デザインパターン入門 レポート

平川 一樹

2018 年 4 月 5 日

概要

2018 年 3 月 30 日から作成開始。

デザインパターンの学習を目的にレポートを書いていく。3 日坊主にならないように気をつけよう^^

このレポートは「Java 言語で学ぶ　デザインパターン入門」（図 0.1 参考）を元に作っている。本を読んで理解した後、なるべく本を見ずに概要・例・私見・疑問点を書いて理解を確認する形式で進めていこうと思う。

1 日の終わりには感想をまとめておく。

図 0.1　参考文献

目次

1　Iterator パターン

1.1　概要

あるコレクションの要素に便利にアクセスするためのデザインパターン。

イテレータクラスを作り、コレクションを持つクラスがイテレータクラスを返すようにする事で、後はイテレータクラスのみを用いてコレクションにアクセスできる。

イテレータクラスとイテレータを返すクラスはインターフェースを用いて実装するべきメソッドを定めておく。

1.2　例

図 1.1 は Iterator パターンの一例だ。Aggregate、Iterator というインターフェースを作り、それらを実装する事で Iterator パターンを実現している。

ConcreteIterator は ConcreteAggregate が持っているコレクションなどにアクセスする。Next はコレクションの要素を返して、参照を次に移動するメソッド、hasNext は次の参照を持っているかどうか[*1]を判断して返すメソッドとなっている。

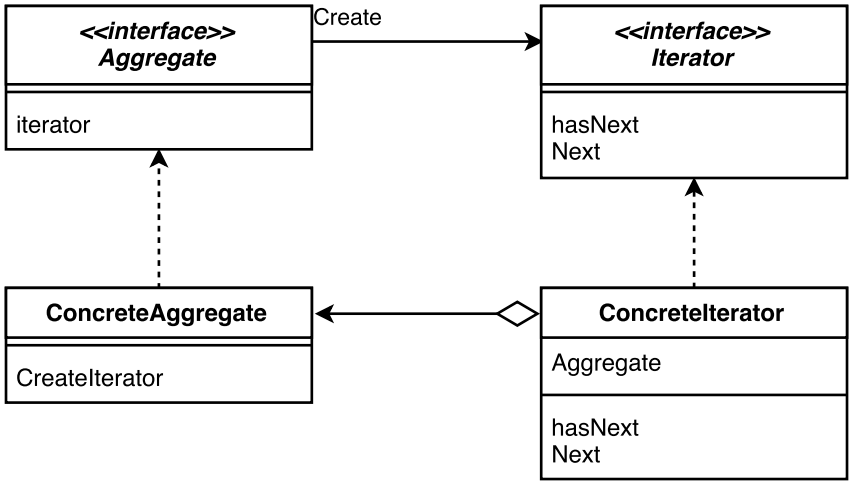

図 1.1　（例）Iterator パターン

1.3　私見

C++ の Iterator と同じようなものかと思ったが、それは今回のパターンとは少し実装の仕方が違っている気がした。特に C++ はインターフェースを使っておらず、Iterator クラスの定義が、Aggregate クラスの中に書かれていた覚えがある。C#の IEnumerable と IEnumerator がこのパターンを使っているのだろうと感じた。

[*1] コレクションの最後の参照かどうかとも言い換えられる

自分でこのパターンを使ったこともあったが、その時はインターフェースを作らずに直接 Iterator クラスを作ってしまった。その結果、Iterator クラスの内容はかなり独特なものになってしまったと思う。インターフェースを用いた方が、例でいう Concrete クラスの内容がどうであれインターフェースの様式[*2]のみを知っていれば扱えるようになるのがかなりのメリットだと感じた。

1.4 疑問点

特になし。

2018 年 3 月 30 日時点のコメント

感想

レポートを書こうと思い立って 1 日目が終了。

TEX の環境を構築したり、クラス図を作るための手段を調べるのに時間がかかってしまい、今回は 1 つのデザインパターンしか書けなかったが今後はもう少し早いペースで書いていけると思う。

Iterator パターンも読み始める前はわかっているつもりだったが、読んでみるとなかなか有益だったと思う。

[*2] 今回の例では hasNext、next のメソッドを知っているだけで使える

2　Adapter パターン

2.1　概要

あるクラスの振る舞いを他のクラスを用いて作り変えるようなイメージを持っている。

既に提供されているクラスを自分が必要としているクラスに作り変えて、提供されている機能と必要としている機能のズレを修正できる。

例えば、全言語対応の翻訳を行うクラスなどが提供されているとする。しかし、今必要なのは日本語と英語の翻訳機能だけだと言った場合に、このパターンを使ってシンプルな振る舞いのクラスに作り変えることができる。

出てくる用語としては以下の 3 つがある。

Target　現在の環境で必要な振る舞い
Adaptee　適合される対象
Adapter　適合させるクラス

このパターンの実現には以下の 2 通りの実現方法がある。

- 継承を用いる
- 委譲を用いる

2.1.1　継承を用いる実現方法

まずどのような機能が欲しいかを Target インターフェースに定めておき、Adaptee を継承し、Target を実装した Adapter クラスを作る。後は、

```
Target obj = new Adapter();
```

というイメージで、Adaptee の存在は無いものとして使用していく。

2.1.2　委譲を用いる実現方法

Adapter クラスに Adaptee クラスのデータを持たせる。Target クラスは Adapter を継承する。使う側は Target クラスを使用していく。

2.2　例

様々な動物の鳴き声音声を管理する AnimalVoice クラスが提供されていたとする。しかし、実際にはクイズアプリを作っていて、正解の場合に犬の鳴き声、間違いの場合は猫の鳴き声を取得したいものとする。

図 2.1 と図 2.2 はそれぞれ、継承・委譲による Adapter パターンで作成している。

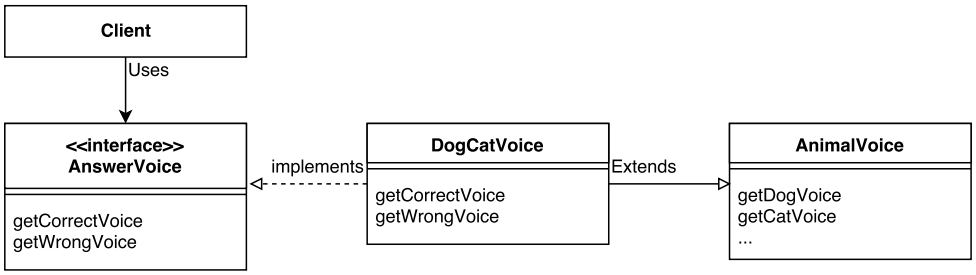

図 2.1　継承による Adapter パターン

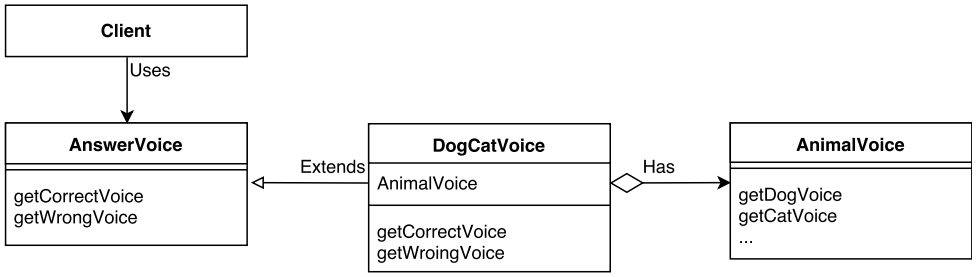

図 2.2　委譲による Adapter パターン

Target が AnswerVoice、Adpter が DogCatVoice[*1]と割り当てている。この方法で作成すれば、使用者 Client は AnimalVoice の仕様を理解していなくてもコードが書け、メソッド名も getCorrectVoice のように、現在のプロジェクトに適したメソッド名に置き換えることができる。

2.3　私見

Adapter パターンは今までよく使っていたが、今回学んだ設計方法とは少し異なっていた。特に、Target の概念を無視して作っていたと思う。今までは Adapter を直接使用するイメージだ。C 言語ライブラリを C++ でクラスのように扱うというような場合は今までの方法で良かったのかもしれないが、今後は Target の存在も意識しながら Adapter パターンを使っていこうと思う。

2.4　疑問点

2.4.1　Adapter の使用範囲

どのぐらいまで Adapter パターンを使った方が良いのかが疑問になっている。例えば、IOS の UIButton クラスなどを Adapter するのは少々やり過ぎなイメージがある。自分に分かりやすいメソッドをいくら定義しても、周りからは新しいクラスを覚える必要性が生じるので、見づらく感じてしまうのでは無いか?と思う。

しかし、マイナーなライブラリを使っている場合などに、今の環境に適したメソッド名をつけたりしながら、分かりやすいクラスに作り変えていく方法なら積極的に活用するべきだとも思う。

どのぐらいの範囲で Adapter パターンを使っていくのかが現在の疑問点だ[*2]。

[*1] 命名が少し不安 · · · 少し命名の仕方が違うような気がする

[*2] それを経験で養うのかもしれないが · · ·

2.4.2　Target と Adapter を分ける理由

Target をインターフェースなどで定めなくとも、Adapter クラスを使用すれば解決するのでは無いか？という疑問も少し浮かんだ。

節 2.2 の継承を使用した例で言うなら、後で仕様が変わった場合などに、Target インターフェースを実装した新たな Adapter クラスを作ることで、正解なら像、不正解ならライオンみたいな実装に容易に変えることができるからなのか？という感じで現在のところは理解している。

2018 年 3 月 30 日時点のコメント

感想

レポートを書こうと思い立って 2 日目が終了。

今回は、ソースツリーを用いて、このレポートの環境を GitHub のリポジトリにプッシュしてみることにして見た。ソースツリーの外観に慣れてみるのと、レポートを公開できる環境にしたかったからだ。レポートは PDF 形式なので、差分の恩恵は受けられないと思うが、TEX のコードは差分が見れるかもしれない。

勤務日だったのもあり、前回同様 1 つのデザインパターンしか書けなかったが、この調子で進めていこう。

3　Template Method パターン

3.1　概要

処理の大まかな流れを定めて、細かな処理は子クラスに任せるようなイメージを持っている。

親クラスでいくつか抽象メソッドを作っておき、その抽象メソッドを使って、別のメソッドの内容を定義する。その親クラスは抽象クラスとなるが、継承を用いて抽象メソッドの実装を行うことで、全てのメソッドが使用可能となる。

3.2　例

図 3.1 は TemplateMethod パターンの一例を考えたものだ。Products クラスは商品を扱うクラスで、商品の価格と数量から合計金額を getPrize メソッドで計算出来るようにする。

Product1〜3 のクラスは Products クラスを継承し、getUnitPrize という商品単価をもとめるメソッド、getNumber という商品の個数をもとめるメソッドをオーバーライドする[*1]。

このようにすることで、Product1〜3 の getPrize はそれぞれ適切な方法で単価と数量を取得し、合計金額を計算する getPrize メソッドを適切な方法で計算させることができる。

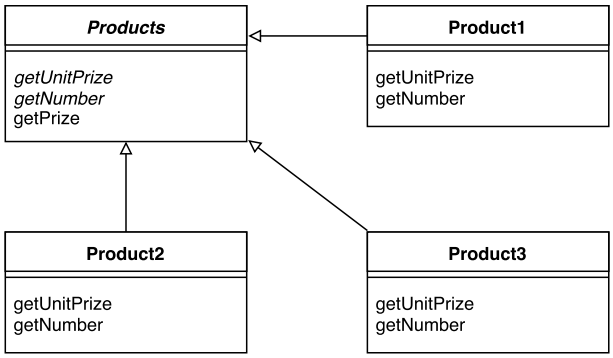

図 3.1　TemplateMethod パターンの一例

3.3　私見

このパターンは何回か使ったことがあったが、あまり積極的に使ったことは無かった。

複数のクラスに継承される親クラスを作る場合は常にこの考え方を念頭に入れるべきだと思った。「何か似ているクラスを作っているな〜」と感じた時にもこのパターンを視野に入れて作ってみたい。

3.4　疑問点

特になし

[*1] 実際にはメソッドよりプロパティにするべきかもしれない。このパターンにする必要性はあまり感じないかもしれないが、Product1 はネットから、Product2 は HD から、といったように情報の取得方法が違う場合を想定している。

2018 年 4 月 4 日時点のコメント

感想

最近忙しくてなかなか手がつけられていなかったが、今日はなんとか 1 章進めることができた。

FactoryMethod、Singleton など 2 章先ぐらいまでは読んでいるが、レポートの方が遅れ気味なので追いつきたい。