

オブジェクト指向からアスペクト指向へ

written by ている (tail@oucc.org)

Last Update: 2002/11/12

1 はじめに

本稿では、アスペクト指向プログラミングについて紹介します。この記事を読まれる方の大半は、何らかの雑誌や書籍などでオブジェクト指向プログラミングという言葉を目にしたことがあると思いますが、アスペクト指向プログラミングという名前については初耳だという方のほうが多いのではないのでしょうか。

ここ十数年、オブジェクト指向プログラミングが広く使われるようになってきましたが、それに伴って、オブジェクト指向の問題点というのも次第に判明してきました。アスペクト指向プログラミングはその問題点を解決するために提案され、まだ五年ほどしか経過していない新しいプログラミングのパラダイムです。^{[AOP96][AOPatDSL97]}

この記事ではまずオブジェクト指向について簡単に説明し、その問題点、オブジェクト指向ではうまく対処できないような要素について説明します。そして、アスペクト指向プログラミングのベースとなった Metaobject Protocol という仕組みについて簡単に触れてから、その発展であるアスペクト指向の基本的な考え方、また用語、言語の特徴、そしてそれを使うことでどのような利点があるのか、応用としてどのようなものがあるのか、説明していきます。

2 オブジェクト指向プログラミング

2.1 オブジェクト指向の特徴

オブジェクト指向の特徴を、箇条書きにして書き出してみました。

- 「オブジェクト」という概念の導入
 - ✓ システムをオブジェクトに分解
 - ✓ オブジェクトの協調動作でシステムが動作
 - ✓ 分析から実装まで一貫したモデルを使える
- 機能性の分類と階層化

オブジェクト指向の最大の特徴は、やはりオブジェクトという概念を導入したことにあります。図1を見てください。従来は、データと機能は分離して扱うのが常でした。しかし、データの分析と機能の分析を別個に行うため、実装段階でそれらのマッピングを取ることが問題となりました。これに対してオブジェクト指向では、関連したデータと機能を合わせて(システム上での役割という単位で)「オブジェクト」として取り扱い、システムをオブジェクトの相互通信による協調動作という形でモデル化します。これによってシステムの分析から実装まで一貫したモデル化を行うことができます。このような一貫性によ

って、システム全体の見通しがよくなり、メンテナンス性の向上にもつながります。^[URL1]

また、同種の機能を持ったオブジェクトをクラスという単位で分類することにより、複数のオブジェクトに共通した知識を階層化して記述することができます。これによって、オブジェクト間の関係を明確に表すだけでなく、プログラムコードの量を減らし、またオブジェクトの再利用性を向上させることができます。

これらオブジェクト指向プログラミングの詳細については、Meyer などが詳しく説明しています^[Meyer]ので、そちらを参照してください。

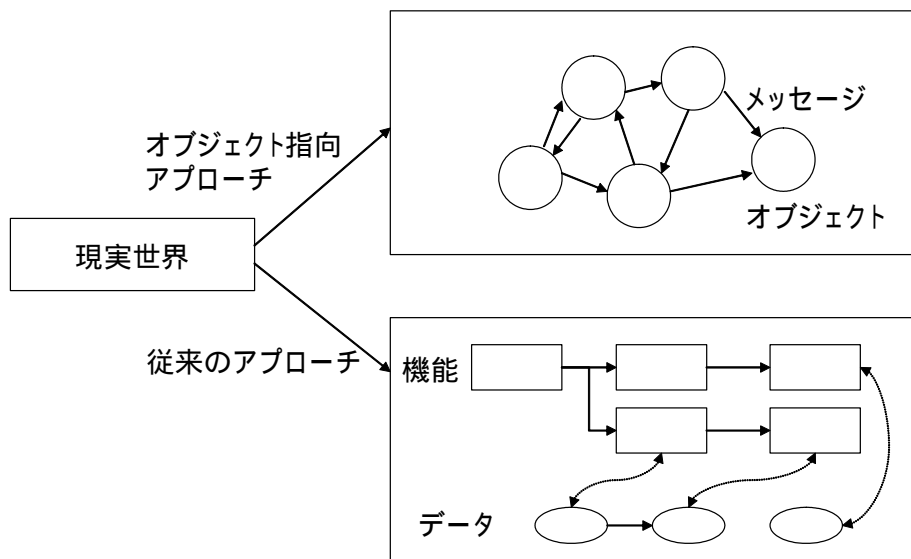


図 1

2.2 オブジェクト指向の問題点

オブジェクト指向では、オブジェクトの階層化という形でシステムの機能の分類やデータの隠蔽が行われます。しかし、単一のオブジェクトの中に閉じ込められない処理、というものも世の中には存在しています。例を次に示します。

- ログイン
- 同期処理
- オブジェクトの多重化

ここでいうログインというのはプログラムの各時点での動作を記録すること、同期処理は複数のオブジェクト間での排他制御（あるオブジェクトに対して、他のオブジェクトからのアクセスを完全に排除して独占的にアクセスすること）を意味します。オブジェクトの多重化というのは、複数のオブジェクトに同一のデータを保持することで、ひとつのオブジェクトが計算誤りを起こした場合に他のオブジェクトからデータをもらうことで誤り訂正する機能を意味します（オブジェクトをネットワーク上の複数のコンピュータ上に配

置するなどして利用します)。

これらの処理はいずれも複数のオブジェクトが関連する動作であるため、それらのオブジェクトそれぞれに処理を分散して記述する必要があります。このような処理のことをシステムの「横断要素(Crosscutting Concerns)」と呼び、これらのプログラムコードがオブジェクト上で絡み合うことから、プログラムの見通しの悪さ、メンテナンス性の悪化の原因となります。例として、次のプログラムを示します(言語としては Ruby 風ですが、文法などはあまり気にしないでください)。

```
class SampleClient
  def getServerResponse
    # この処理が実行されたことを記録する(会社のコーディング規則として)
    log.outputMessage "begin getServerResponse"
    # サーバへのアクセスは排他制御が必要
    server.lock
    # サーバから返答を得る
    data = server.getResponse
    # ロギング(要求仕様の P.15)
    if data.isValid then server_log.outputMessage "getServerResponse success"
    else server_log.outputMessage "getServerResponse failure"
    # サーバの排他制御を解除
    server.unlock
    # ロギング(会社のコーディング規則)
    log.outputMessage "end getServerResponse"
  end
end
```

このプログラムは、「サーバへのアクセスは排他制御が必要である」「先頭と最後には常にロギングが実行される」「サーバに対する処理の成否が記録される」といったルールに基づいて記述されています。そのことはコメントによってのみ記述されていますが、プログラムとしてこれらのルールを記述しているわけではありません(これらのルールに従わないプログラムを書くことは簡単にできてしまいます)。このようなプログラムコードがあちこちに分散・増殖していき、その後ルールが変更された場合には多大な影響が発生します。また、本質的な処理が何かという見通しが悪くなり、後でプログラムを変更・再利用する際には、このようなルールを一度除去して本質的な機能部分だけを取り出す、という作業が待っていることとなります。

これらのルールを、単一のオブジェクトとして記述することは不可能です。では、この

ような処理を記述するにはどうすればいいのでしょうか。その解決方法のひとつとして、Metaobject Protocol と呼ばれる技術があります。

2.3 Metaobject Protocol

Metaobject Protocol(以降 MOP と略します)は、オブジェクトのメッセージ通信に注目した技術で、Smalltalk や CLOS などのオブジェクト指向言語で利用されているものです。他の分野では、メッセージフックなどと呼ばれていることもあります。図 2 に簡単なメッセージ通信のモデルを示します。通常は Sender オブジェクトが Receiver オブジェクトに対してメッセージを送ります。しかし、本来 Receiver オブジェクトに送られたメッセージを、「メタオブジェクト」と呼ばれる別のオブジェクトが受け取り、代わりに処理を実行することを許すことにします。メタオブジェクトはオブジェクトにメッセージを送って本来の実行結果を取得したり、あるいは別のオブジェクトにメッセージを送って、まったく別の処理を行うこともできます。メタオブジェクトはメッセージ受信の代理人であるというだけで、その性質自体は通常のオブジェクトと変わりません。このようなメタオブジェクトを利用するための機構のことを MOP と呼びます。MOP の利点は、本来メッセージ通信を行っているオブジェクトをまったく書き換えることなしに、プログラムの挙動を変えることができるという点にあります。

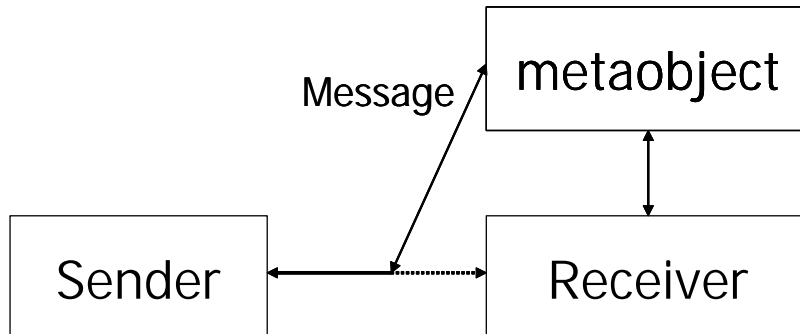


図 2

プログラムを MOP を用いて記述する場合、メタオブジェクトが「どのオブジェクトへの」「どのメッセージを」代理で受け取るかということ、その際の処理と同時に記述することになります。例として、先ほどの登場したプログラムを MOP を用いて書き直すと、次のようになります(これも言語としては Ruby 風ですが、本来の文法に従っているわけではありません)。

```
# 本来のオブジェクトのコード
class SampleClient
```

```

def getServerResponse
  data = server.getResponse # サーバから返答を得る
end

end

# 会社のコーディング規則によるロギング : getServerResponse に対して動作
def-MOP logging : SampleClient.getServerResponse
  log.outputMessage "begin getServerResponse"
  Receiver.getServerResponse
  log.outputMessage "end getServerResponse"
end

# サーバへのアクセス(server.getResponse)の前後には排他制御が必要(他の MOP より優先)
def-MOP lock : Server.getResponse dominates all
  server.lock
  data = server.getResponse # 本来実行しようとした処理を実行
  server.unlock
  return data
end

# サーバアクセスへの成否を記録 (要求仕様の P.15)
def-MOP validation : Server.getResponse
  data = server.getResponse
  if data.isValid then server_log.outputMessage "getServerResponse success"
  else server_log.outputMessage "getServerResponse failure"
  return data
end

end

```

このような記述によって、オブジェクトの本来の機能だけをオブジェクトに記述し、ロギングやアクセス制御に関する処理をメタオブジェクト側に追い出してしまいました。今回のようなケースにおいては、それほど有効ではないように思われるかもしれませんが、複数のサーバへのアクセスを行っていたり、ロギングのルールが複雑だったりした場合は、この分解は非常に有効に働きます。これと同様に、オブジェクトを横断するような要素をオブジェクトから分離し、独立したモジュールとして記述してしまおう、というのがアスペクト指向プログラミングです。

3 アスペクト指向プログラミング

アスペクト指向プログラミングの目的は、簡単に言うと、「複数のオブジェクトを横断するような要素を、別のモジュールに分離して記述してしまおう」ということを目的として

います。この分離記述するモジュール単位の名前をアスペクトと呼びます。これはクラスという分類単位とは異なった分類単位となっています。

先ほど、MOP ベースのプログラムでは、「どのオブジェクト、どのメッセージに対して Metaobject をセットしておくか」という単位でメタオブジェクトを記述しましたが、アスペクトはそれと同様に「どの join point で、どのような処理を実行するか」という形式でプログラムを記述します。

新しい用語として、join point というものが出てきました。join point とは、プログラム内に含まれるいくつかの実行時点のことです。join point の例としては、次のようなものがあります。

- ✓ call (メソッド呼び出し、メッセージ送信)
- ✓ execution (メソッド本体、メッセージ受信)
- ✓ get (フィールド参照、オブジェクト内部データの参照)
- ✓ set (フィールドへの代入、オブジェクト内部データの変更)
- ✓ initialization (オブジェクト初期化)

この中から、オブジェクトの属するクラスなどの条件や集合演算を用いて、興味のある(処理を結合したい)集合を取り出します。このようにして取り出された join point の集合のことを pointcut と呼びます。利用できる join point やその演算は、処理系によって異なります。

次に、先ほど MOP で記述したプログラムを AspectJ と Java で記述して示します(アスペクトとしてはロギングの例だけを示し、サーバの排他制御などは省略しました)。

```
// 本来のオブジェクトのコード
class SampleClient {
    ServerData data;

    public void getServerResponse {
        data = server.getResponse // サーバから返答を得る
    }
}

//コーディング規則によるロギング:
// 会社で作成したパッケージ(jp.co.foo.*)のクラスは、
// メソッド実行の開始と終了を記録しなければならない。
// ただし、private メソッド呼び出しは記録しなくてよい。
aspect CompanyLoggingAspect {
    // アスペクトの共有オブジェクトを宣言
    static Logger log = new Logger();

    // メソッド呼び出し点(メソッドの名前、引数、戻り値をワイルドカードで記述)
```

```

pointcut all_method_execution() : execution(* * jp.co.foo.*(..));
pointcut private_method_execution() : execution(private * jp.co.foo.*(..));
// 取り出された pointcut に結合される処理を記述する
Object around logging() : all_method_execution() && !private_method_execution() {
    log.outputMessage("begin", thisJoinPoint.getSignature());
    proceed();
    log.outputMessage("end", thisJoinPoint.getSignature());
}
}

```

この例では、アスペクトの内部に、明快にロギングのルールが記述されています。この記述されたアスペクトには、元のオブジェクト側のコードに関する情報がほとんど埋め込まれていない（ワイルドカードを用いて一般的に記述されている）ため、オブジェクトのコードとは独立して、ロギングのルールなどを変更することができるようになります。また、オブジェクト側からはロギングに関する処理のすべてが取り除かれており、本来の機能が明快に表現されるようになっていきます。これによって、ロギングに依存せずにこのオブジェクトのコードを別の部分で再利用することができるようになります。これが、アスペクト指向プログラミングの有効な例といえるでしょう。

このようにして記述されたアスペクトは、その処理系によって、コンパイル時や実行時に元のようなクラスで書かれたプログラムに変換されます（アスペクトを直接扱える環境は今のところありません）。図3は、HTTP Serverにおけるセッション管理の、クラスでの実現とアスペクトでの実現の関係を表したものです。アスペクトはその処理系によってクラスによる実現に変換されるため、プログラムの実行環境は従来のものをそのまま利用できます。図中の矢印は依存関係を表しており、クラスによる記述に比べ、アスペクトによる記述のほうが相互依存関係が解消されていることがわかります。相互依存関係の解消は、プログラム変更の手間を減少させるだけでなく、プログラムの欠陥の検出を容易にするなど、様々な形で保守性を向上させます。 [Lakos]

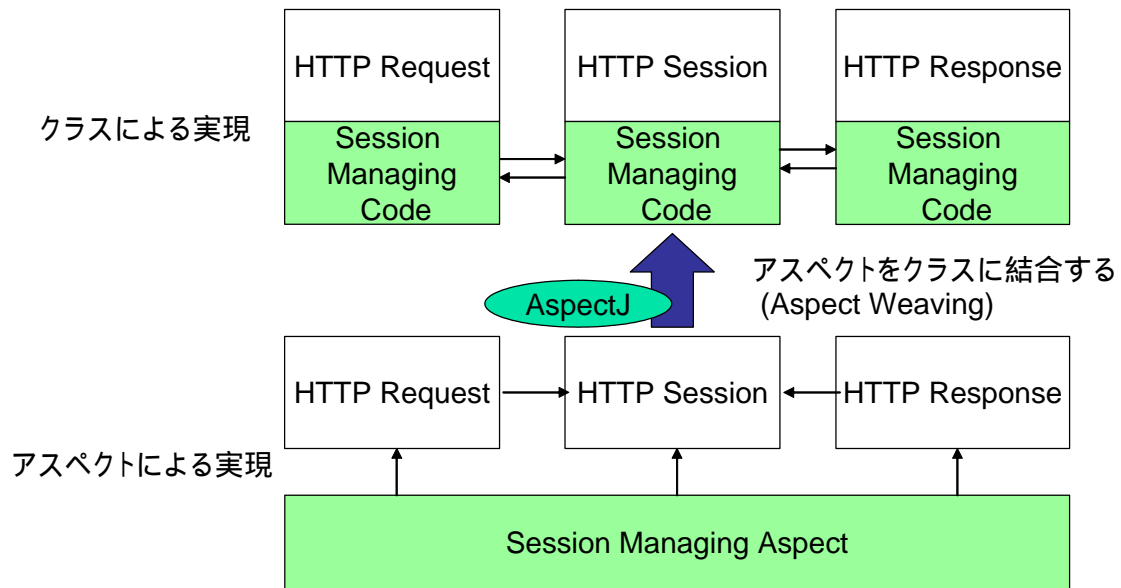


図 3

4 まとめ

本稿では、新しいプログラミング手法のひとつ、アスペクト指向プログラミングについて紹介しました。アスペクトは、複数のオブジェクトを横断するようなプログラム要素（その多くはルール、制約条件などですが、今後はそれ以外にも発展していくと思われます）を記述するためのモジュール単位で、これを用いることでプログラム全体の見通しをよくするだけでなく、オブジェクトとアスペクトの独立した再利用性を向上させます。

アスペクト指向プログラミングには、それに対応した新しいプログラム設計方法の考案やアスペクト間の干渉問題など、まだまだ課題は数多く残されています。しかし、ここ数年「プログラムを横断するような要素の分離(Separation of Crosscutting Concerns)」をキーワードとした多くのプログラミング手法やツールが提案されるようになっており、アスペクト指向がこれからのプログラミングのひとつの方向性を示すものであることは間違いないでしょう。

興味を持たれた方は、一度 AspectJ の公式サイト^[URL2]やアスペクト指向技術の紹介サイト^[URL3]を訪れてみてください。AspectJ の公式サイトではツールとそのマニュアル以外にも数々の分かりやすいチュートリアルや論文が紹介されていますし、またアスペクト指向技術に関連したいくつかの論文やツールが公開されています（これらはすべて英語です。残念ながら、まだ日本語のサイトは数が少なく、用語の和訳も統一されていません）。アスペクト処理系・記述ライブラリの実装は AspectJ(Java), AspectR(Ruby)など、現段階でもいくつかの言語で利用可能になっています。ぜひ、一度使ってみてください。

疑問や感想、意見などありましたら、tail@oucc.org までメールしてください。筆者はア

スペクト指向プログラミングについて人に説明する（説明せざるを得ない）機会が多いため、説明の流れや意味の把握しやすさなど、些細なことでもかまいませんので、フィードバックしていただくと幸いです。

参考文献

[AOP96] Gregor Kiczales et al., Aspect-Oriented Programming, ACM Computing Surveys Vol. 28, No. 4es, 1996

[AOPatDSL97] Gregor Kiczales et al., Aspect Oriented Programming, DSL '97 - First ACM SIGPLAN Workshop on Domain-Specific Languages, 1997

[Meyer] Bertland Meyer 著, 二木厚吉 監訳, オブジェクト指向入門, アスキー, 1990

[Lakos] John Lakos 著, 滝沢 徹, 牧野 祐子 訳, 大規模 C++ソフトウェアデザイン, ピアソン, 1996

[URL1] オブジェクト指向技術セミナー, http://www.njk.co.jp/otg/Study/index_ns.html

[URL2] AspectJ Official Site, <http://aspectj.org/>

[URL3] Aspect Oriented Software Development, <http://aosd.net/>