5 アスペクト指向ソフトウェア開発とそのツール

Aspect-Oriented Software Development and Tools

千葉　滋
東京工業大学
chiba@is.titech.ac.jp

オブジェクト指向は理想論？

モデル駆動による開発手法では，まず始めにモデリングありき，である．開発したいシステムをうまくモデル化できなければ，その先へ進めない．

一般に，モデリングはオブジェクト指向で行われる．しかし，近年の研究により，オブジェクト指向では，必ずしもすべてのシステムをうまくモデリングできないことが指摘されるようになってきた．

非常に直感的に説明すると，オブジェクト指向ではシステムを全体から部分へ階層的に分割する．システム全体は，いくつかのコンポーネントからなり，コンポーネントは，いくつかのサブコンポーネントからなり，と徐々に細かく分割していった最小単位がオブジェクトである．オブジェクト指向は，このように，すべてのシステムが，木構造になるように階層的に分割可能であることを前提にしている．

しかしこの還元論にも通じる前提は，必ずしもすべてのシステムで成り立つわけではない．むしろ，多くのシステムで成り立っていないと考えられる．

ある機能を実現するコンポーネントやオブジェクトは，別な機能の実現にはまったく無関係ということがあり得るだろうか？　むしろそうでない方が普通だろう．現実のシステムの構造は，枝がからみあった木？構造をしているのである（**図 -1**）．

実際のモデリングの現場でも，ある機能をどのコンポーネントあるいはオブジェクトに担当させればよいか，決めかねて悩んだ経験はないだろうか？　あるいは，個々のコンポーネントが主だった機能それぞれに対応するように分割したら，いくつかのコンポーネントにまたがってしまい，どの特定のコンポーネントにも属さない機能が出てしまって困ったことはないだろうか？

アスペクト指向が目指すもの

現実の開発では，主要な機能に着目して，has-a 関係や is-a 関係にそってシステムをコンポーネントやオブジェクトに分割してゆくことが多い．そこから漏れた機能については，後から適宜，関係するコンポーネントやオブジェクトにまたがるように，ばらばらに割り当てることになる．図 -1 の薄い水色の部分がこれに該当する．

ここまで「機能」という用語を使ってきたが，これは説明を単純にするためで，正しくは，モデリングや設計，実装に登場するさまざまな観点，と書くべきであった．アスペクト指向では，これを「関心事（concern）」という．システムが提供する機能は，関心事の一例であるが，モデル上のエンティティや，分散処理など実装上の留意点も関心事の例である．

いくつかのコンポーネントやオブジェクトにまたがってしまう関心事は，補助的な取るに足らないものとして，オブジェクト指向では注目されてこなかった．しかしながら，このような関心事は，ソフトウェアの開発や保守を困難にしている元凶の 1 つである．分割統治が基本のソフトウェア開発で，これを妨げるからである．

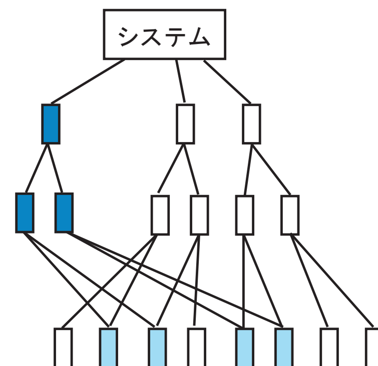

図 -1　枝がからみあった木？構造
（四角はコンポーネントやオブジェクトを表す）

ロギングに関連するコード

```
class Car {
    void start() {
        Logger.log("start");
        engine.ignite();
    }
      :
}

class Computer {
    void turnOn() {
        Logger.log("start");
        disk.start();
        os.boot();
    }
      :
}

class Logger {
    void log(String msg) {
        print(msg);
    }
      :
}
```

図 -2　横断的関心事としてのロギング

アスペクト指向では，このような関心事を「横断的関心事（crosscutting concern）」と呼び，これを独立したコンポーネントに分離することを目指す．横断的関心事を，他の関心事に基づく分割で得られた複数のコンポーネントのあちこちに間借りさせるのではなく，その関心事に対応する専用のコンポーネントにまとめられれば，ソフトウェアの開発効率や保守性が向上する．

アスペクト指向はオブジェクト指向の限界を補完しようとするものである．決してオブジェクト指向を置き換えるものではない．ちょうど手続きや関数による抽象化では不十分な点をオブジェクト指向が補っているように，オブジェクト指向では不十分な点をアスペクト指向は補おうとしている．オブジェクト指向が普及しても，手続きや関数が依然として基礎であるように，アスペクト指向にとってもオブジェクト指向は基礎である．

ロギング

オブジェクト指向では，うまく独立したコンポーネントに分離できない機能の例として，よく取り上げられるのがロギング（logging）である．ロギングとは，システムの実行中にさまざまなメッセージをログ・ファイルに記録することである．記録されるメッセージは，たとえば，デバッグ出力であったり，セキュリティ上の警告であったり，あるいは通常の処理の完了通知であったりする．

ロギングは一見，独立したコンポーネントに分離できそうである．たとえば，ログ・ファイルを準備したり，メッセージを統一的なフォーマットで記録したりする処理は実際，容易にコンポーネントにできる．

しかしながら，そのようなコンポーネントに定義されたメソッドを呼び出すコードは，他の関心事を実現する，いくつものコンポーネントに散らばってしまう（図 -2）．ソフトウェアの開発者は，ログ出力を行いたいプログラム中のすべての場所で，ログ出力用のメソッドが忘れずに呼ばれるように注意して他のコンポーネントのプログラムを書かなければならない．ログ出力用のメソッドの呼び出しまで，ロギングに関連したコードと考えると，ロギングは独立したコンポーネントに分離できない関心事といえる．複数の無関係なコンポーネントにまたがって実装される横断的関心事である．

ロギングはごく簡単な処理だが，コードがあちこちのコンポーネントに散らばってしまう弊害は，意外に現実的なものである．たとえばデバッグ作業中に，デバッグ用に多数のメッセージをログとして出力していたとしよう．無事にデバッグが終わった後はすべてのログ出力用のメソッド呼び出しを手で取り除かなければならないが，あちこちに散らばっていると，一部を取り除き忘れることが多々ある．C/C++ 言語ならマクロを使ってこの問題を避けられるが，Java 言語ではこの問題は深刻である．

画面の再描画

横断的関心事は，GUI（グラフィカル・ユーザ・インタフェース）のプログラミングにも見られる．ボタンなどの GUI の部品を画面上に表示しているとき，ボタンの名前を変えるなど，GUI 部品の表示内容を変えたときには，ウィンドウシステムに再描画処理を依頼して，画面に変更を反映しなければならない．

ウィンドウシステムに再描画処理を依頼するコードを，GUI 部品に共通の親クラスのメソッド redraw として用意し，各 GUI 部品のクラスはそれを継承して再利用するように設計することはできる．ところが，再描画に関連するコードをすべて redraw メソッドにまとめられるわけではない．開発者は，各 GUI 部品のクラスを調べて，表示内容の変更をともなうメソッドが呼び出された後は常に，redraw メソッドも一緒に呼び出される

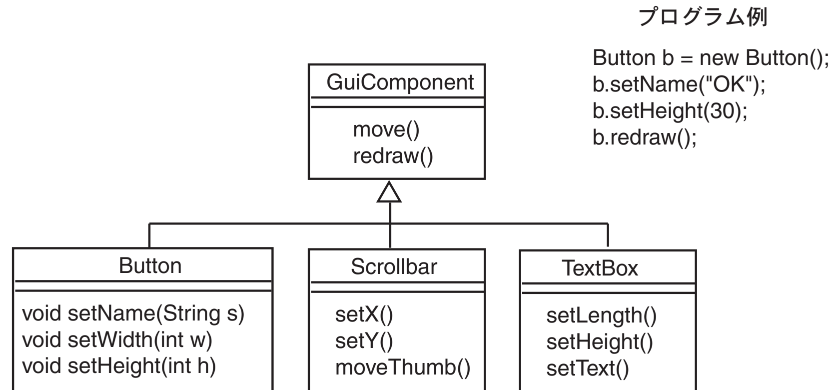

図-3　部品の形状を変えるメソッドを呼んだ後は必ず redraw() を呼ぶ

ように，注意してプログラムを書かなければならない．ロギングと同様，redraw メソッドを呼び出すコードまでを，再描画に関連するコードと考えると，再描画処理もあちこちのクラスに散らばってしまうといえる．

たとえば**図-3**では，GUI 部品のクラスとして Button，Scrollbar，TextBox の 3 つがある．これらのクラスのメソッドのうち，部品の形状を変更する setName や setHeight のようなメソッドが呼ばれた後には，親クラス GuiComponent の redraw メソッドも必ず呼び出されなければならない．redraw メソッドの呼び出しを setName や setHeight メソッドの中に含めてしまう方法もあるが，関連するコードが複数のクラスに散らばってしまっていることには変わりない．開発者は，GuiComponent を継承しているすべてのクラスのメソッドのうち，部品の形状を変更するメソッドが，最後に必ず redraw メソッドを呼んでいるか，注意しながらプログラムを書かなければならない．また，この方法では redraw メソッドが無駄に連続して呼ばれてしまうことがあるので，実装上，好ましくない．同じオブジェクトについて setName と setHeight メソッドが連続して呼ばれるときは，redraw メソッドの呼び出しを最後の 1 回だけにしたい．

このように横断的関心事は，継承機構を駆使しても，独立したコンポーネントに分離することができない．多重継承機構を使った場合も同様である．継承機構を使えば，複数のクラスに共通するメソッドを親クラスにまとめることは可能である．しかし，複数のクラスのメソッドの中に共通して現れる処理を 1 カ所にまとめるためには，多重継承機構であっても不十分である．

ポイントカットとアドバイスによる横断的関心事の分離

横断的関心事を単一のコンポーネントにまとめられるようにする，このような技術を総称してアスペクト（aspect）指向技術という．アスペクトとは側面の意味である．ソフトウェアのどんな側面，つまり横断的関心事，もコンポーネント化可能にするのがこの技術の目的である．

具体的なアスペクト指向技術としてはいくつかの種類があるが，ここでは代表的なものとして，AspectJ で採用されているポイントカット（pointcut）とアドバイス（advice）を用いた技術を紹介する．AspectJ とは，Java 言語にアスペクト指向技術のための拡張をほどこしたプログラミング言語である．したがって以下ではプログラミングを例に説明を行うが，基本的な概念はアスペクト指向ソフトウェア開発にも共通する．

継承機構の非常におおざっぱな（かつ一面的な）理解の仕方として，親クラスで定義されたメソッドは，コピーされて，すべての子クラスの定義に自動的に挿入される，というものがある（**図-4**）．これによって開発者はメソッドを単一コンポーネント（親クラス）にまとめて定義することができる．そのメソッドを必要とする場所（子クラス）には，言語処理系が自動的にそのメソッドを挿入するので，開発者が個別に手で挿入する必要はない．

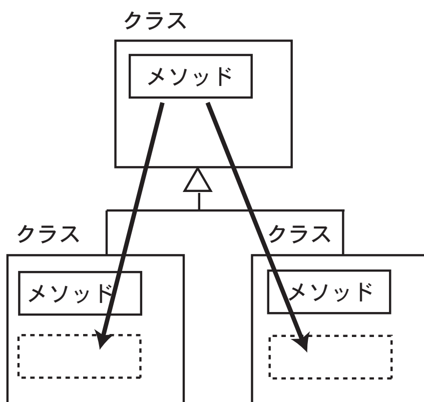

図-4　継承機構によるメソッドの挿入

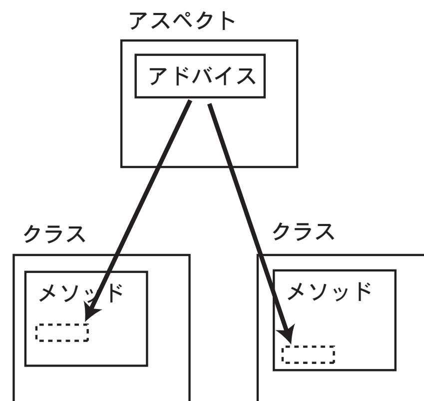

図-5　AspectJにおけるアドバイスの挿入

横断的関心事を単一コンポーネントにまとめられるようにするため，このような機構のアスペクト指向版がAspectJで採用されている．AspectJでは，classやinterfaceのほかに，aspectというコンポーネントを宣言できる．以下はRepaintという名前のaspectの宣言である．

```
1: aspect Repaint {
2:   after(GuiComponent gui):
3:     (call(void GuiComponent+.set*(..))
4:       ||call(void Scrollbar.moveThumb(..)))
5:     && target(gui){
6:     gui.redraw();
7:   }
8: }
```

aspectの中には，いくつかの「アドバイス（advice）」を書くことができる．アドバイスは，classでいうメソッドに相当する．上の例では，2から7行目でアドバイスを1つ宣言している．

このアドバイスの宣言により，GuiComponentクラスのサブクラスで定義されており，setから始まる名前を持つメソッドを呼び出した直後には，必ずGuiComponentクラスのredrawメソッドが呼び出されるようになる．またScrollbarクラスのmoveThumbメソッドの呼び出し直後にもredrawメソッドが呼び出される．

アドバイスの宣言は，3から5行目のポイントカットと呼ばれる部分と，{}で囲まれた6行目のアドバイスの本体からなる．ポイントカットで指定された場所にプログラムの実行が到達すると，アドバイスの本体がちょうどメソッドのように実行される．2行目のafterは，プログラムの実行が指定された場所に到達した直後に，アドバイスの本体を実行することを意味する．guiはメソッドの引数のようなものである．例の場合は，setNameやsetHeightメソッドが呼び出される対象のオブジェクトがguiの値となる．

メソッドが継承機構によって子クラスの定義に自動的に挿入されたように，アドバイスの本体は，メソッド本体の中の，ポイントカットで指定された場所に自動的に挿入されると考えられる（**図-5**）．もちろん，継承機構と異なり，挿入先のメソッドが，共通の親クラスを持つクラスのメソッドである必要はなく，まったく関係のないクラスのメソッドであってもよい．AspectJでは，さまざまなアドバイスの挿入場所を柔軟に指定できるようにポイントカットの記述方法が工夫されている．

なお，アドバイスの本体が挿入されるといっても，実装上も，アドバイスのコードのコピーが実際にメソッドの中に挿入されるとは限らない．現行のAspectJの処理系では，アドバイスの本体は特別なメソッドとしてコンパイルされ，メソッドの中に挿入されるコードは，この特別なメソッドを呼び出すためのコードだけである．

ポイントカットで指定できる場所とは？

アドバイスの本体がいつ実行されるかを決めるのがポイントカットである．AspectJでは，ポイントカットで指定可能な場所のことをジョインポイント（joinpoint）と呼ぶ．例に出てきたメソッド呼び出しのほかに，フィールドの参照や，オブジェクトの生成もジョインポイントである．

先に示した例では，ポイントカットで，setName等のメソッドが呼び出された直後を指定した．しかしポイントカットで指定できる場所は，ソースコード中の特定

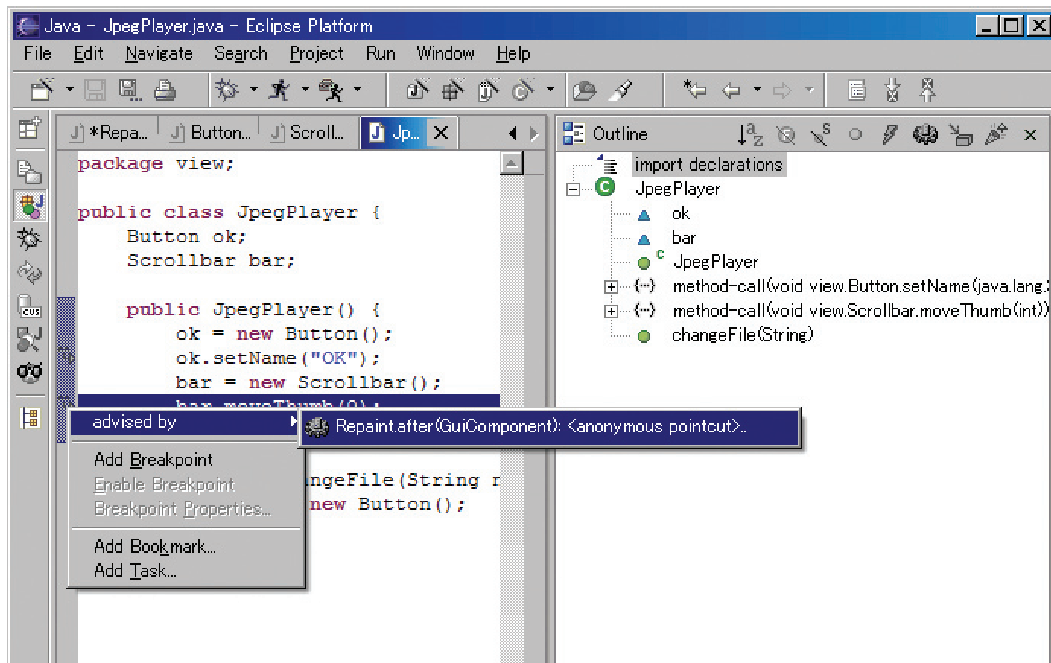

図-6　Eclipse 上での AspectJ による開発

の場所，というわけではない．AspectJ のおけるジョインポイントの正確な定義は，プログラム実行の途中のある時点，である．したがって，たとえばプログラムの実行が，ソースコード中のあるメソッドを呼び出している場所に到達して，かつ，呼ばれているオブジェクトのクラスが指定のものであったときだけ，アドバイスの本体を実行するように，ポイントカットを記述することができる．Java 言語では呼ばれたオブジェクトの静的な型と，そのオブジェクトの実際の型が一致するとは限らない（後者が前者のサブクラスである可能性がある）ので，アドバイス本体を実行するか否かは実行時でないと決まらない．

このように，アドバイスの本体は，プログラムの実行がソースコード中の指定された場所に到達し，かつ，その時の実行状態が指定された条件を満たしているときに実行される．このように理解すると，先に説明した，アドバイスの本体がポイントカットで指定された場所に挿入される，という説明は正確ではないことが分かる．しかし，ポイントカットとアドバイスの基本的なアイディアを理解するだけなら，先の説明でも十分だろう．

開発支援ツールが重要

AspectJ で書かれたプログラムでは，プログラムの実行途中に，ポイントカットで指定された場所で，暗黙のうちにアドバイスの本体が実行される．このため，ポイントカットの指定を間違えると，予期せぬところでアドバイスの本体が誤って実行される危険がある．普通のプログラムの場合は，プログラムを 1 行ずつ丹念に追ってゆけばどこで何が実行されるかが明らかになるが，AspectJ のプログラムの場合，ポイントカットが aspect の宣言中に分離されているので，なかなか明らかにはならない．

アスペクト指向技術の研究者の中にも，この点はアスペクト指向の弱点である，と指摘する人たちがいる．そのような人たちは，アスペクト指向のこの性質を obliviousness（忘れっぽさ）と呼ぶ．一方で，これを逆にアスペクト指向の利点と考える研究者もおり，対抗して同じ性質を noninvasiveness（非侵入性）と呼ぶ．この性質が利点であるのは，関心事の実装を分離するという視点からは，ポイントカットが aspect の宣言中に分離されており，望ましいからである．

非侵入性を利点であると考える研究者は，この性質の負の側面は適切な開発支援ツールの利用で回避できると考える．たとえば最近人気のソフトウェア開発環境である Eclipse 上で AspectJ による開発を行うためのプラグイン AJDT は，ポイントカットで指定されたソースコード中の場所を，エディタ上に対話的に表示する機能がある．図-6 は AJDT を使っている様子だが，表示されたソースコードの左端に，ポイントカットで指定されている行を示すマークが見える．図-6 は，下のマークをクリックし，どの aspect のアドバイスがその行で実行さ

れるのかを調べているところである．

AspectJ を 1 次元のテキストで表現される旧来のプログラミング言語と考えると，非侵入性は大きな弱点になる．そう考える研究者は，ポイントカットで指定されている行を示すマークを，AJDT の代わりに開発者がプログラム中に明示的に書くべきだと指摘している．つまり，

```
1: Button ok = new Button();
2: pointcut(Repaint):
3:         ok.setName("OK");
```

の pointcut(Repaint) のような，何らかのマークを書かせようというのである．しかし，このアイディアは，開発者がマークを入れ忘れたり，誤った場所に入れてしまう可能性があるので，あまりよいとはいえない．

一方，AspectJ のプログラムは AJDT のような常に開発ツールを使って書かれると考えると，非侵入性は利点である．開発者がマークを入れ間違えることもなく，また，マークの表示を必要に応じて on/off すれば，aspect で実装されている処理に関心がないときは off にして，それを忘れることができる．

モデリングにおけるアスペクト指向

AspectJ のポイントカットとアドバイスによるアスペクト指向技術は，単なるプログラミング上のテクニックではない．その本質は，モデリングの際は，全体の整合性を無理に考えずに個々の関心事ごとに別々に対象をモデリングし，得られたばらばらのモデルをシステムとしてどう統合するかは別途与える，というものである．AspectJ のアドバイスは，前者の関心事ごとのモデルを表現するためのもので，ポイントカットは，それらのモデルの統合を規定するためのものである．

従来の技法では，このような 2 段階の手順をふまず，すべての関心事を統合したモデルを一足飛びに作成していた．しかし，このような方法で得られるモデルは，本稿の最初に指摘したように，取り扱いが難しい．そのようなモデルは，開発者が直接扱うべきではない，というのがアスペクト指向の立場である．

まとめ

アスペクト指向技術の研究はプログラミング言語から始まったが，現在では，徐々にソフトウェア開発の上流にも利用されつつある．アスペクト指向技術は，ソフトウェアをさまざまな側面からモデル化し，そのモデルにそって実装まで行えるようにする技術である．それぞれのモデルは互いに重なり合っているので，実現には，従来にないモデルの表現技術が必要である．特に 1 次元のテキストによるプログラムの表現では，互いに重なり合うモデルをうまく表現できない．今後は AJDT のような開発ツールのかたちで，アスペクト指向技術の利点をより引き出すようなモデル表現が，開発されてゆくものと思われる．

参考文献

1）AspectJ, http://www.eclipse.org/aspectj
2）AJDT, http://www.eclipse.org/ajdt

（平成 15 年 11 月 26 日受付）