより、Line の move 内で呼び出される Point の move の呼び出し時に Display.update();が呼ばれなくなるために、多重更新を避けることが出来る。

このように宣言的で抽象的な記述を行うことで、結果の振舞を理解するのが困難な場合があると言える。

Making Code Description of an Aspect Module Easy through Specifying Crosscutting Concerns
Hitomi Saito†, Kohei Sakurai‡, Yudai Yamazaki‡, Masanori Sakakibara†,Seiichi Komiya‡
†Shibaura Institute of Technology
‡Graduate School of Engineerimg, Shibaura Institute of Technology

```
1| interface Figure {
2|   void move(int dx, int dy);
3| }
4|
5| class Point implements Figure {
6|   void move(int dx,int dy){x+=dx; y+=dy;}
7| }
8|
9| class Line implements Figure {
0|   void move(int dx, int dy) {
1|     p1.move(dx,dy);  p2.move(dx,dy);
2| }}
3|
4| class Display {
5|   static List figures;
6|   static void update() {
7|    //figures の要素を描画
8| }}
```

図1 ： 図形を表示するプログラム

```
1| aspect DisplayUpdate {
2|    after():    call(void    *.move(int,
   int)){
3|       Display.update();
4|    }
5| }
```

図2 ： ディスプレイ更新のためのアスペクト

３． アスペクトモジュールの生成支援

本研究では、アスペクトモジュールを正しく実装するために、その基準となる既存のモジュールに対する利用例に横断的関心事の手続きを付加させた手続き的なコード、つまり横断的関心事が特定されたコードを開発者が記述し、アスペクトモジュールをツールによって生成する方法を提案する。

開発者は、図３のようにアスペクトをどこにし挿入したいのかを、手続き的なコードで記述します。図３では Point が move メソッドを実行した際に Display.update()を挿入し、また Line が move メソッドを実行した際に line.move で Display.update();を一度だけするように挿入するということをコードに記述する。

図３の横断的関心事が特定されたコードを用いてポイントカットの位置とアドバイスの情報から図４のようなアスペクトが生成される。図 2 の手動でアスペクトモジュールの記述を行った場合は、Display.update();の影響範囲を把握するのが困難であるため、多重更新を行うコードとなっている。しかし、図３のコードに基づいてツールによりアスペクトモジュールの記述を行うことが出来れば、横断的関心事の把握やアスペクトの実装などに悩まされることが減ると期待される。

```
1| void ExamplePoint() {
2|   point.move(1, 2);
3|   Display.update();
4|   //point.move で Display.update();
5| }
6| void ExampleLine() {
7|   line.move(4, 2);
8|   Display.update();
9|   //line.move で
10|    １度だけ Display.update();する
11| }
```

図3 ： Display.update 生成のためのアスペクト

```
1| aspect DisplayUpdate {
2|    after(): call(void move(int, int))
         &&   !cflow(call(void   move(int,
  int))){
3|          Display.update();
4|   }
```

図4 ： 自動生成されたディスプレイ更新のためのアスペクト

４． おわりに

段階的開発手法を想定し、横断的関心事が特定されたコードを用意して用いることにより、アスペクトモジュール記述の支援が可能であることを示した。結果として横断的関心事の把握やアスペクトの実装の誤りなどの問題が緩和されるだろうと考えられる。

今後の課題として、より一般化されたアスペクトモジュールを生成するための手法の検討や、ツールの実装、ツールを使用して実装を行った場合と、手動で行なった場合の比較評価を行い、有効性を検証が必要であると考えられる。

参考文献

[1]Gregor Kiczales, et al. “Aspect-Oriented Programming” in Proc. of ECOOP 97
[2] Gregor Kiczales, et al. “An Overview of AspectJ” in Proc ECOOP ’01