Java を用いたデザインパターンコーディングの必要性

文理学部情報科学科
5419045 高林 秀

2021 年 11 月 12 日

概要

本稿では、今年度発展プログラミングの課題研究として「デザインパターン」に準拠したコーディングスタイルの必要性について、実際に自身でコーディングを行い、論ずる。本演習には Java を利用した。

1　目的

本稿は今年度発展プログラミングの課題研究として、「デザインパターン」に準拠したコーディングスタイルの必要性について論ずることを目的とする。本稿前半では、デザインパターンの利点・効果・必要性などについて説明し、後半では、実際にコーディングを行う。その際、デザインパターンを使用する前のコードとデザインパターンを使用した際のコードを比較・考察していく。

2　事前知識

本稿では Java を使用する。クラス定義の仕方や、インタフェースなどの Java で登場する用語についての説明は省略する。これらの解説は以下の URL からレポート「interface、抽象クラスを利用した Java のペア・プログラミング」を参考いただきたい。

- url:`https://drive.google.com/drive/folders/1QEt-NBptDGq2J1BgOyFnGUx8SMwL5oNc?usp=sharing`

3　デザインパターンとはなにか

この章では、プログラムコーディングおけるデザインパターンとはどのようなものなのか、歴史的背景や利点・効果、その必要性について述べる。

3.1　歴史的背景

デザインパターンは、オブジェクト指向型プログラミング言語において、記述したコードを様々なプログラムで再利用できるようにするために考案された「プログラムの設計ルール」のようなものである。1955 年に出版された書籍「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 [1]」にて、初めて「デザインパターン」と呼ばれる用語が使用された。その書籍の広がりにより、デザインパターンの考え方が広く知られる

ようになった。

その書籍の著者ら（※参考文献の原著 4 名）は、23 種にもおよぶデザインパターンを取り上げており、デザインパターンとはなにか、以下のように述べている。以下、https://ja.wikipedia.org/wiki/%E3%83%87%E3%82%B6%E3%82%A4%E3%83%B3%E3%83%91%E3%82%BF%E3%83%BC%E3%83%B3_(%E3%82%BD%E3%83%95%E3%83%88%E3%82%A6%E3%82%A7%E3%82%A2) より引用する。

- 原文

  [Design patterns] solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

- 訳文 (Deepl[*1]による翻訳)

  特定の設計上の問題を解決し、オブジェクト指向設計をより柔軟に、エレガントに、そして最終的には再利用可能にします。デザインパターンは、新しいデザインを過去の経験に基づいて行うことで、成功したデザインを再利用するのに役立ちます。このようなパターンに精通している設計者は、パターンを再発見することなく、すぐに設計問題に適用することができます。

この書籍の著者たちは「Gof（Gang of Four)」と呼ばれデザインパターンの名前の 1 つにもなっている。

余談だが、Gof のデザインパターンはかなり前のデザインパターンである。そのため、現代では賛否両論あるようで、一部では批判的な意見も挙げられている。というのも、後に説明するが、デザインパターンは Java や Ruby などオブジェクト指向型言語で使用される考え方だ。しかし Gof のデザインパターンが発表されたのは Java がリリースされる 1995 年よりも前、つまりオブジェクト指向プログラミングというものが未熟な時代に発表されたからだ。したがって、現代のプログラミングと合致しない場合もあるという。この意見は https://qiita.com/irxground/items/d1f9cc447bafa8db2388 より参考にさせていただいた。

3.2 デザインパターンの利点

前述した通り、デザインパターンの主目的は「コードを様々なプログラムで再利用できるようにする」ことである。ただし、利用するデザインパターンにより、その効果は少し異なる。今回扱う、Gof のデザインパターンは大きく分けて、「生成」、「構造」,「振る舞い」の 3 つのカテゴリに分類され、その総数は後述するように計 23 パターンにおよぶ。

以上のように分類され、体系が出来上がった概念、考え方なので巷では初心者エンジニアが、オブジェクト指向を学習するときの題材として利用しやすいことも 1 つの利点と言えるだろう。

さて、デザインパターンの大きな利点として、先に上げた「コードの再利用」の他に以下の 3 つが挙げられるだろう。

- 可読性の向上
- 保守性の向上
- 設計の短時間化

[*1] ドイツに拠点を置く DeepL GmbH によって開発された、ニューラルネットワークによる翻訳を行うサービス。Google 翻訳よりも精度が高く、微妙なニュアンスのある翻訳ができると話題。

■可読性の向上　まず、「可読性の向上」だがこれは自分が作成したプログラムを、他のエンジニアやプログラマーが閲覧するときに、そのプログラム、クラスがどんな役割を果たしているのか、どのような意図のコードなのかが把握しやすくなるという意味だ。先に示したように、デザインパターンはすでに確立されたプログラムの設計概念である。したがって、いわば共通言語のようなものであるので、デザインパターンを意識したプログラムは他人からも読みやすいということになる。

■保守性の向上　次に、「保守性の向上」だがこれは、例えば呼び出し元プログラムの実装方法が変更になった場合でも呼び出し側のプログラムに修正を加えることなく実行できるということだ。大規模な開発になると、実装元と呼び出し側にコードを分散させることが多々あるが、仕様変更やアップデート等で実装側のコードを改変した場合、デザインパターンを意識しないで呼び出し側を設計すると、実装元と呼び出し側両方のプログラムを修正しなければならない。この様なプログラムは、実装側の修正箇所が多くなるたびに呼び出し側も修正しなければならないため非常に効率が悪い。加えて、実装側のコードや動作を把握しなければ呼び出し側の設計ができないという欠点も生じる。

デザインパターンを導入すれば、少ない修正でプログラムを動かすことが可能になる。

■設計の短時間化　最後に「設計の短時間化」であるが、これはコードの再利用の点と一部重なるかもしれない。デザインパターンは、Gof のデザインパターンの著者を初め、すでに誰かが考え出した設計にしたがってプログラムの設計を行う。したがって、自身で保守性や再利用性を向上させようとあれこれ試行錯誤する必要はなく、コーディング作業全体の効率化を図ることができる。

3.2.1　デザインパターンの必要性

ここまで、デザインパターンの利点を述べてきたが、ここではデザインパターンの利用が推奨されるケースについて説明する。

現代でプログラミングをする際、すでにこの世に存在している機能をもう一度初めから開発する手間を省くため「ライブラリ」や「フレームワーク」と呼ばれるものが存在する。「ライブラリ」とは、すでに誰かが開発した機能や技術を、他人のプログラム内に組み込めるようにしたものである。有名どころをいくつか列挙すると、Python ならば、プログラムからウェブブラウザを操作できる機能を提供する「selenium」や、数値計算を高速化、容易にする「numpy」,「pandas」などがあるだろう。Java ならば、web サーバーを構築する際に使用される「Apache HTTP Server」や、HTML を解析する際に使用される「Jsoup」など多種多様なものが存在する。

これらのライブラリやフレームワークが提供する機能を、指定されるコードを用いて、自身の目的に合わせて作成するプログラムに組み込むことができる。

つまり、ライブラリやフレームワークを利用するときは、あらかじめ用意されているプログラムを再利用することになる。また、ライブラリやフレームワークが提供する関数やメソッドの処理内容は原則変更することができない。

例えば外部に公開されているライブラリを利用し、自身のプログラムを記述する場面を考える。ライブラリから提供される関数やメソッドを呼び出しているのが原因で、自身の意図する目的と異なる動作をすることがある。しかし、ライブラリ側のコードを自身で書き換えると利用している他の関数やメソッドの動作に影響するため、安易に改変することはできない、とする。この様な場合、ライブラリ側のプログラムと、自身で作成したプログラムの動作の差異を吸収するプログラムが必要になってくる。この様な場合は、後に紹介している

「Adapter パターン」と呼ばれるデザインパターンを利用することで解決できる場合がある。

当然、上で示した各利点を目的とする場合は、それぞれのケースに適したデザインパターンを利用することが推奨される。

3.3 デザインパターンの種類

先に示したように、デザインパターンは合計 23 種類ものパターンが存在する。まず「生成」、「構造」,「振る舞い」の 3 つのカテゴリそれぞれに該当するデザインパターンについて軽く説明を行う。加えて、最後に本稿の演習で使用するデザインパターンである Bridge パターンについて詳細に取り上げる。

3.3.1 オブジェクト生成に関するパターン

オブジェクトの生成を以下のパターンによってコントロールすることができる。状況に応じてオブジェクトの生成をコントロールすることにより、アプリケーション全体の設計に悪影響を与えることを防ぐことができる。以下は、生成に関するパターンに属するデザインパターンである。

- Abstract Factory
  - インスタンスの生成をコントロールすることを専門に扱うデザインパターンである。プログラム内に整合性が必要とされるとき、関連するクラスのインスタンスをミスなく生成する必要があるときに利用される。Factory クラスと呼ばれるクラスを切り替えることで、利用するインスタンスを切り替えることができる。
- Builder
  - 主に、Burider クラスと Director クラスで構成されるデザインパターンである。Builder クラスはいわゆる「表現形式」を定めた抽象クラスで、Director クラスは「処理課程」を定めた実装クラスである。同様な処理でも、異なる形式の結果を得たい場合に利用する。オブジェクトの生成を容易にしかつ、その生成もコントロールすることを目的としている。
- Factory Method
  - インスタンスの作成手順を親クラスで定義し、その具体処理を継承先の子クラスで記述するデザインパターンである。生成と、処理内容を分けて記述することにより、より柔軟にオブジェクトの生成を行うことができる。
- Prototype
  - 日本語では「試作品」「実験版」「原型」という意味を持つ。あらかじめ準備してある「原型」をから、インスタンスを生成するようにするデザインパターンである。例えば、同じ様なオブジェクトが複数欲しいとき、最初に生成したインスタンスをコピーするという方法を行うことで効率よくオブジェクトを得ることができる。つまり、Prototype パターンは、インスタンスからインスタンスを生成する様なデザインパターンと言える。
- Singleton
  - 生成されるインスタンスを 1 つに制限することができるデザインパターンである。このようにすることで、インスタンスの状態を保持することができたり、プログラム内のインスタンスが 1 つであることを保証できるようにすることができるので、安全性という意味で向上が期待できる。Singleton パターンは、コンストラクタに修飾子 private を付与し、他のクラスからアクセスでき

ないようにすることで、他のクラス内でインスタンスを生成することがないようにしている。

3.3.2 プログラム構造に関するパターン

オブジェクト同士の関係性を用意にする設計を提供するデザインパターンである。以下のものが構造に関するデザインパターンに該当する。

- Adapter
  - 本来関係のないクラス同士を関連付けることができるデザインパターンである。先に示したように、提供されているコードと、作成コードの差異を埋めるようなデザインパターンである。
- Bridge
  - 親クラスに抽象メソッドが定義されている場合、そのメソッドの実装と、実装とは関係ない新機能の記載を分けるようなデザインパターンである。これに関しては、本演習で使用するので、後ほど改めて説明する。
- Composite
  - 日本語では「複合物」を意味する。全体を囲む「容器」とその「中身」を同じものとして捉え、共通機能を持たせる際に利用する（ファイルシステムなど）。オブジェクトの再帰的な取扱を容易にすることができる。より、具体的にいうと、オブジェクトの集合（グループ）を、オブジェクトの１つのインスタンスとして取り扱うことである。
- Decorator
  - 同じ様な機能の追加を行うとき、クラスの継承以外の手法で既存のクラスを拡張する際に用いられる。
- Facade
  - 既存のクラスを複数組み合わせて使うような処理があるとき、複雑さを回避するため、それらのクラスの「窓口や受付」の役割を果たすようなクラスを作成することで構造をシンプルにしようとするデザインパターンである。
- Flyweight
  - このデザインパターンは計算機のリソース利用の効率化に焦点をあてたデザインパターンである。具体的には、同一のインスタンスを共有したりすることで、無駄なインスタンスの生成を防ぐことができ、プログラムのメモリ使用量などを軽くすることができる。
- Proxy
  - 日本語では「代理人」を意味する。Proxy パターンは、元のオブジェクトの処理を、別のオブジェクトが肩代わりするようなデザインパターンである。元オブジェクトが行う必要がない処理を代理する別オブジェクトを作成することで、処理の効率化を図ることができる。

3.3.3 オブジェクトの振る舞いに関するパターン

オブジェクト間の通信を容易に行えるようにするデザインパターンである。例えば、異なるクラスでデータのやり取りを行いたい場合などが該当する。以下のものが振る舞いに関するデザインパターンに該当する。

- Chain of Responsibility
  - 日本語で、「責任の連鎖」を意味する。つまり、オブジェクト間を鎖で接続し、その鎖をたどって、

要求される処理ができるオブジェクトか否かを探す。メリットとして、要求を出すクラスと要求を処理するクラスが分離しているので「クラス間の独立性を確保している」という点が挙げられる。反対にデメリットとしては、鎖を渡り歩てオブジェクトを捜索するので、「処理が遅くなる」という点が挙げられる。

- Command
  - 処理の命令を記載するクラスのインスタンスを 1 つのオブジェクトとして表現するデザインパターンである。例えば、命令の履歴を管理する際、インスタンスの集合を管理すれば良いことになるので、再度同じ命令を実行するときは、インスタンスを再利用することで実行できる。加えて、 1 つの引数のみで複数のデータを扱うことができる。
- Interpreter
  - 任意の形式で書かれたファイルの内容を、いわゆる「通訳」を行うプログラムを介して解析するデザインパターンである。主な使用用途として、interface を用いて構文解析の結果を階層構造で示すことができる。
- Iterator
  - 日本語では、「繰り返し」や「反復」といった意味を持つ。オブジェクトの集合を列挙する方法を提供するデザインパターンである。メリットとしては、オブジェクト間の関係性をシンプルにすることができることが挙げられる。配列など、要素が集合しているオブジェクトにアクセスする場合は特にこのデザインパターンに準拠したコードを記述することが推奨される。
- Mediator
  - 日本語では、「仲介者」や「まとめ役」といった意味を持つ。オブジェクト同士が互いにアクセスを行わないようにする仕組みを提供するデザインパターンである。これによって、オブジェクト間の関連性の強さをさげることができ、オブジェクトの独立性を保つことができる。
- Memento
  - 日本語では、「記念品」や「形見」といった意味を持つ。インスタンスの保存と復元を行う場合、インスタンスのもつ情報に無制限にアクセスする必要が生じる。しかしそれでは private などの修飾子が使用できない上、カプセル化の意味が無くなってしまう。Memento パターンは、インスタンスの状態を示す機能を導入することで、カプセル化を無効化することなく、インスタンスの復元と保存を行うことができる。
- Observer
  - 日本語では、「監督者」といった意味を持つ。「Subject」と呼ばれる監督される側オブジェクトと、「Observer」と呼ばれる Subject の状態を監督する側のオブジェクトに分けられる。Subject の状態が変化した際に、Observer に通知される仕組みを持つ。別名 Publish-Subscribe パターンと呼ばれることもある。
- State
  - オブジェクトではなく、状態をクラスとして捉えるデザインパターンである。その時点での状態によってプログラムの振る舞いを変化させるときに適したデザインパターンとなっている。
- Strategy
  - 日本語では、「戦略」といった意味を持つ。アルゴリズムの実装部分を切り替えることができるデザインパターンである。アルゴリズムの部分を別クラスとして作成し、別のアルゴリズムに切り替えたい場合は、そのクラスを変更してアルゴリズムを切り替えることができる。したがって、メ

ソッド中にアルゴリズムを記載する方法よりも、より柔軟にメンテナンスを行うことができるので、コードの保守性の観点から良いと言えるだろう。

- Template Method
  - 親クラスで、処理の大枠を決め、具体的な処理内容を継承先の子クラスで決めるデザインパターンである。異なる振る舞いであっても、処理的には似ているとき、それぞれ個別に記載するのではなく、共通化することができる。また、処理の大枠は継承元の親クラスで実装しているので、その部分にバグがあったとしても、継承先の子クラスすべてを修正する必要はない。また、メソッドの機能を部分的に変更することも容易となる。
- Visitor
  - 日本語では、「訪問者」といった意味を持つ。Visitor パターンは「データ構造とそれを処理する部分を分離する」デザインパターンである。訪問者クラスと呼ばれる、データ構造を淘汰するクラスを作成し、各データの処理はその訪問者クラスが請け負うという構造である。したがって、新たな処理を追加したい場合は、その都度新たな訪問者クラスを作成しさえすれば良い。利用シーンとして、オブジェクトに動的に機能を追加する際に用いられる。

3.4 Bridge パターン

本稿後半で行う演習では、デザインパターンとして Bridge パターンを採用している。ここでは Bridge パターンの詳細な説明を行う。

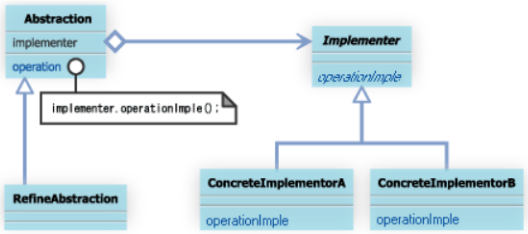
Bridge とは日本語で「橋」を意味する。その意味の通り、Bridge パターンは目的が異なる 2 つのクラスを結びつける橋渡しの役割をしている。そのクラスとは、「機能のクラス階層」と「実装のクラス階層」と呼ばれるものである。

■機能のクラス階層　親クラスで基本的なメソッド（機能）を決定し、継承先の子クラスで新たな機能を追加するクラスの階層のことを「機能のクラス階層」と呼ぶ。

■実装のクラス階層　親クラスの抽象メソッドにより、interface を決定する。継承先の子クラスにて、メソッドを具体化しその interface を実装するクラス階層のことを「実装のクラス階層」と呼ぶ。

■想定場面　例えば、Bridge パターンを使用しない場合について考える。親クラス内で、そのオブジェクトの基本機能である抽象メソッドを定義し、継承先の子クラスで抽象メソッドを実装したとする。後から、そのオブジェクトに新たな機能を持たせたいと思い、そのまま子クラスに新たなメソッドを定義したとしよう。最初に行った継承によって作成された子クラスは、オブジェクトの基本機能を定めた「抽象メソッドの実装」が目的である。しかし、次に行った新たなメソッドを定義によって、この子クラスの目的が、「基本機能の実装」と「全く新しい新機能の追加」という複数の目的を持ったクラスになってしまい、一貫性が損なわれてしまう。加えて、抽象メソッドは継承先の子クラスで必ず実装しなければならない性質を持つ。もし、新機能を定義したクラスを作成した場合、基本機能の実装の抽象メソッドを定めている親クラスを継承しているので、そのクラスの数が数個ならばあまり手間はかからないかもしれないが、数十個、数百個となった場合、それぞれ同じ内容の抽象メソッドの実装をしなければならない。このような問題を避けるために Bridge パターンは利用される。

■Bridge パターンまとめ　まとめると、Bridge パターンは元からある基本機能を実装するクラスの拡張と、新たな機能を付加するためのクラスの拡張をそれぞれ分離することにより、前述した問題を回避し、クラスの拡張を容易に行えるようにするためのデザインパターンであると言える。



[引用]『Java言語で学ぶ デザインパターン入門』(結城浩 ソフトバンクパブリッシング株式会社出版 2001年)

図 1　Bridge パターンのクラス図（UML）

※引用元：https://www.techscore.com/tech/DesignPattern/Bridge.html/

4　実際のコーディングでデザインパターンを利用する

4.1　課題説明

■問題　任意のデザインパターンを 1 つ選択し、そのパターンを用いる前と用いた後のコード両方を作成して示し、そのパターンを用いたことの効果を説明せよ。

- 選択したデザインパターン：Bridge パターン

4.1.1　演習環境

今回の演習は仮想マシン上で Java を使用し行った。下記に演習時の環境を示す。

- ホスト OS：Window10 Home 20H2
- 仮想 OS：Ubuntu 20.04.2 LTS
- CPU：Intel(R)Core(TM)i7-9700K @ 3.6GHz
- GPU：Nvidia Geforce RTX2070 OC @ 8GB
- ホスト RAM：16GB
- 仮想 RAM：4GB
- 使用言語：Java
  - バージョン情報は下記に示す。

```
openjdk version "11.0.11" 2021-04-20
OpenJDK Runtime Environment (build 11.0.11+9-Ubuntu-0ubuntu2.20.04)
OpenJDK 64-Bit Server VM (build 11.0.11+9-Ubuntu-0ubuntu2.20.04, mixed mode, sharing)
```

参考文献

[1] Erich Gamma (原著), Ralph Johnson (原著), Richard Helm (原著), John Vlissides (原著), 本位田 真一 (翻訳), 吉田 和樹 (翻訳)「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」改訂版（ソフトバンククリエイティブ,1999/10/1)