デザインパターン入門 レポート

平川 一樹

2018 年 4 月 20 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**??**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・考察・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。

この本は Java で書かれているが、自分でテストコードを作る場合は C#で書いていく。

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**??**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。

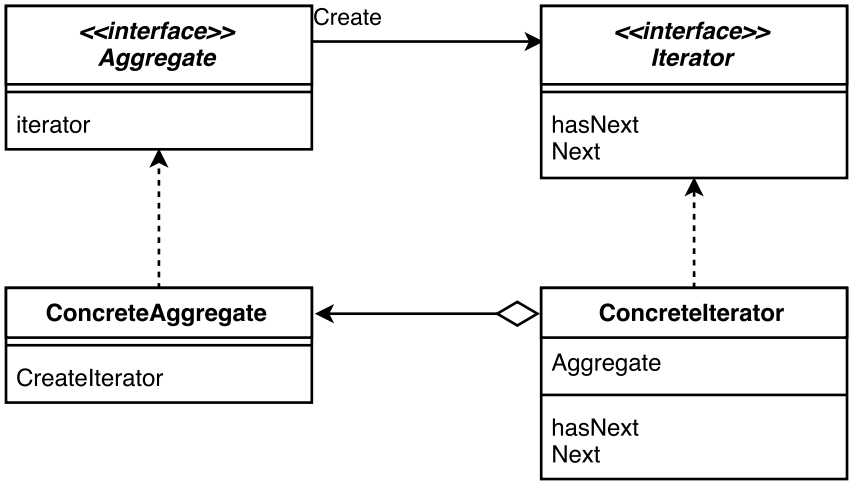

図 1.1　（例）Iterator パターン

1.3　考察

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、

```
1 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**??**と図**??**はそれぞれ、継承・委譲による Adapter パターンで作成している。

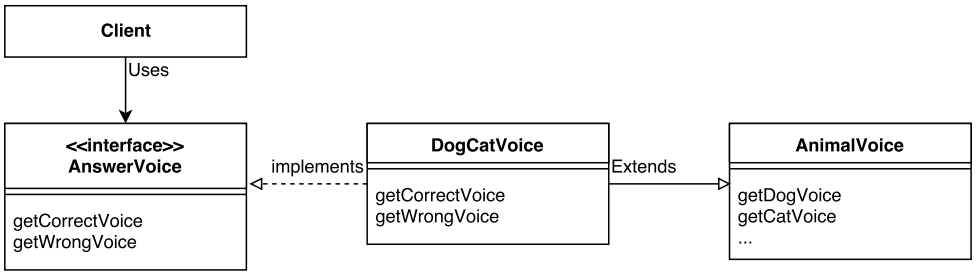

図 2.1　継承による Adapter パターン

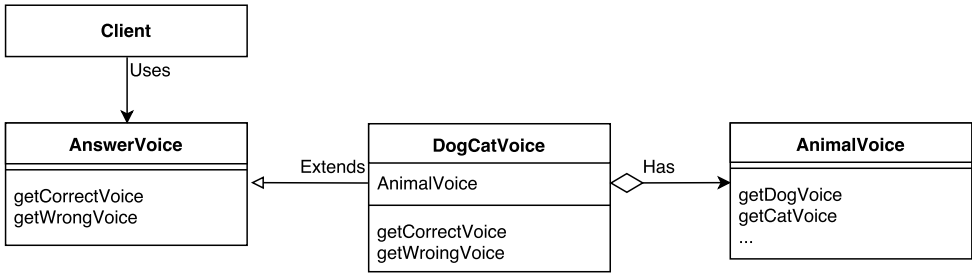

図 2.2　委譲による Adapter パターン

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　考察

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**??**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**??**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。

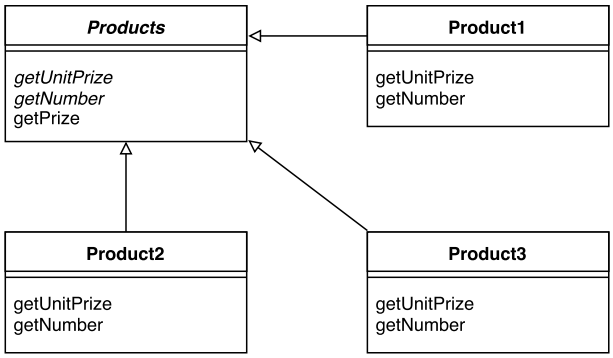

図 3.1　TemplateMethod パターンの一例

3.3　考察

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。

4　FactoryMethod パターン

4.1　概要

あらかじめ抽象クラスを用いて、作成者と製品を作る。

この際、作成者クラスには製品クラスを作成するためのメソッド（ここでは Create）を設ける。Create メソッド自体を抽象メソッド[*1]として子クラスに実装を任しても良いが、Create メソッドの中身が複雑な場合に TemplateMethod パターンを用いて、子クラスに実装を任すことで、インスタンスの作成方法を一般化することができる。

製品クラスは抽象クラスとし、製品の振る舞いを抽象メソッドで定めておく。

あとは、この 2 つのクラスを継承し、具体的な作成者クラス、具体的な製品クラスを作っていく。

4.2　例

シューティングゲームなどに用いられる弾やレーザーなどのオブジェクトを管理する際などに、このパターンを用いることを例に挙げていく。弾の作成にはルールがあり、1 秒以内に複数の弾は作成できないものとする。つまり、一度弾を作ると 1 秒間新しい弾は作れないものとする。ただ、レーザーについては 3 秒間に 1 回とする。

クラス名については以下のように定める[*2]。コードは図**??**の様になった。

Factory　作成者クラス
Product　製品クラス
BulletFactory　弾の作成者クラス
Bullet　弾の製品クラス
LaserFactory　レーザーの作成者クラス
Laser　レーザーの製品クラス

このように作ることで、何秒間に 1 回までしか作成できないという決まりごとを定めることができる。また新しく爆弾オブジェクトを作ることになった場合でも、時間確認の処理の実装を忘れることはない。

4.3　考察

このデザインパターンはまだ使ったことがなかった。TemplateMethod パターンに似ているが、作成方法の枠組みを定めているところが前回との違いだと思う。

インスタンスの作成方法から振る舞いまで酷似しているようなクラスを設計する場合には、このパターンを使うことになるのかもしれない。

[*1] 抽象ではなく仮装メソッドとして、デフォルトの処理を書いておいても良い。

[*2] もう少し良い命名があったとは思うけど...

```
abstract class Factory{
	private Time beforeTime;

	public Product create(Point pt){
		Time now = Now();
		bool judge = judgeCreate(now - beforeTime);
		beforeTime = now;

		if(judge){
			return createProduct(pt);
		}else{
			return null;
		}
	}

	public abstract Product createProduct(Point pt);
	public abstract bool judgeCreate(Time tm);
}

class BulletFactory : Factory{
	public override Product createProduct(Point pt)(
		return new Bullet(Point pt);
	}

	public override bool judgeCreate(Time tm){
		return tm.toSecond() >= 1.0;
	}
}

class LaserFactory : Factory{
	public override Product createProduct(Point pt){
		return new Laser(Point pt);
	}

	public override bool judgeCreate(Time tm){
		return tm.toSecond() >= 3.0;
	}
}

abstract class Product{
	...
}

class Bullet : Product{
	public Point centerPoint;
	public Bullet(Point pt){
		centerPoint = pt;
	}

	...
}

class Laser : Product{
	...
}
```

図 4.1　FactoryMethod パターンの一例

4.4　疑問点

4.4.1　製品クラスの設計

今回の例では、内容について省略させてもらったが、製品クラスに振る舞い（つまりメソッド）のみしか記入しないのであれば、インターフェイスを使うこともできるのではないか、という疑問が残った。

製品クラスを継承するような事例がないのでどちらでもいいのかもしれない。もしくは製品クラス

に変数を持たせる必要が出てくる場合もあるのかもしれない。

5　Singleton パターン

5.1　概要

プログラム上であるクラスのインスタンスが必ず 1 つしか存在しないことを保証するためのデザインパターン。

クラスのフィールドに自分自身を保持する静的メンバを持ち、メソッドでその変数を返せるようにする。変数、コンストラクタは private とし、クラス外から新しいインスタンスは作らせないようにする。

5.2　例

図**??**はメソッドを呼ぶたびに、1 · 2 · 3 · · · と数値を返す例だ。

```
1  class CountNumber{
2        private num = 0;
3
4        private static CountNumber singleton = new CountNumber();
5
6        private CountNumber() { }
7
8        public CountNumber Singleton{
9              get{
10                   return singleton;
11             }
12       }
13
14       public getCount(){
15             return ++num;
16       }
17 }
```

図 5.1　Singleton パターンの一例

5.3　考察

このパターンについては聞いたことがあるだけのレベルで、実際どんな時に使えば良いかは考えたことがなかった。

しかし、前章の FactoryMethod パターンの製作者クラスを Singleton パターンで実装することは有効なのではないかと思った。

5.4　疑問点

5.4.1　静的クラスとの使い分け

Singleton パターンと静的クラスは非常によく似ていると思う。

静的クラスは継承出来ないので、FactoryMethod パターンなどに応用する場合は、Singleton パ

ターンを使うべきだと思う。

しかし、使い分けの基準が継承できるかどうかだけなのかはまだ疑問な部分。Singleton は引数に出来る... という理由も考えたが、そもそも一つしかインスタンスはないのだから引数にする理由もあまりない気がしている。

2018 年 4 月 6 日時点のコメント

感想

今日は FactoryMethod パターン、Singleton パターンの 2 章を書いた。

FactoryMethod はなかなか複雑で例を考えるのに苦労したが、慣れればとても有用なパターンだと思った。

全 23 章なので 20% ほど終わったところ。次は Prototype パターン。なにやらインスタンスを作成する方法らしいので、FactoryMethod と同じような分類なのか。今後もこの調子で進めていこう。

6　Prototype パターン

6.1　概要

Prototype パターンは Product インターフェースのインスタンスを登録できる Manager クラスを作り、その Manager クラスが保持しているインスタンスを複製することで、目的のインスタンスを取得していくパターンだ。

- クラスの種類が多すぎる場合
- クラスからのインスタンス生成が難しい場合
- フレームワークと生成するインスタンスを分けたい場合

などの理由がある場合に使用する。

6.2　例

指定されたパスの音声を再生する処理を例を考えてみる。コードは図**??**のようになった。

今回の例はこのパターンを使う必要性があまりないかもしれないが、小中大の 3 つの音量で再生する SoundPlayer クラスのインスタンスを Manager 登録しておくことで、再生する場合は Manager からインスタンスをクローンすることで解決できる。

今回 Product を実装しているクラスは SoundPlayer のみだが、画像や動画にも対応させたければ新たに Product を実装しているクラスを作れば良い。

6.3　考察

なんとなく FactoryMethod パターンに似たものを感じた。FactoryMethod はクラス設計の大枠を指定してあげるようなものだが、Prototype はこれを使用者側目線でも大枠を指定してあげるようなもの？[*1] とも感じた。

少しメンバーをいじるだけで挙動が大きく変わるようなクラスを Manager に複数登録しておき、後で簡単にインスタンスを作成できるようにする役割もあると思う。

6.4　疑問点

6.4.1　クラスからのインスタンス生成が難しい?

このパターンを使用する場合の説明で、クラスからのインスタンス生成が難しい場合とあるが、この点がまだいまいちよく分かっていない。まだ実例が思い浮かばないからかもしれない。

[*1] うまく説明できない><

```
1  ////////////////////////////////////////////////////////////
2  //Package.css
3  public interface Product : ICloneable {
4      public abstract void play(string Path);
5      public abstract Product createClone();
6  }
7
8  public class Manager {
9      private Dictionary<string, Product> showCase;
10
11     public Manager() {
12         showCase = new Dictionary<string, string>()
13     }
14
15     public void register(string name, Product proto) {
16         showCase.Add(name, proto);
17     }
18
19     public Product create(String protoname){
20         return (Product)showCase[protoname];
21     }
22 }
23
24 ////////////////////////////////////////////////////////////
25 //Main.css
26 public SoundPlayer : Product{
27     private double sizeOfSound;
28
29     public SoundPlayer(double size){
30         sizeOfSound = size; //音量指定
31     }
32
33     public void play(string Path) {
34         //Pathの音声をsizeOfSoundの音量で再生
35     }
36
37     public Product createClone() {
38         try {
39             return (Product)clone();
40         }catch{
41             //エラー処理
42         }
43     }
44 }
45
46 public class MainClass{
47     public static void Main(string[] args){
48         //準備
49         Manager manager = new Manager();
50
51         manager.register("Small", new SoundPlayer(0.2));
52         manager.register("Normal", new SoundPlayer(0.5));
53         manager.register("Big", new SoundPlayer(0.8));
54
55         //生成
56         var p1 = manager.create("Small");
57         var p2 = manager.create("Normal");
58         var p3 = manager.create("Big");
59
60         p1.play("Test/Sound1.mp3"); //小さい音量で再生
61         p2.play("Test/Sound2.mp3"); //普通の音量で再生
62         p3.play("Test/Sound3.mp3"); //大きな音量で再生
63     }
64 }
```

図 6.1　Prototype パターンの一例

6.4.2　クローンの意味

初めこのパターンを見たときは、初期化に多くの引数が必要になるような複雑なクラスがあった時、いちいち引数を入れるのは面倒なので一旦どこかにインスタンスを保存しておくようなもの。というイメージを持ったが、そのような考え方は本には書いていない。

しかし、その目的なら Clone 関数などを使う理由が無くなってしまう。今回の例では Manager クラスにインスタンスを保存したが、別にクローンを使わなくても、new で作ったインスタンスを登録すれば目的は満たされる。フレームワークと生成するインスタンスを分けたい場合という意味なら理解はできるが...

... というより今回の自分の例、本に載っている例では、結局 Manager に登録したインスタンスのメソッドを使うだけで終わりなので、create でインスタンスを複製しなくても参照を返すだけで解決してしまう。

まだ Prototype だからこそスッキリする！というような例が見つからない。

2018 年 4 月 20 日時点のコメント

感想

仕事の出張とその準備もあり、なかなか進められていなかったが、今日は Prototype パターンを書いた。

なかなかこのパターンを使った実例が思いつかず、メリットについてあまり理解できていないかもしれない。もう少し実例を考えてみよう。

次は Builder パターン、なかなか複雑そうな名前だが何かを生成するパターンなのだろう... 多分。