第 1 回 デザインパターンゼミ

ゼミ担当者　：　西岡 雅史，青木 大，木田 清香，木浦 正博
指導院生　：　細江 則彰，天白 進也
開催日　：　2006 年 5 月 22 日

ゼミ内容: 本ゼミでは、設計上の問題に対しての解決方法であるデザインパターンについて学ぶ.

1　はじめに

オブジェクト指向プログラミングにおいて，作成されたプログラムを効率的に再利用するためにデザインパターンという手法がある．本ゼミでは，数あるデザインパターンの中から特によく知られている GoF の 23 のデザインパターンのうち 4 つについて解説する．また，別紙に載せたプログラムを実際に書いて実行することでデザインパターンに慣れていただきたい.

2　デザインパターンとは

デザインパターンはクラス設計の場面で頻繁に表れる具体的な「設計のコツ」をパターンとして捕らえ，それに名前をつけて特徴を整理したものである．したがって，デザインパターンは具体的な設計の詳細ではなく，それらの「考え方」や「アイデア」を抽象化して，さまざまな局面で使いやすいように整理したものである．オブジェクト指向との親和性から Java や C++などのプログラミング分野で広く用いられている．そのため，設計のコツを示すだけでなく，プログラムの再利用についても重点をおいている．Table 1 にデザインパターンの分類を示した．デザインパターンを学ぶ上で参考にしていただきたい.

Table 1　デザインパターンの分類

種別	パターン名
生成に関するパターン	Abstract Factory
	Builder
	Factory Method
	Prototype
	Singleton
構造に関するパターン	Adapter
	Bridge
	Composite
	Decorator
	Facade
	Flyweight
	Proxy
振る舞いに関するパターン	Chain of Responsibility
	Command
	Interpreter
	Iterator
	Mediator
	Memento
	Observer
	State
	Strategy
	Template Method
	Visitor

3　抽象クラスとインターフェイス

3.1　抽象クラス，インターフェイスとは

- 抽象クラス
  抽象メソッドや抽象変数を持つクラスである．持っている抽象メソッドは宣言のみであり，内部の実装は継承するサブクラスによって決められる．抽象クラスは継承を前提としているため，抽象クラス自体のインスタンスの作成はできない．なお，抽象クラスの宣言はクラス名に abstract 修飾子を付加する (例 public class abstract MyFrame).

- 抽象メソッド
  メソッド名に abstract 修飾子が付けられたメソッドであり (例 abstract int addNum(int a,int b);)，メソッド名，引数，戻り値の型のみ宣言する．具体的な処理内容は抽象メソッドをもつ抽象クラスを継承したサブクラス内に記述される．(記述しない場合はコンパイルエラーとなる)

- インターフェイス
  クラスが実装するメソッドを定義したものである．インターフェイスは抽象メソッドと定数のみをもち,「クラス名 implements インターフェイス名 (例 class MyFrame implements ActionListener)」と宣言されることにより実装される．実装したクラス内では，必ずインターフェイスで定義した変数およびメソッドの内容を記述しなければならない．クラスはインターフェイスをインプリメントすることによって，インターフェイスが定義したメソッドを持つことが保証される．なお，クラスはインターフェイスを複数個インプリメントすることが可能である.

3.2 インターフェイスと抽象クラスの違い

- メソッドの実装
  抽象クラスは抽象メソッドと抽象メソッドでない内部に実装を持つメソッドの両方を宣言することができる．一方でインターフェイスは抽象メソッドのみしか宣言することができない．

- 多重参照
  抽象クラスはサブクラスによって，継承されることで抽象メソッドをサブクラスに実装させる．しかし，Java は多重継承をサポートしていないため，継承できる抽象クラスは 1 つのみである．インターフェースは複数インプリメントすることができる．

4 Iterator パターン

4.1 Iterator パターンとは

Java 言語では，配列 arr の要素を全て表示するには，for 文を使って以下のように書く．ここでは for 文の中で i++で i を 1 ずつ増加させていくことにより，配列の要素全体を最初から順番にスキャン (走査) している．ここで使われている変数 i の動きを抽象化し，一般化したものを，デザインパターンでは Iterator パターンという．

```
for(int i = 0; i < arr.length; i++){
  System.out.println(arr[i]);
}
```

「iterate」という英単語は「繰り返す」という意味であり，Iterator は日本語にすると「反復子」などと呼ばれる． 本章で学ぶ Iterator パターンは，デザインパターンの中でも最もシンプルで頻繁に使われているパターンの 1 つで，要素の集まりを保有するオブジェクトの各要素に順番にアクセスする方法を提供するためのものである．このパターンは，データの内部表現の詳細に感知することなく，標準的なインタフェースを使ってデータのリスト中あるいはコレクション中を順次移動することを可能にする．

Iterator パターンは「データの集合を表現するオブジェクト」から,「集合の各要素へ順番にアクセスするアルゴリズム」を切り離すことを目的としている．Iterator を使う一番の理由は，実装と切り離して数え上げを行うことができる点である．

4.2 クラス構成

この節では Iterator パターンを使用する前後でプログラムがどのように変化するかを比較，検討する．なお，使用するプログラムは別紙に掲載した．

今回使用するプログラムは，名簿に記載されたメンバーを順番に読み込み，出力するというものである．

Iterator パターンを適用していない場合のプログラムのクラス図を Fig. 1 に，適用した場合のプログラムのクラス図を Fig. 2 に示す．

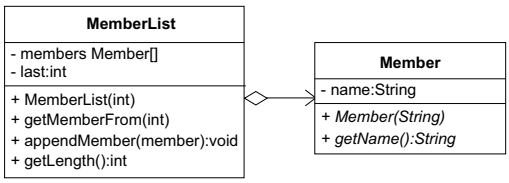

Fig. 1　Iterator パターン適用前のクラス図 (出典 : 自作)

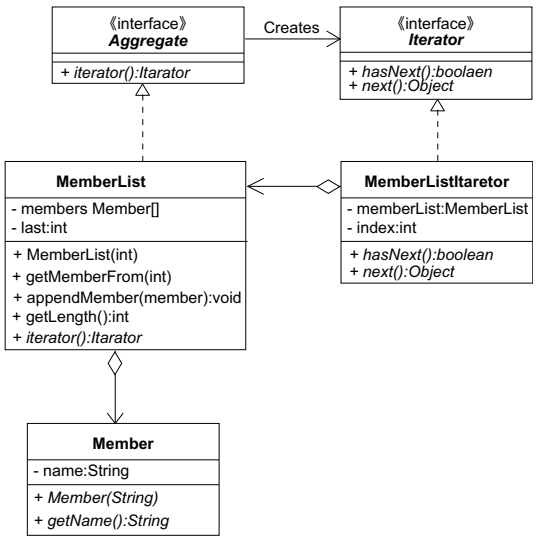

Fig. 2　Iterator パターン適用後のクラス図 (出典 : 自作)

また，各クラスの説明を以下に示す．

- Aggregate インタフェース
  Iterator 役を作り出すインタフェースを定める役目を持ち，iterator メソッドを定めている．

- Iterator インタフェース
  要素を実際にスキャンしていくインスタンスを定める役目を持つ．

- Member クラス
  名簿に入っているメンバを表すクラスである．本の名前を getName メソッドで得る．

- MemberList クラス
  名簿を表現しているクラスである．Aggregate インタフェースを実装する．配列である members というフィールドの大きさ (maxsize) は，MemberList のインスタンスを作るときに指定する．
  iterator メソッドは，MemberList クラスに対応する Iterator として，MemberListIterator というクラスのインスタンスを生成してそれを返す．メンバ

の数を数え上げたいときに，この iterator メソッドが呼び出される．

- MemberListIterator クラス
  MemberListItarator を Iterator として扱うため，Iterator インタフェースを実装している．MemberList クラスのスキャンを行う役目を持つ．index フィールドが現在注目しているメンバを指す添え字になる．
  hasNext メソッドは，Iterator インタフェースで宣言されているメソッドを実装したものである．

5 Adapter パターン

5.1 Adapter パターンとは

Adapter パターンとは「すでに提供されているもの」と「必要なもの」の間のずれを埋めるクラスを作るという考え方である．具体的にはあるクラスのインターフェイスを，求められる他のインターフェイスに適合させるためのクラスを設けることで，インターフェイスに互換性のないクラス同士を組み合わせようというものである．

2 つのクラスのインターフェイスが合わないというのであれば，それらのクラスを書き換えればよいと考えるかもしれない．しかしこの場合，変更したクラスが別のクラスに利用されるなら，その別のクラスまで変更しなければならないということになりかねない．また，十分にテストされたクラスについても新たにテストし直さなければならなくなる．Adapter パターンを用いればこの 2 つのクラスを一切変更せずに結びつけられ，この問題を解消できる．

5.2 クラス構成

サンプルとして給与計算を行うプログラムを示す．このプログラムでは Fig. 3 のように Shacho クラスが Keiribu クラスとつながらないことが問題となっている．

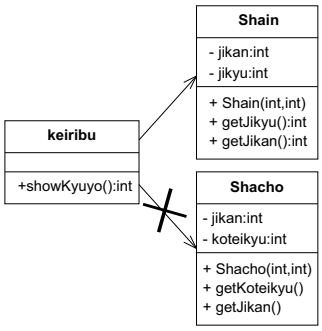

Fig. 3　サンプルプログラムの問題を示すクラス図 (出典:自作)

Adapter パターンを利用せずに Shacho クラスの書き換えによって対応する場合には，(ソースファイルに示すように)Shacho クラスが Shain クラスを継承する必要があり，その際重複するメソッド名の変更が生じてしまっている．この場合のクラス図を Fig. 4 に示す．

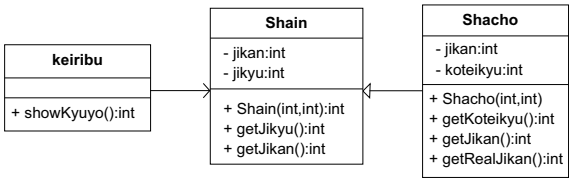

Fig. 4　Adapter パターン適用前のクラス図 (出典:自作)

これに対し，Adapter パターンで対処した場合，クラス図は Fig. 5 のようになる．

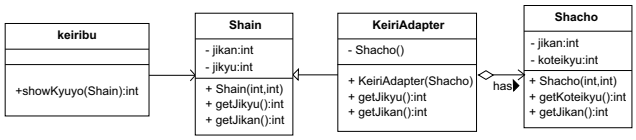

Fig. 5　Adapter パターン適用後のクラス図 (出典:自作)

それぞれのクラスの概要は以下のようになる．

- Shain クラス
  社員を表すクラス．時間と給与を持つ．

- Keiribu クラス
  経理部を表すクラス．Shain クラスのオブジェクトを使い，社員の給与を表示する．

- Shacho クラス
  社長を表すクラス．Keiribu クラスで使用したいが，このままではインタフェースが合わない．

- KeiriAdapter クラス
  Shacho クラスと Shain クラスを結びつけるアダプター．Shain クラスを継承し，Shacho クラスのメソッドを利用して必要なメソッドを提供する．

6 Template Method パターン

6.1 Template Method パターンとは

Template Method パターンは，全体的な処理の流れをメソッドとしてスーパークラスに定義し，その具体的な処理内容をサブクラスにまかせるデザインパターンである．個々のサブクラスはテンプレートとして異なる部分があるが，全体的な処理の流れは常に変わらない．

Template Method パターンは等しい処理の流れをスーパークラスで定義することでサブクラスのコーディングを効率的に行うことができる．また，プログラムを修正する場合には，上位クラスの変更を少なくし，下位クラスを変更できる点でも効率的であると言える．

6.2 クラス構成

例として単純なプログラムのクラス図の解説を行いながら，Template Method パターンについて解説する．Template Method パターンを適用せずに作成したのが Fig. 6 である．

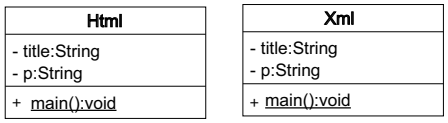

Fig. 6 Template Method パターン適用前のクラス図 (出典：自作)

Fig. 6 では，Html，Xml それぞれのクラスが処理の流れに沿ってプログラムされている．Template Method パターン適用後のクラス図が Fig. 7 である．

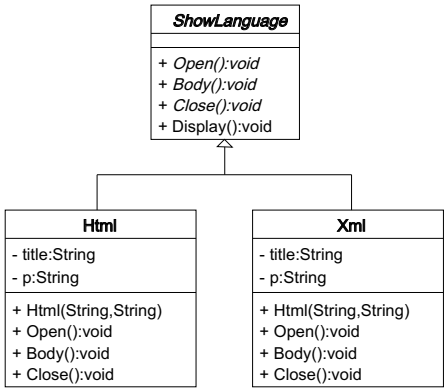

Fig. 7 Template Method パターン適用後のクラス図 (出典：自作)

なお，それぞれのクラスの概要は以下のようになる．

- ShowLanguage クラス
  抽象メソッド定義し，処理の流れを記述しているクラスである．
- Html クラス
  ShowLanguage クラスを継承するサブクラスである．引数から Html ソースを表示する．
- Xml クラス
  ShowLanguage クラスを継承するサブクラスである．引数から Xml ソースを表示する．

Fig. 7 では，ShowLanguage クラスで Html クラスと Xml クラスの処理の流れを一元化し，具体的な処理をサブクラスに任せている．適用後のサブクラスは処理の流れをほとんど考えることなく，プログラムできたのに対し，適用前のプログラムでは，常に処理の流れを考えながら具体的なプログラムを書かなければならない．このようなことからも Template Method パターンの有用性がわかる．

7 Singleton パターン

7.1 Singleton パターンとは

Singleton パターンは生成に関するパターンに含まれており，指定したクラスのインスタンスが絶対に 1 個しか存在しないことを保証するものである．このパターンが有効なのは，システムの中に 1 個しか存在しないものを表現する時である．例えば，コンピュータそのものを表現したクラスや，ウィンドウマネージャ，図書館の貸出帳などがこれにあたる．Singleton パターンの利点としては，プログラム中でそのクラスのインスタンスが 1 個しか存在しないことを保証し，それを表現できるということがある．Singleton パターンを使用しない場合には，プログラマ自身がインスタンスを 1 個しか生成しないように気をつける必要があり，間違って複数生成されてしまう場合も考えられる．

7.2 クラス構成

Singleton パターンで生成したプログラムのクラス図を Fig. 8 に示す．

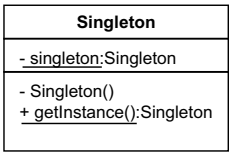

Fig. 8 Singleton パターン適用後のクラス図 (出典:自作)

Singleton クラスでは static なクラス変数として singleton を用意している．なお，singleton は private であるので，クラス内からしか使うことができない．次に，Singleton クラスのコンストラクタは private になっている．これにより，クラス内からしかコンストラクタを呼び出すことができなくなる．つまり，他のクラスからコンストラクタを呼び出すことは禁止されている．最後に，インスタンスを得るためのメソッドとして getInstance メソッドが用意されている．このメソッドは public static であるので他のクラスから呼び出すことが可能である．getInstance メソッドでは下記のような処理を行う．

```
public static Singleton getInstance(){
    if(singleton == null){
        singleton = new Singleton();
    }
    return singleton;
}
```

ここでは，最初にメソッドが出された際にインスタンスを生成し，それ以降は生成済みのインスタンスを返すよ

うになっている．これにより，1 個のインスタンスしか存在しないことが保証される．

8　おわりに

本ゼミでは，プログラミングの中で特によく使用する 4 つのデザインパターンについて解説した．オブジェクト指向の特性を生かしたプログラミングをめざすには，デザインパターンに沿ったプログラミングが必須である．ここに挙げたパターン以外のデザインパターンについても書籍やインターネットから勉強していただきたい．

参考文献

1) 結城 浩 : 増補改訂版 Java 言語で学ぶデザインパターン入門 ソフトバンクパブリッシング (2004)
2) 杉浦 K. : あなたのコードを [賢く] するデザインパターン Java プログラミング 秀和システム (2005)
3) PC View :Solution
   http://www.pc-view.net/Solution/
4) ObjectClub -Java プログラマのためのデザインパターン入門
   http://www.objectclub.jp/technicaldoc/pattern/DPforJavaProgrammers
5) nulab - サルでもわかる 逆引きデザインパターン
   http://www.nulab.co.jp/designPatterns/designPatterns1/designPatterns1-2.html

[問題 1]

Adapter パターンに関する問題

※ HumanAdapter 以外のクラスは書き換えずに，実行できるようにしてください※

依頼者 (以下の Main 文) は名字，名前，年齢をフィールドとしてもち，profile() メソッドでそれらを一覧表示するクラスを求めている．

一方で，そのような属性を持つクラスとしては Human クラスがすでに作られており，このクラスを利用すれば楽に要求を満たせるクラスを作成できそうである．

Human クラスを利用し，かつ profile() メソッドを持つことが保証された HumanAdapter クラスを作成し，実行できるようにせよ．

[Main クラス]

```
public class Main {
          public static void main(String[] args) {

                HumanAdapter h1 = new HumanAdapter("Dai","Aoki",22);
                HumanAdapter h2 = new HumanAdapter("Masahiro","Kiura",22);
                HumanAdapter h3 = new HumanAdapter("Kiyoka","Kida",21);
                HumanAdapter h4 = new HumanAdapter("Masashi","Nishioka",21);

                h1.profile();
                h2.profile();
                h3.profile();
                h4.profile();

        }
}
```

[実行例]

```
---------------
名字：Aoki
名前：Dai
年齢：22
---------------

---------------
名字：Kiura
名前：Masahiro
年齢：22
---------------

---------------
名字：Kida
名前：Kiyoka
年齢：21
---------------

---------------
名字：Nishioka
名前：Masashi
年齢：21
---------------
```

[問題 2]

Adapter パターンに関する問題

※ TokenizerAdapter クラス以外書き換えずに，実行できるようにしてください※

依頼者 (以下の Main 文) は CSV ファイルの記述などに使われるカンマ区切りの文字列を引数として渡したときに，その各要素を int 型の配列にいれて返してくれるメソッド separate() を要している．

このメソッドを持つクラスを作るにあたり，どうやら Java のクラスライブラリにある StringTokenizer クラスを利用すれば効率よく作成できそうである．(TokenizerAdapter クラス作成時に import java.util.StringTokenizer; とする)

以下のメイン文を実行し，下記の通りの実行結果が得られるように TokenizerAdapter クラスを作成せよ．なお，Demand インターフェイスは separate() メソッドの実装を保証するためのものである．

[Main クラス]

```
public class Main {

        public static void main(String[] args) {


                String str = "10,40,33,100,37";
                int[] response;
                Demand d = new TokenizerAdapter(str);
                response = d.separate();
                for(int i = 0;i<response.length;i++){
                        System.out.println(i +"番目の要素:" +response[i]);
                }
        }
}
```

[実行結果]

```
0 番目の要素:10
1 番目の要素:40
2 番目の要素:33
3 番目の要素:100
4 番目の要素:37
```

[StoringTokenizer クラスの概要]

- コンストラクタ StringTokenizer(String str1,String str2)
  str1 の文字列を str2 の文字で分割するための StringTokenizer インスタンスを作成する (new StringTokenizer(”私は，日本，人です”),”，”) とすれば，以下の NextToken() メソッドで”私は”，”日本”，”人です”という文字列が順に取得できる

- String nextToken()
  実行するたびに，コンストラクタで与えた文字列を区切り文字で区切ったものを，nextToken() が呼び出されるたびに先頭からひとつずつ返す.

- Boolean hasMoreToken()
  まだ nextToken() メソッドで取り出されていない要素があるか，即ち nextToken() メソッドがまだ実行可能かを返す.

- int CountTokens()
  nextToken() メソッドを呼び出せる回数を返す.

[StringTokenizer 使用例]

```
import java.util.StringTokenizer;

public class Main {
        public static void main(String[] args) {
                String str = "これは, サンプル, プログ, ラムです";
                StringTokenizer st = new StringTokenizer(str,",");

                while(st.hasMoreTokens()){
                        String s =st.nextToken();
                        System.out.println(s);
                }
        }
}
```

[実行結果]

```
これは
サンプル
プログ
ラムです
```