よっては係員の操作が義務づけられているところもあります。

- 実生活における Facade の例をもう一つ挙げてください。

### 6.7.3 あなたの意見

□1. 既存システムが提供していない機能を必要とする場合、Facade パターンを使用することができるでしょうか？

□2. Facade パターンを使ってシステム全体をカプセル化する理由は何でしょうか？

□3. Facade を用いて従来のシステムをカプセル化するのではなく、新たなシステムを作成するようなケースはあるのでしょうか？ それはどういったものでしょうか？

□4. GoF がこのパターンを Facade と呼んだのは何故だと思いますか？ これは行っていることに合った適切な名前でしょうか？ その答えと理由を述べてください。



# 第7章

# Adapterパターン

## 7.1 概 要

章の概要

では、次に Adapter パターンを見ていくことにしましょう。Adapter パターンは非常に一般的なパターンであり、他の多くのパターンとともに利用されます。

この章では、以下のことを解説しています。

- Adapter パターンとは何か、そしてその使用局面
- このパターンの鍵となる特徴
- このパターンを使用したポリモーフィズムの説明
- さまざまな詳細レベルに UML を使用する方法
- Adapter パターンと Facade パターンの比較を含む、今までの経験から見た Adapter パターンについての所見
- CAD/CAM の問題と Adapter パターンの関連

## 7.2 Adapterパターンの紹介

目的：新たなインタフェースを生成する

GoF によれば、Adapter パターンの目的は、以下のようなものとなっています。

> **あるクラスのインタフェースを、クライアントが望むインタフェースに変換する。Adapter によってクラス群は互換性のあるインタフェースを持つことになり、協調して動作できるようになる**[*1]。

この文章が基本的に意味していることは、オブジェクト自体の動作が望み通りであるものの、インタフェースが望み通りでない場合、新たなインタフェースを生成する必要があるということなのです。

[*1] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston: Addison-Wesley, 1995, p. 139
邦訳は『オブジェクト指向における再利用のためのデザインパターン』（ソフトバンクパブリッシング刊）ISBN-4-79731-112-6

## 7.3 Adapterパターンの学習

典型的な例：クライアント側オブジェクトを詳細から解放する

Adapterパターンの目的を理解する最も簡単な方法は、その実例を見てみることです。例えば、以下のような要求があったと考えてください。

- `display`（表示する）という振る舞いを有した、**Point**（点）、**Line**（直線）、**Square**（四角形）というクラスを生成してください。
- クライアント側オブジェクトは、自分が保持しているものが実際に点、直線、四角形であるかどうかを知るべきではありません。それらが、いずれかの形状であるということを知っておくだけにしたいのです。

言い換えれば、特定の形状を「表示可能な形状」という、よりレベルの高い概念として捉えたいわけです。

この例とよく似たシチュエーションに、以下のようなものがあります。

- 他人の作成したサブルーチンやメソッドが、あなたの必要な機能を実現しているため、それを使用したい。
- それらを直接、あなたのプログラムに取り込むことはできない。
- それらのインタフェースや呼び出し方法は、あなたのプログラム中の関連オブジェクトと整合性を持っていない。

…これによって詳細を共通の方法で扱えるようになる

言い換えれば、システムに点、直線、四角形を**持ち込む**ものの、それらを形状として**扱いたい**というわけです。

- こういったことが実現できれば、クライアント側はこれらのオブジェクトを同じ方法で扱えるようになり、それぞれの詳細に注意を払わなくてもよくなるのです。
- また、将来新たな形状が追加されたとしても、クライアント側を変更する必要がなくなるのです（図7.1を参照してください）。

その実現方法：派生クラスを多態化する

これには、ポリモーフィズム（多態）を利用します。つまり、システム中には複数のオブジェクトを持ち込むものの、こういったオブジェクト群とのやり取りは、共通の方法で行えるようにするわけです。

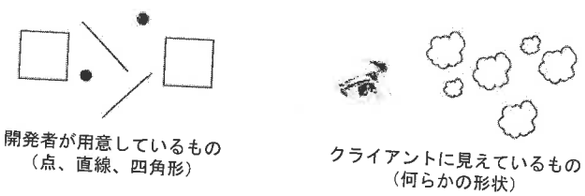


図7.1 開発者が用意しているオブジェクト群は、、、すべて「形状」に見えなければならない



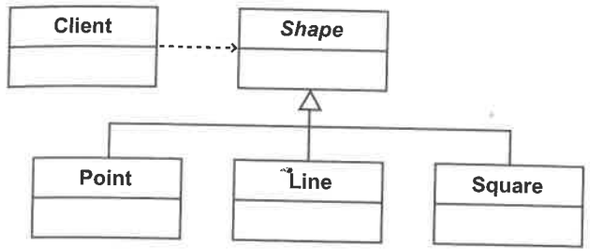


図7.2 **Point**、**Line**、**Square**は***Shape***の一種である

こうすることで、クライアント側のオブジェクトは、点、直線、四角形に対して、自らを描画させる、あるいは消去させる等の指示を行うだけで済むようになるのです。指示を受けたそれぞれの点、直線、四角形は、自らの型に合った適切な振る舞いを、責任を持って実行するわけです。

こういったことを達成するにはまず、***Shape***（形状）クラスを生成し、そのクラスから**Point**（点）、**Line**（直線）、**Square**（四角形）といったクラスを継承します（図7.2を参照してください）。

実現方法：インタフェースを定義し、派生クラスで実装を行う

まず、***Shape***が提供する特定の振る舞いを指定する必要があります。これを行うには、***Shape***クラス内にインタフェースを定義した後、その派生クラス毎に適切な振る舞いを実装します。

***Shape***が提供する特定の振る舞いとは以下の通りです：

- ***Shape***の位置を設定する（`setLocation()`）。
- ***Shape***の位置を取得する（`getLocation()`）。
- ***Shape***を表示する（`display()`）。
- ***Shape***を塗りつぶす（`fill()`）。
- ***Shape***の色を設定する（`setColor()`）。
- ***Shape***を消去する（`undisplay()`）。

これらを表現したものが図7.3です。

新たな形状を追加する

この後、新たな***Shape***として**Circle**（円）を実装するよう要求されたと考えてください（要求とは常に変化するものなのです！）。この要求に対応するには、***Shape***クラスから**Circle**クラスを派生させ、実装するだけです。これによって、ポリモーフィズムに則った振る舞いを実現できるようになります。

…しかし外部にある機能を流用したい

次の作業は、**Circle**の`display()`、`fill()`、`setColor()`、`undisplay()`といったメソッドの作成となります。これは結構大変な作業となります。

しかし幸運なことに、周囲を見渡すと（優れたプログラマは常に周囲を見ているのです）、廊下の突き当たりで仕事をしているジルが、円を取り扱う**XXCircle**というクラス（図7.4を参照してください）をすでに開発していたことが判ったのです。ただ残念なことに、彼女の付けたメソッド名は私のプロジェクトの規約とは異なったもの

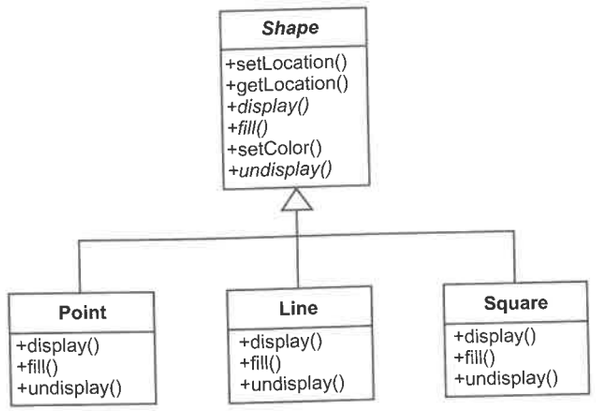


図7.3 Point、Line、Squareとそのメソッド

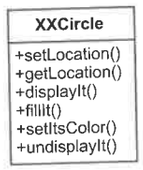


図7.4 ジルの作っていたXXCircleクラス

だったのです。彼女の付けたメソッド名は、以下のようなものでした：

- `displayIt()`
- `fillIt()`
- `setItsColor()`
- `undisplayIt()`

**XXCircleを直接使うことができない**

私は ***Shape*** を用いたポリモーフィズムが必要であったため、**XXCircle** を直接使うことができないのです。その理由は2つあります：

- **名前やパラメータリストが異なっている**—**XXCircle** におけるメソッド名と引数の並びが、***Shape*** におけるそれと異なっています。
- **派生させることができない**—こういった名前が異なっているだけでなく、ポリモーフィズムを利用するには、クラス自体を ***Shape*** から派生させる必要があります。

ジルに対して、メソッド名を変えて、**XXCircle** の派生元を ***Shape*** にしてくれとお願いするわけにはいきません。そんなことをすれば、すでに **XXCircle** を使用しているオブジェクトすべてに影響が発生するでしょう。また、ジルからコードをもらって



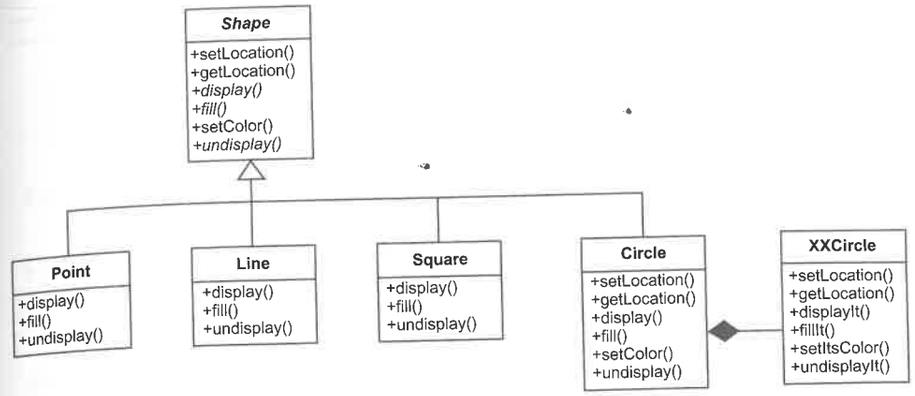


図7.5 Adapterパターン：**Circle** は **XXCircle** クラスの「ラッパ」となる

修正するにしても、他人のコードですから、思いがけない副作用を作り込んでしまう危険性があります。

ほとんどのものが手に届くところにあるのに、それらを使うことができず、また書き直すこともできないのです。どうすればいいのでしょうか？

答えは、既存コードを修正するのではなく、既存コードを**適合**（*Adapt*）させればよいのです。

つまり、***Shape*** から派生させることで新規クラスを作成し、**XXCircle** 中にある円の実装を使用することで ***Shape*** のインタフェースを実装すればよいのです（図7.5を参照してください）。

**実装方法**

- ***Shape*** から **Circle** クラスを派生させる。
- **Circle** に **XXCircle** を保持させる。
- **Circle** オブジェクトに対するリクエストを **XXCircle** オブジェクトに転送する。

図7.5において、**Circle** と **XXCircle** の間に引かれたダイヤモンド形のついた直線は、**Circle** が **XXCircle** を保持していることを示しています。**Circle** オブジェクトは、実体化される時点で **XXCircle** オブジェクトを実体化します。そして、**Circle** オブジェクトに対する指示を、**XXCircle** オブジェクトに転送することになるわけです。つまり、**XXCircle** オブジェクトが **Circle** オブジェクトの機能的ニーズを満足している限り、**Circle** オブジェクトは **XXCircle** オブジェクトに仕事を任せることで、自らの振る舞いを実現できるようになるわけです（機能的ニーズを満足していないケースについては、この後で考察しています）。

コーディングについては例7.1を参照してください。

例 7.1 Javaによるコーディング例： Adapterパターンの実装

```
class Circle extends Shape {
  ...
  private XXCircle myXXCircle
  ...
  public Circle() {
    myXXCircle = new XXCircle();
  }

  void public display() {
    myXXCircle.displayIt();
  }
  ...
}
```

**達成したこと**

Adapterパターンを使用することで、***Shape***を用いたポリモーフィズムの恩恵を被り続けることができるようになります。言い換えれば、***Shape***のクライアントオブジェクトは、実際に存在している型を意識しなくても済むようになるわけです。これは、新たな観点に立ったカプセル化の一例です。つまり、***Shape***クラスは既存の具体的な形状をカプセル化しているのです。Adapterパターンは、ポリモーフィズムを実現するために多用されている一般的なパターンです。後の章でも見ていただきますが、他のデザインパターンでポリモーフィズムが必要となる場合にも、Adapterパターンが多用されます。

## 7.4 フィールドノート： Adapterパターン

**ラッパ以上のことができる**

上記で解説したような状況はしばしば発生しますが、適合対象となる既存クラスだけでは必要なことをすべて行えない場合もあります。



### Adapterパターン： 鍵となる特徴

| | |
|---|---|
| 目的 | 修正することのできない既存オブジェクトを、特定のインタフェースに適合させる。 |
| 問題 | 使用したいデータや振る舞いが既存システム内に存在しているものの、そのインタフェースが正しくない場合。通常は、抽象クラスから何らかの派生物を作成しなければならない場合に使用する。 |
| 解決策 | 必要なインタフェースを保持したラッパをAdapterによって提供する。 |
| 構成要素と協調要素 | **Adapter**は、その***Target***（派生元クラス）のインタフェースに合うよう、**Adaptee**のインタフェースを適合させる。これにより**Client**は、**Adaptee**をあたかも***Target***型であるかのように使用できるようになる。 |
| 因果関係 | Adapterパターンにより、既存オブジェクトをそのインタフェースに制限されることなく、新たなクラス構造に取り込むことが可能になる。 |
| 実装 | 他のクラスを用意し、既存クラスを保持させる。そして保持しているクラス側で、必要なインタフェースを提供し、既存クラスのメソッドを呼び出すようにする。 |

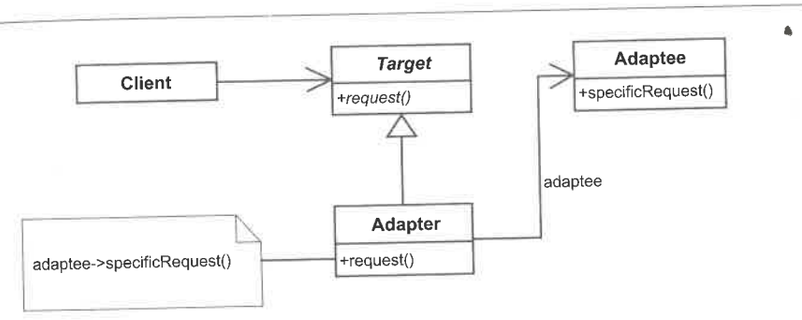


図 7.6 Adapterパターンの一般的構造

こういった場合でもAdapterパターンを利用することはできますが、それだけで終わるわけではありません。この場合、

- 既存クラス内に存在している機能は適合の対象となります。
- 既存クラス内に存在しない機能は、ラッパクラス内で実装することになります。

こうすることにより、純粋なAdapterパターンとまったく同じメリットを得られ

るわけではありませんが、少なくとも必要な機能すべてを自分で実装しなくても済むようになります。

**Adapterによって既存クラスのインタフェースによる制約から解放される**

Adapterパターンにより、既存クラスのインタフェースによる制約から解放された設計が可能になります。少なくとも概念的に必要なことを実現しているクラスがあれば、Adapterパターンを使用することで、常に必要なインタフェースに適合させることができるわけです。

さらにいくつかのパターンを学習すると、この重要性がより明確に理解できるようになるはずです。多くのパターンでは実装の際、関連クラス群を1つのクラスから派生させておく必要があるのです。このため、既存クラスがある場合、Adapterパターンを使用することで、（**Circle**によって**XXCircle**を***Shape***に適合させたように）既存クラスを適切な抽象クラスに適合させることができるようになるわけです。

**バリエーション：Object AdapterとClass Adapter**

実際のAdapterパターンには以下の2種類があります。

- **Object Adapterパターン**—上述したAdapterパターンは、あるオブジェクト（適合を行うオブジェクト）が他のオブジェクト（適合される既存オブジェクト）を保持しているため、Object Adapterパターンと呼ばれています。
- **Class Adapterパターン**—多重継承を使用してAdapterパターンを実装することもできます。これは、Class Adapterパターンと呼ばれています。

Class Adapterパターンを実装するには、以下のような新規クラスを作成します。

- 抽象クラスのインタフェースを定義するため、その抽象クラスから`public`で派生する。
- 既存クラスの実装にアクセスするため、その既存クラスから`private`で派生する。

新規クラス内におけるインタフェースの実装は、そのメソッドに対応付けられた`private`メソッド、つまり既存クラスから継承されたメソッドを呼び出します。

どちらのAdapterパターンが適しているのかという判断は、問題領域におけるフォースに依存します。概念レベルでは、こういった違いを無視しても構いませんが、実装段階になった時点で、どういったフォースが関与しているのかを考察する必要があります[*2]。

**AdapterとFacadeの比較**

私の実施しているデザインパターン講座では、AdapterパターンとFacadeパターンの違いが判らないという受講者が必ず1人はおりました。どちらのパターンも、既存クラス（またはクラス群）が存在し、そのインタフェースが望み通りになっていないのです。そしてどちらのパターンも、新規オブジェクトを生成することで望み通りのインタフェースを実現しているのです（図7.7を参照してください）。

**どちらもラッパ**

あなたも、**ラッパ**や**オブジェクトラッパ**という言葉を何度も目にしたことがあるはずです。レガシーシステム（従来からあるシステム）をオブジェクトで包み込み

[*2] Object AdapterとClass Adapterのいずれが適しているのかを考察するには、GoF本のp.142-144を参照してください。



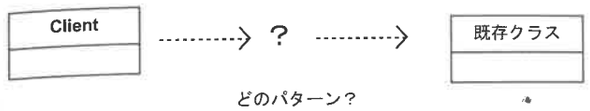


図7.7 インタフェースが望み通りでない既存オブジェクトを使用する

表7.1 FacadeパターンとAdapterパターンの比較

| | Facade | Adapter |
|---|---|---|
| 既存クラスがあるか？ | はい | はい |
| インタフェースを設計する必要があるか？ | いいえ | はい |
| ポリモーフィズムに則ったオブジェクトの振る舞いが必要になるか？ | いいえ | おそらくはい |
| より簡潔なインタフェースが必要か？ | はい | いいえ |

（ラップする）使用しやすくするという考え方は、珍しいものではありません。

こういったレベルから見た場合、FacadeパターンとAdapterパターンはよく似たものに見えるはずです。確かにこれらはどちらもラッパと言えます。しかし、ラッパの種類が違うのです。こういった違いは、些細なものなのですが、しっかりと理解しておく必要があります。違いを見極め、理解することによって、パターンが持っている属性についての深い洞察を得ることができるようになり、設計を考察したり、文書化する際、他の人に該当オブジェクト（群）の役割をきっちりと伝えられるようにもなるのです。では、こういったパターンにおけるさまざまなフォースを見てみることにしましょう（表7.1を参照してください）。

表7.1は以下のことを示しています。

- FacadeパターンとAdapterパターンは、いずれも既存クラスを使用する。
- しかし、Facadeパターンではインタフェースを設計し直す必要がなく、Adapterパターンではインタフェースを設計し直す必要がある。
- Facadeパターンではポリモーフィズムに則った振る舞いは不要であるが、Adapterパターンではおそらくそれを利用することがある（単に望み通りのインタフェースが欲しいというだけの理由で、Adapterパターンを使用することもある。この場合にはポリモーフィズムが不要となる。このため「おそらく」という表現になっている）。
- Facadeパターンの目的はインタフェースを簡素化することにある。Adapterパターンでもこういった目的が重視されるものの、本来の目的は既存インタフェースの再設計となっている。このため、インタフェースを簡素化することが可能であったとしても、それを行わない場合がある。

**これらの違いすべてがパターンの本質というわけではない**

Facadeパターンは複数のクラス群を隠蔽し、Adapterパターンは単一のクラスを隠蔽するという点に目を付ける人もいます。たいていの場合、これは状況を正しく言

い表しています。しかし、これはパターンの本質ではありません。Facadeパターンによって複雑な単一オブジェクトが隠蔽されることもあり得ますし、Adapterパターンによって必要な機能を実装している数多くの小さなオブジェクトが隠蔽されることもあり得るのです。

**まとめ：** Facadeパターンはインタフェースを**簡素化**し、Adapterパターンは既存のインタフェースを他のインタフェースに**変換**するのです。

## 7.5 CAD/CAMの問題とAdapterパターンの関係

Adapterパターンによって***OOGFeature***とやり取りできるようになる

CAD/CAMの問題（第3章「柔軟なコードを必要とする問題」）において、V2モデルの機能は***OOGFeature***オブジェクト群によって実現されることになります。しかし残念なことに、これらのオブジェクトは私が設計したものではないため、（私の観点から見た）望み通りのインタフェースになっていないのです。

こういったオブジェクト群のクラスをフィーチャークラスから派生させることはできません。しかし、V2システムは私の作業を完璧にこなしてくれるものなのです。

今回のケースでは、新規クラスを作成してこれらの機能を自ら実装するという選択肢は存在すらしておらず、必ず***OOGFeature***オブジェクト群とやり取りを行わなければならないのです。つまり、Adapterパターンを用いるのが最も適切であると言えるのです。

## 7.6 サマリ

この章のまとめ

この章のまとめ

Adapterパターンによって、クラス（またはクラス群）のインタフェースを、望み通りのインタフェースに変換することが可能になります。これには、望み通りのインタフェースを保持した新規クラスを生成し、既存クラスを包み込むことで、既存クラスのメソッドに対するラッパを実装することになります。

## 7.7 練習問題

### 7.7.1 基 礎

- □ 1. Adapteerを定義してください。
- □ 2. Adapterパターンの目的は何でしょうか？
- □ 3. Adapterパターンの因果関係とは何でしょうか？ 例を挙げてください。
- □ 4. ***Shape***と**Point**、**Line**、**Square**の間の関係を定義するために使用されているオブジェクト指向の概念とは何でしょうか？
- □ 5. Adapterパターンの最も一般的な使用方法とは何でしょうか？
- □ 6. Adapterパターンを採用することによって、どういったことを意識しなくてもよいようになるのでしょうか？
- □ 7. Adapterパターンは2種類に分類できます。それらを答えてください。



### 7.7.2 応 用

- □ 1. GoFはAdapterパターンの目的を「あるクラスのインタフェースを、クライアントが望むインタフェースに変換する」ものであると述べています。
  - この意味を説明してください。
  - 例を挙げてください。
- □ 2.「**Circle**オブジェクトは**XXCircle**オブジェクトのラッパです」という表現は、どういう意味を持っているのでしょうか？
- □ 3. Facadeパターンと Adapterパターンは、一見するとよく似ています。これら2つの本質的な違いを述べてください。
- □ 4. 以下はソフトウェアとは関係のない Adapterの例です： 国連でさまざまな国の外交官たちが集まり、自国の立場を自国の言語で討論していると考えてください。この時、通訳はある言語から他の言語への変換を行うことで、架空の外交官を「動的に」作り出し、受け手が聞きたい、そして聞く必要のあることを伝えています。
  - 実生活における Adapterの例をもう一つ挙げてください。

### 7.7.3 あなたの意見

- □ 1. Adapterパターンよりも Facadeパターンを使った方が適切となるのは、どういった場合でしょうか？ また、Facadeパターンよりも Adapterパターンの方が適切となるのは、どういった場合でしょうか？
- □ 2. GoFがこのパターンを Adapterと呼んだのは何故だと思いますか？ これは行っていることに合った適切な名前でしょうか？ その答えと理由を述べてください。