プログラミング言語各論

第6回

2005年度担当: 中井央

2005年10月27日

1 はじめに

今回も引き続き、デザインパターンを3つ紹介する。Template Method, Factory Method, Abstract Factory である。

できる限り、着目点を述べていくつもりであるので、「オブジェクト指向」の部分を感じ取ってもらいたい。

オブジェクト指向でのプログラミングができるようになるには、そのように作られたプログラムの例をいくつも見て、考え方を理解し、それらに慣れていく必要がある。今回示すパターンも手続き型でのプログラミングでは考えられない (考えにくい) ものである[1]。

今回もサンプルプログラムは結城さんの本 [3] に頼った。

2 Template Method パターン

Template は雛形と訳されることが多い。大まかに手順が決まっているものをここではテンプレートという。

たとえばC言語でファイル入出力をする際の手順はおおまかに次のようになる。

1. ファイルのオープン
2. そのファイルへの入出力処理
3. ファイルのクローズ

基本的にこの流れが変わることはない。もちろん、各プログラムの目的にあわせて、ファイルのオープンでは、固定したファイルを開くのか、指定されたファイルを開くのか、ファイルへの入出力では具体的に何をするのか (読み込みか、書き出しか、両方か) など、変わってくる。

Java などのオブジェクト指向言語によるプログラミングでは、クラスの継承を利用することで、継承したクラスで独自の実装をすることができる。Template Method パターンは、全体的な処理の流れを定義しておき、その流れの中の個々の処理をサブクラスで具体的に実装させたい場合に使用する。

さて、具体例を示そう。プログラムを図1、図2、図3、図4に示す。今回のサンプルでは、前処理、表示、後処理という3つの処理の流れをまず、固定する。しかし、具体的なそれぞれの処理の実装はサブクラスで行なう、というものである。

以下、各々説明していく。

2.1 AbstractDisplay.java

```
 1 public abstract class AbstractDisplay {
 2   public abstract void open();
 3   public abstract void print();
 4   public abstract void close();
 5
 6   public final void display(){
 7     open();
 8     for(int i = 0; i < 5; i++){
 9       print();
10     }
11     close();
12   }
13 }
```

図 1: AbstractDisplay.java

AbstractDisplay.java には今回のお仕事の大まかな流れになる部分を決定するプログラムが記述されてい

[1] C 言語を使ってもオブジェクト指向の考えに則ってプログラミングができないわけではない。
しかし、言語によってあらかじめ継承などの概念が使用できるようになっていることはプログラムの作成者にとって便利であり、オブジェクト指向のプログラミングを (多少は) 容易にしてくれる。

る。AbstractDisplay クラスは抽象クラスであり、継承されたクラスで、open, print, close が実装されることを期待している。一方、display というメソッドには open, print, close の呼び出しを含んでおり、display 自体の宣言には final がつけられている。すなわち、このメソッドは継承したクラスで書き換え (オーバーライド) ができないようになっている。これは全体の処理の流れは継承によって変えられては困るからである。一方で、open, print, close は継承したクラスで実装しなければならない。このようにすることで、処理の流れは固定し、個々の処理の実装は必要に応じて変えることができるという枠組ができるわけである。

2.2 CharDisplay.java

```
 1 public class CharDisplay
     extends AbstractDisplay {
 2   private char ch;
 3   public CharDisplay(char ch){
 4     this.ch = ch;
 5   }
 6
 7   public void open(){
 8     System.out.print("<<");
 9   }
10
11   public void print(){
12     System.out.print(ch);
13   }
14
15   public void close(){
16     System.out.println(">>");
17   }
18 }
```

図 2: `CharDisplay.java`

CharDisplay.java は 1 文字が与えられるとその前後にそれぞれ<<と>>を出し、その間に与えられた文字を 5 回表示するプログラムである。open, print, close のそれぞれでこの機能を実装していることは容易に読めるであろう。

2.3 StringDisplay.java

StringDisplay クラスは文字列を受け取り、その周囲を文字で囲うプログラムである。ここも特に説明は不

```
 1 public class StringDisplay
     extends AbstractDisplay {
 2   private String string;
 3   private int width;
 4
 5   public StringDisplay(String string){
 6     this.string = string;
 7     this.width = string.getBytes().length;
 8   }
 9
10   public void open(){
11     printLine();
12   }
13
14   public void print(){
15     System.out.println("|" + string + "|");
16   }
17
18   public void close(){
19     printLine();
20   }
21
22   private void printLine(){
23     System.out.print("+");
24     for(int i = 0; i< width; i++){
25       System.out.print("-");
26     }
27     System.out.println("+");
28   }
29 }
```

図 3: `StringDisplay.java`

要であろう。

2.4 Main.java

```
 1 public class Main {
 2   public static void main(String args[]){
 3     AbstractDisplay d1 = new CharDisplay('H');
 4     AbstractDisplay d2 =
             new StringDisplay("Hello, world.");
 5     AbstractDisplay d3 =
             new StringDisplay("こんにちは。");
 6
 7     d1.display();
 8     d2.display();
 9     d3.display();
10   }
11 }
```

図 4: `Main.java`

Main では、AbstractDisplay クラスの変数に実際に

は継承したクラスのいずれかのインスタンスが代入されている。**具体的な実装はサブクラスに任す**ということが、このパターンの特徴である。display を呼び出せば、必ず open, print, close の順に処理が実行されることが保証されるわけである。

2.5 Template Method パターンの事例

中井の研究室ではコンパイラのうち、文法にあっているかチェックするためのプログラムを自動生成する構文解析器生成系を作成した。詳細は省くが、Template Method パターンを使用した理由は次の通りである。まず、構文解析処理プログラムの基本的な動作は決まっており、継承したクラスで勝手に書き換えられては困る。構文解析処理プログラムでは現在の状態と現在の入力記号から、エラー、受理、シフト、還元のいずれかの動作をする。つまり、この部分が固定的な流れである。一方でこれらの具体的な処理 (本当はその前処理と後処理) は生成されたプログラムを拡張して使いたい人が自由に決めてよいようにしたかったわけである。このため、それぞれの処理にそれぞれの前処理、後処理を空で埋めておき、具体的になにかさせたい場合は、使いたい人が継承してその部分を実装する。

卒業研究などでプログラムを実装する時には、割と利用できそうなパターンでもある。

2.6 この節のまとめ

この節では Template Method パターンを紹介した。特徴としては、大まかな流れが決まっている場合、まず、それをスーパークラスで設定し、final を宣言することで固定してしまう。つまり、継承したクラスでのオーバーライドを許さない。ただし、そのメソッドの中で使われるいくつかのメソッドは逆に継承するクラスで独自の処理を実装するようにする。

このほかに紹介する多くのプログラムでそうであるが、オブジェクト指向に基づいて作られるプログラムの場合、多くはスーパークラスの変数にそれを継承したクラスのインスタンスが代入される。この例でも宣言されているのはすべて AbstractDisplay クラスの変数であるが、実際にそこに代入されているのはそれを継承した CharDisplay か StringDisplay クラスのインスタンスである。

スーパークラスでインタフェース (インスタンスにアクセスするためのメソッド群) を定義しておき、用途に応じて継承したクラスをいくつか用意し、実際に必要な場面でそれらをスーパークラスの変数で扱うことで、見かけ上統一的に扱えて、かつ実際の場面ではその状況に応じた振る舞いをするプログラムが記述できる。この一文は最初のころは非常に難解に思われるであろうが、継承のありがたみがわかってくると納得がいくはずである。

今回の例での main メソッドを自身の言葉で説明をつけてみたら、もっと理解が深まると思う。

演習 1

1. 図 1 〜 図 4 を入力し、コンパイル、実行せよ。
2. 図 2、図 3 を適当に変更し、コンパイル、実行してみよ。
3. 図 4 を適当に変更し、コンパイル、実行してみよ。

3 Factory Method パターン

まずは例によってサンプルプログラムを示す。この例では 2 つのパッケージを用いている。framework と idcard である。ここでは、このパターンのサンプルのストーリーは次のようなものである。

いま、ID カードが必要であるとする。カードを単体で作ることは可能ではあるが、カードを作るだけでなく、そのカードが有効になるよう、登録作業も必要となる。そのため、ユーザは ID カードが欲しい場合には工場に依頼してカードを発行してもらう。

これをプログラムでシミュレートするには、ID カードという製品をクラスにするとともに必要に応じて ID カードを作り出す工場もクラスとする。

ところで、ここでは製品を ID カードとしたが、別のものであっても同じプロセスを踏んで製品を発行する過程 (プロセス) はありうる。たとえば、ネットワークカードやシリアル番号がつけられるような計算機製品などである。

そうすると発行して欲しい製品とそれを発行する工場を抽象化することができる。逆にそのようにクラスを設計しておけば、新たな製品についてクラスを作り

たくなった場合、その抽象化されたクラスを継承する形で、この製品と工場の関係を実装することが (一から作るより) 容易になる。

さて、これらを踏まえてサンプルプログラムを見ていこう。サンプルプログラムを見た後、もう一度これらのことを反芻してみる。

なお、以下のサンプルを実行するには、まず、framework, idcard の 2 つのディレクトリを作り、Factory(図 5), Product(図 6) の 2 つのクラスは framework ディレクトリの中に入れ、IDCard(図 8) と IDCardFactory(図 7) の 2 つのクラスは idcard ディレクトリの中に入れる。framework と idcard があるディレクトリに Main クラス (図 9) を置き、そこで java コマンドを実行する。

蛇足だが、一応手順を記述しておく。

```
uni% mkdir framework
uni% mkdir idcard
uni% cd framework
...  Factory.java, Product.java を作る
uni% cd ..
uni% cd idcard
...  IDCardFactory.java, IDCard.java を作る
uni% cd ..
...  Main.java を作る
uni% javac Main.java
```

3.1 framework.Factory

```
 1 package framework;
 2
 3 public abstract class Factory {
 4   public final Product create(String owner){
 5     Product p = createProduct(owner);
 6     registerProduct(p);
 7     return p;
 8   }
 9
10   protected abstract Product
       createProduct(String owner);
11   protected abstract void
       registerProduct(Product product);
12 }
```

図 5: Factory.java

Factory クラスは抽象クラスであり、Template Method パターンになっている。4 行目の create メソッドは製品 (Product) を作るものであり、具体的な動作は createProduct を呼び、その後、registerProduct を呼び出す。それぞれは継承したクラスで具体的に実装される必要がある。

3.2 framework.Product

```
1 package framework;
2
3 public abstract class Product {
4   public abstract void use();
5 }
```

図 6: Product.java

こちらは非常に簡単である。製品をシミュレートするのに使用を表す use というメソッドのみを宣言している。

3.3 idcard.IDCardFactory

```
 1 package idcard;
 2
 3 import framework.*;
 4 import java.util.*;
 5
 6 public class IDCardFactory extends Factory {
 7   private Vector owners = new Vector();
 8
 9   protected Product
      createProduct(String owner){
10     return new IDCard(owner);
11   }
12
13   protected void
      registerProduct(Product product){
14     owners.add(((IDCard)product).getOwner());
15   }
16
17   public Vector getOwners(){
18     return owners;
19   }
20 }
```

図 7: IDCardFactory.java

こちらは具体的な ID カードの工場である。Factory での抽象メソッドを具象化している。内容的には特に説明は不要であろう。

3.4 idcard.IDCard

```
 1 package idcard;
 2
 3 import framework.*;
 4
 5 public class IDCard extends Product {
 6   private String owner;
 7   IDCard(String owner){
 8     System.out.println(owner +
                           "のカードを作ります。");
 9     this.owner = owner;
10   }
11
12   public void use(){
13     System.out.println(owner +
                           "のカードを使います。");
14   }
15
16   public String getOwner(){
17     return owner;
18   }
19 }
20
```

図 8: IDCard.java

このクラスは具体的な製品 ID を表現している。こちらも特に説明は不要であろう。

3.5 Main

```
 1 import framework.*;
 2 import idcard.*;
 3
 4 public class Main {
 5   public static void main(String args[]){
 6     Factory factory = new IDCardFactory();
 7     Product card1 =
         factory.create("結城浩");
 8     Product card2 =
         factory.create("とむら");
 9     Product card3 =
         factory.create("佐藤花子");
10
11     card1.use();
12     card2.use();
13     card3.use();
14   }
15 }
```

図 9: Main.java

図 9 の 6 行目では Factory 型の変数 factory に IDCardFactory のインスタンスを代入している。7〜9 行目では factory を使って実際のインスタンス生成を行っている。これを受ける変数はどれも Product 型であることにも注意しよう。

3.6 考察

今回のサンプルプログラムのクラス図を図 10 に示す。

Factory クラスと Product クラスは抽象クラスであった。そして Factory クラス (のサブクラス) のインスタンスは Product クラス (のサブクラス) のインスタンスを生成する、という関係がある。今回のサンプルではこの関係にあてはまる具体的なクラスとして IDCardFactory と IDCard クラスがある。この 2 つはペアで使われることが前提となっている。

Factory Method パターンはこのような対になるクラスを表現するためのパターンである。

[1](日本語版) の 118 ページでは、アプリケーション内で扱う図形オブジェクトとそれを操作するオブジェクトの関係で説明している。それをもとに関連だけを記述した図を図 11 に示す。

この図では、抽象クラスとしての Figure と具体的な図形オブジェクトを表すクラスとしての LineFigure, TextFigure, CircleFigure が記述されていて、Figure クラスのオブジェクトに対する操作をつかさどる Manipulator クラスと、それを継承した具体的な各オブジェクトに対する操作を定義している LineManipulator, TextManipulator, CircleManipulator が記述されている。LineFigure クラスのオブジェクトは LineManipulator クラスのオブジェクトと対になる。他も同様である。このような場合に、Figure クラスを継承した実際に使われるクラスのインスタンスに対して、それを扱うためのクラスを生成させることで、ペアとなるクラスの対を必ず一緒に使うようにプログラムに組み入れることができる。

ペアで使われるクラスを表現するのが Factory Method パターンである。

演習 2

1. 図 5〜図 9 を入力し、コンパイル、実行せよ。

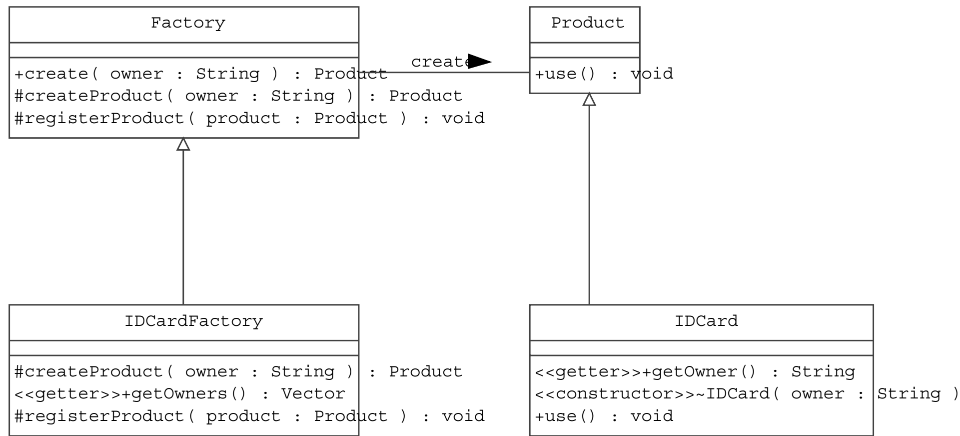

図 10: サンプルプログラムのクラス図

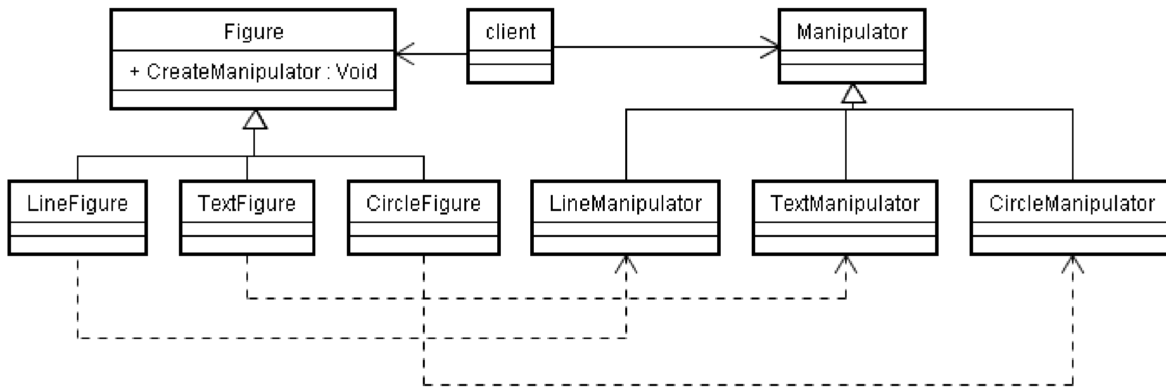

図 11: GoF の本の Factory Method の説明で出てくる図 (関連だけを写し取ったもの)

4 Abstract Factory パターン

さて、今回最後のパターンは Abstract Factory パターンである。まずはサンプルプログラムを大まかなに見てみよう。

この例では、Web ページ作成クラスというものを作っている。ある Web ページを構成するものには、次があると考えてプログラムを構成している (後の参照のため、これらをページ構成案という)。

1. 表示したい個々の要素 (今回はリンク)
2. 表示したい個々の要素をひとまとめにするための要素
3. ページ全体

このような関係は Composite パターンで表現できるのだった。さて、先に図 17 を見てみよう。

12 行目は与えられた引数にしたがって「具体的な工場」を選んでいるのであるが、詳細は後ほど、説明する。工場オブジェクトの役割はその工場が作り出す部品を提供することである。その意味ではこれらの工場は Factory Method パターンになっている。

14 行目から 33 行目まではページを作っているところである。工場に対し、createLink を実行することで具体的なリンクの部品を得ている。同様に createTray を実行することでいくつかの要素をまとめる部品を得ている。

このようにして各部品を作成し、最後に 35 行目で

ページを作っている。

ここまでの説明をまとめておくと、必要な個々の部品を作り、それらをまとめて 1 つの作品 (??) ＝ページを作っている。

さて、このプログラムは結局、リンク集を作るプログラムであるが、ここまでの説明では具体的にどう作るかは特に述べていない。具体的に述べていないというのは抽象的に述べているとも言える (日常会話ではあまりそういう言い方はしないが…)。

で、リンク集を HTML で記述するのだとして、考えられるページの構成の 1 つは、リスト機能 (`<ul>`タグなどを使うもの) である。リストは多段になってもよいとする。

もう 1 つ、リンクの集まりをテーブル (`<table>`タグを使う) の形式にする方法もある。

話を簡単にするため、混在は考えない (:p)。

そうすると、ここまでに「抽象的な話」の中で述べてきた「工場」を具体的に考えていくことになる。1 つはリスト機能のタグからなる部品群からなるもの、もう 1 つはテーブルのタグからなる部品群からなるもの、である。

上記のページ構成案で示した要素間の関係は保ったまま、具体的なものを作っていくわけである。このとき、ページ構成案にしたがって「抽象的な工場」(とその部品群) をクラスとして構成しておき、「具体的な工場」を抽象的な工場の個々の要素 (クラス) を継承して作っていくようにする。すると、ページ構成案に沿った具体的な工場ができあがる。

では、以下で各クラスを見ていこう。なお、以下に示すソースは、次のディレクトリに置いてある。パターンを知るには自身で打ち込むほうがよいと思うが、適宜コピーして利用しても良い。

```
~nakai/Abstract_Factory
```

4.1 factory パッケージ

リンクのページを作成するための工場を構成するクラス群を factory パッケージの中に入れている。このパッケージには次のクラスがある。

- Item.java
- Link.java
- Tray.java
- Page.java
- Factory.java

4.1.1 factory.Item

```
1 package factory;
2
3 public abstract class Item {
4   protected String caption;
5   public Item(String caption){
6     this.caption = caption;
7   }
8   public abstract String makeHTML();
9 }
```

図 12: Item.java

4 行目の caption はリンク先の説明 (名前) などを示す文字列である。`<a href="abc.html">abc</a>`とする際の、間に挟まれた文字列 abc のことである。抽象メソッドとして makeHTML が宣言されている。このオブジェクトが関与する、生成すべき文字列を返す。具体的にはこのクラスを継承したクラスでそれを実装する。

4.1.2 factory.Link

```
1 package factory;
2
3 public abstract class Link extends Item {
4   protected String url;
5   public Link(String caption, String url){
6     super(caption);
7     this.url = url;
8   }
9 }
```

図 13: factory.Link.java

クラス Link は HTML のハイパーリンクを表現した抽象クラスである。リンクを作る際、`<a>`タグには、リンク先の url が必要だし、そのリンク先を説明する文字

列も必要となる。コンストラクタではそれをセッティングしている。

4.1.3 factory.Tray

```
 1 package factory;
 2 import java.util.Vector;
 3
 4 public abstract class Tray extends Item {
 5   protected Vector tray = new Vector();
 6   public Tray(String caption){
 7     super(caption);
 8   }
 9
10   public void add(Item item){
11     tray.add(item);
12   }
13 }
```

図 14: factory.Tray.java

トレイは複数の Item(すなわちそれを継承した Tray または Link) を持つので、Vector の変数を保持している。トレイに対しては add メソッドを利用することで、そのトレイに追加したいオブジェクトを指定できる。

4.1.4 factory.Page

クラス Page は最終的に生成される「製品」を表している。ここで定義しているメソッドは add と output である。add はページへの要素の追加を表しており、output はこのページを指定されたタイトルを名前とするファイルへ出力する。このクラスの makeHTML は実装されていない。

4.1.5 factory.Factory

Factory クラスはページを作るための工場を表すクラスである。5 行目から 8 行目では、文字列として受け取ったクラス名を持つクラスのインスタンスの生成を行っている。

```
 1 package factory;
 2 import java.io.*;
 3 import java.util.Vector;
 4
 5 public abstract class Page {
 6   protected String title;
 7   protected String author;
 8   protected Vector content = new Vector();
 9
10   public Page(String title, String author){
11     this.title = title;
12     this.author = author;
13   }
14
15   public void add(Item item){
16     content.add(item);
17   }
18
19   public void output(){
20     try {
21       String filename = title + ".html";
22       Writer writer = new FileWriter(filename);
23       writer.write(this.makeHTML());
24       writer.close();
25       System.out.println(filename +
                           " を作成しました。");
26     } catch (IOException e){
27       e.printStackTrace();
28     }
29   }
30
31   public abstract String makeHTML();
32 }
```

図 15: factory.Page.java

4.2 Main

main メソッドでは、まず、Factory のインスタンスを作成し、工場に対してそれぞれの要素の作成を依頼する形でページを作成している。まず、14～20 行目ではリンクを作っている。それぞれのリンクはいくつかのカテゴリに分けられる。それぞれのカテゴリを表現するため、それぞれを Tray クラスのインスタンスとして作り、そこに各リンクを格納している (22～33 行目)。31 行目では Tray の中に Tray を入れていることに着目してほしい。

最後にページを作成し、ページにこれまでに作った Tray を追加している。

もちろん、ページを作っていく手順はこれ以外でもよい。たとえば、最初にページのインスタンスを作成しておき、そこに要素を追加していってもよい。ただし、入れ子が何重かになる場合はボトムアップに (深

```
 1 import factory.*;
 2
 3 public class Main {
 4   public static void main(String args[]){
 5     if (args.length != 1){
 6       System.out.println("Usage: java Main class.name.of.ConcreteFactory");
 7       System.out.println("Example 1: java Main listfactory.ListFactory");
 8       System.out.println("Example 2: java Main tablefactory.TableFactory");
 9       System.exit(0);
10     }
11
12     Factory factory = Factory.getFactory(args[0]);
13
14     Link asahi = factory.createLink("朝日新聞", "http://www.asahi.com/");
15     Link yomiuri = factory.createLink("読売新聞", "http://www.yomiuri.co.jp");
16
17     Link us_yahoo = factory.createLink("Yahoo!", "http://www.yahoo.com/");
18     Link jp_yahoo = factory.createLink("Yahoo!Japan", "http://www.yahoo.jp/");
19     Link excite = factory.createLink("Excite", "http://www.excite.com/");
20     Link google = factory.createLink("Google", "http://www.google.com/");
21
22     Tray traynews = factory.createTray("新聞");
23     traynews.add(asahi);
24     traynews.add(yomiuri);
25
26     Tray trayyahoo = factory.createTray("Yahoo!");
27     trayyahoo.add(us_yahoo);
28     trayyahoo.add(jp_yahoo);
29
30     Tray traysearch = factory.createTray("サーチエンジン");
31     traysearch.add(trayyahoo);
32     traysearch.add(excite);
33     traysearch.add(google);
34
35     Page page = factory.createPage("LinkPage", "結城 浩");
36     page.add(traynews);
37     page.add(traysearch);
38     page.output();
39   }
40 }
```

図 17: Main.java

```
 1 package factory;
 2
 3 public abstract class Factory {
 4   public static Factory
      getFactory(String classname){
 5     Factory factory = null;
 6     try {
 7       factory = (Factory)Class.
               forName(classname).newInstance();
 8     } catch (ClassNotFoundException e){
 9       System.out.println("クラス " +
                           classname +
                         " が見つかりません。");
10     } catch (Exception e){
11       e.printStackTrace();
12     }
13
14     return factory;
15   }
16
17   public abstract Link createLink
                  (String caption, String url);
18   public abstract Tray createTray
                  (String caption);
19   public abstract Page createPage
                  (String title, String author);
20 }
```

図 16: factory.Factory.java

いほうから) 作るほうがわかりやすいかもしれない。

4.3 listfactory パッケージ

listfactory パッケージには、HTML のリスト機能 (<ul>など) を使用してページを構成するためのクラスが集められている。このパッケージには次のクラスがある。

- ListFactory.java
- ListLink.java
- ListTray.java
- ListPage.java

4.3.1 listfactory.ListFactory.java

ListFactory クラスはリスト機能を使って要素を作り出す具体的な工場である。後に述べるようにユーザはこの工場を介して部品を得ることで、一連のリスト機能を持つ部品が得られる。

```
 1 package listfactory;
 2 import factory.*;
 3
 4 public class ListFactory extends Factory {
 5   public Link createLink(String caption,
                            String url){
 6     return new ListLink(caption, url);
 7   }
 8
 9   public Tray createTray(String caption){
10     return new ListTray(caption);
11   }
12
13   public Page createPage(String title,
                            String author){
14     return new ListPage(title, author);
15   }
16 }
```

図 18: ListFactory.java

4.3.2 listfactory.ListLink

```
 1 package listfactory;
 2 import factory.*;
 3
 4 public class ListLink extends Link {
 5   public ListLink(String caption, String url){
 6     super(caption, url);
 7   }
 8
 9   public String makeHTML(){
10     return "  <li><a href=\"" + url + "\">"
              + caption + "</a></li>\n";
11   }
12 }
```

図 19: ListLink.java

Link クラスを継承した ListLink クラスでは、makeHTML を実装している。

4.3.3 listfactory.ListTray

Tray クラスを継承した ListTray クラスでも、やはり、makeHTML を (具体的に) 実装している。リストの中にリストを入れる形となるので、<ul>タグの入れ子なる。

```
1 package listfactory;
2 import factory.*;
3 import java.util.Iterator;
4
5 public class ListTray extends Tray {
6   public ListTray(String caption){
7     super(caption);
8   }
9
10   public String makeHTML(){
11     StringBuffer buffer = new StringBuffer();
12     buffer.append("<li>\n");
13     buffer.append(caption + "\n");
14     buffer.append("<ul>\n");
15
16     Iterator it = tray.iterator();
17     while(it.hasNext()){
18       Item item = (Item)it.next();
19       buffer.append(item.makeHTML());
20     }
21
22     buffer.append("</ul>\n");
23     buffer.append("</li>\n");
24
25     return buffer.toString();
26   }
27 }
```

図 20: ListTray.java

4.3.4 listfactory.ListPage

ListPage クラスでも同様に makeHTML を実装している。

4.4 tablefactory パッケージ

さて、抽象的な工場に対して具体的な工場をもう 1 つ用意している。HTML のリストとしてリンクのページを構成するのではなく、<table>タグを用いたページを構成するための具体的な工場である。

基本的には makeHTML を実装する部分が異なるだけなので、リストを掲載するだけにする。

- TableFactory.java (図 22)
- TableLink.java (図 23)
- TableTray.java (図 24)
- TablePage.java (図 25)

```
1 package listfactory;
2 import factory.*;
3 import java.util.Iterator;
4
5 public class ListPage extends Page {
6   public ListPage(String title, String author){
7     super(title, author);
8   }
9
10   public String makeHTML(){
11     StringBuffer buffer = new StringBuffer();
12     buffer.append("<html><head><title>" +
                   title + "</title></head>\n");
13     buffer.append("<body>\n");
14     buffer.append("<h1>" + title + "</h1>\n");
15     buffer.append("<ul>\n");
16
17     Iterator it = content.iterator();
18     while(it.hasNext()){
19       Item item = (Item)it.next();
20       buffer.append(item.makeHTML());
21     }
22
23     buffer.append("</ul>\n");
24     buffer.append("<hr><address>" + author
                   + "</address>\n");
25     buffer.append("</body></html>\n");
26
27     return buffer.toString();
28   }
29 }
```

図 21: ListPage.java

演習 3

1. Abstract Factory パターンの例題プログラムを打ち込んで、実行してみよ。出力された結果をブラウザで確認してみよう。
2. また、main メソッドの中身を書き換えて、自身のリンクページを出力するようにしてみよ。

4.5 Abstract Factory のまとめ

今回のサンプルは、ページを作るためのプログラムを考えた時、どのような設計にするかを考えるものであった。

日常では、自分のためだけにリスト機能だけのプログラムを作るかも知れない。しかし、欲 (??) を出して、テーブル版も作ってみたいと考えた時、リスト機能のプログラムと大雑把に構成が同じでよいことに気づくかもしれない。そうすると個々の機能 (リストや

```
 1 package tablefactory;
 2 import factory.*;
 3 import java.util.Iterator;
 4
 5 public class TableTray extends Tray {
 6   public TableTray(String caption){
 7     super(caption);
 8   }
 9
10   public String makeHTML(){
11     StringBuffer buffer = new StringBuffer();
12     buffer.append("<td>");
13     buffer.append("<table width=\"100%\" border=\"1\"><tr>");
14     buffer.append("<td bgcolor=\"#cccccc\" align=\"center\" colspan=\"" +
15                   tray.size() + "\"><b>" + caption + "</b></td>");
16     buffer.append("</tr>\n");
17
18     buffer.append("<tr>\n");
19
20     Iterator it = tray.iterator();
21     while(it.hasNext()){
22       Item item = (Item)it.next();
23       buffer.append(item.makeHTML());
24     }
25
26     buffer.append("</tr></table>");
27     buffer.append("</td>");
28
29     return buffer.toString();
30   }
31 }
```

図 24: TableTray.java

らテーブルやら) のことは忘れて、構成そのものに着目してクラス間の関係を考えることができる (かもしれない)。

プログラムを作る時に始めから抽象的に捉えてクラスを設計できるのが理想かも知れないが、実際にはそのようにできないことが多い。自分がコーディングをしてみて、向き合っている問題をよく把握できてきたら、そのような抽象化も頭に浮かんでくるようになる。

今回紹介した Abstract Factory では、Factory Method パターンも使用している。もう一度、図 17 を眺めてみるとわかると思うが、利用者から見ると、12 行目を除いて、具体的な工場についての記述は一切ない。このサンプルでは getFactory で少しトリッキーなことをして、図 17 には、ListFactory も TableFactory も現れていないが、実際に Abstract Factory を使う際には具体的な工場のインスタンスを与える記述はあっても良い。ただし、それ以外は抽象的な工場として定義したメソッドのみを使うことで、具体的な実装とは関係なくページが作成できている点を理解して欲しい。

5 おわりに

これまでにいくつかのパターンを紹介してきた。確かに「パターン」といって紹介したのであるが、パターンの名前にとらわれる前に、そこで使用されているオブジェクト指向の考え方に目を向けてほしい。

まず、抽象的にものを捉えるというものである。同じカテゴリに属するものが存在すれば、カテゴリ名とその具体的なものと言う風に抽象的に捉えることができる。たとえば、足し算や掛け算を考えると、これらはまとめて演算と呼ぶ。これらは二項演算であり、(中置記法であれば) 左の項と右の項を持つ。項は演算かもしれないし単なる数値かもしれない。

抽象化は、聞いてみるとなるほどと思えるが、自分がなにかことにあたって抽象化しないといけないとな

```
 1 package tablefactory;
 2 import factory.*;
 3 import java.util.Iterator;
 4
 5 public class TablePage extends Page {
 6   public TablePage(String title, String author){
 7     super(title, author);
 8   }
 9
10   public String makeHTML(){
11     StringBuffer buffer = new StringBuffer();
12     buffer.append("<html><head><title>" + title + "</title></head>\n");
13     buffer.append("<body>\n");
14     buffer.append("<h1>" + title + "</h1>\n");
15     buffer.append("<table width=\"80%\" border=\"3\">\n");
16
17     Iterator it = content.iterator();
18     while(it.hasNext()){
19       Item item = (Item)it.next();
20       buffer.append("<tr>" + item.makeHTML() + "</tr>");
21     }
22
23     buffer.append("</table>\n");
24     buffer.append("<hr><address>" + author + "</address>\n");
25     buffer.append("</body></html>\n");
26
27     return buffer.toString();
28   }
29 }
```

図 25: TablePage.java

ると、果たしてうまく抽象化できているのかわからなくなってしまうこともある。このようなとき、経験がものをいう。すなわち、いろいろなプログラムのパターンを知っていればこういう場合にはこのようにクラスを作っておけばよいのだ、というヒントになる。それを示した 1 つの指標がデザインパターンであり、自分でプログラムを作る際に無理やりデザインパターンにする必要はない。しかし、自分がプログラムを作ろうとしている目的を考えて、適用できそうなパターンを探してみるのは、問題解決への 1 つのアプローチではある。

その意味で、デザインパターンについて、詳しい理解は必要になった時でも構わないので、どのようなものがあるのかを見知っておくことは重要である。

Java の API をいくつか紹介したが、それらの API でもデザインパターンに基づいて作られているものがたくさん存在する。中井は Java を Java らしく使うには、必要な API を探し、使い方に慣れることが重要だと思っている。必ずしも Sun が用意する Java API のページの情報だけでは、プログラムの書き方がわからないことが多い。いまはインターネットを利用することで、その API の具体的な利用方法を得たり、Java を利用したプログラムのソース (Java SDK の配布物にも含まれている) を得たりすることができる。大いに活用されたい。

最後に中井が参考にしたデザインパターンの本を列挙する。

1. Design patterns: Elements of Reusable Object-Oriented Software [1]
   (邦訳：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)
2. Java 言語で学ぶデザインパターン入門 [3]
3. Java 開発者のためのアンチデザインパターン [2]
4. Java デザインパターン徹底攻略 [4]

```
 1 package tablefactory;
 2 import factory.*;
 3
 4 public class TableFactory extends Factory {
 5   public Link createLink(String caption,
                            String url){
 6     return new TableLink(caption, url);
 7   }
 8
 9   public Tray createTray(String caption){
10     return new TableTray(caption);
11   }
12
13   public Page createPage(String title,
                            String author){
14     return new TablePage(title, author);
15   }
16 }
```

図 22: TableFactory.java

```
 1 package tablefactory;
 2 import factory.*;
 3
 4 public class TableLink extends Link {
 5   public TableLink(String caption, String url){
 6     super(caption, url);
 7   }
 8
 9   public String makeHTML(){
10     return "<td><a href=\"" + url + "\">" +
              caption + "</a></td>\n";
11   }
12 }
```

図 23: TableLink.java

参考文献

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1995.
(邦訳) 本位田真一、吉田和樹: 『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』, ソフトバンクパブリッシング株式会社, (1999).

[2] 安藤利和: *Java* **開発者のためのアンチデザインパターン**, 技術評論社, 2002.

[3] 結城浩: *Java* **言語で学ぶデザインパターン入門**, SOFTBANK Publishing, 2001.

[4] 日立ソフトウェアエンジニアリング (株) インターネットビジネス部: *Java* **デザインパターン徹底攻略**, 技術評論社, 2002.