プログラミング言語各論

第5回

2005年度担当: 中井央

2005年10月20日

1 はじめに

デザインパターンは、最初にGoF[1]によって著された[1]。デザインパターンは熟練プログラマがこれまでに積んだ経験に基づいて、どのような問題に対し、どのようにクラスどうしの関係を作ればよいか、ということを体系的にまとめたものといえる。

ここでは、デザインパターンのうちのいくつかをとりあげ、C言語でプログラムを作ってきた場合とは異なる面を見て、そこからオブジェクト指向的なプログラミングに触れることを目的とする。Javaの入門的な部分で出てきた継承という言葉とそれをどのように活かすのかということが少しでも**体感**できればよいと思う。

以下、今回はComposite, Decorator, Visitorという3つのパターンを紹介する。なお、以下におけるサンプルは結城浩さんの「Java言語で学ぶデザインパターン」[3]による。この他、次の本を参考にした。

1. Java デザインパターン徹底攻略 [4]
2. Java 開発者のためのアンチデザインパターン [2]

2 UMLについて簡単に

オブジェクト指向でプログラミングを行う際、クラスとクラスの関係や制御の流れなどが目に見えたほうがよい場合が(結構)ある。そのような表現にUML (Unified Modeling Language) [2]を使うことがある。UMLはかなり大きな仕様なので、それだけで一冊(以上??)の本が書けるほどのものである。ここでは、この演習で必要となる、クラス間の関係の表現のいくつかを紹介する。

2.1 クラス図

UMLのクラス図 (Class Diagram) はクラスやインスタンス、インタフェースなどの静的な関係を表現したものである。ここでは関係としては、クラスの継承関係、インタフェースと実装の関係、クラスに宣言されているフィールドが別のクラス(のオブジェクト)を指す関係を示す。

まず、クラスを表現するには矩形(長方形)を用いる。

クラスを表す図は、矩形が横の線で区切られていて、一番上はクラス名が入り、2番目にはフィールド名の一覧が入る。3番目にはメソッド名の一覧が入る(図1)。abstractクラスの場合、クラス名は斜字体で書かれる。staticなフィールドやメソッドにはその名前に下線が引かれる。private, protected, publicなどのアクセス制御はフィールドやメソッドの名前の前に、それぞれ、-, #, +がつく。

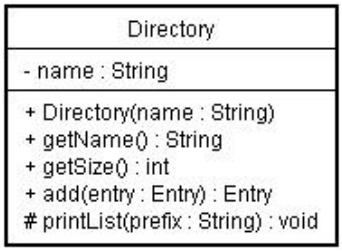

図 1: クラスを表現する図

継承を表すには2つのクラス間を白抜き三角を継承元とする矢印で結ぶ(図2)。

[1]Design Patternという本を著した4人の著者のことをGang of Fourと呼び、それを縮めたGoFという表現が使われる。

[2]http://www.uml.org/

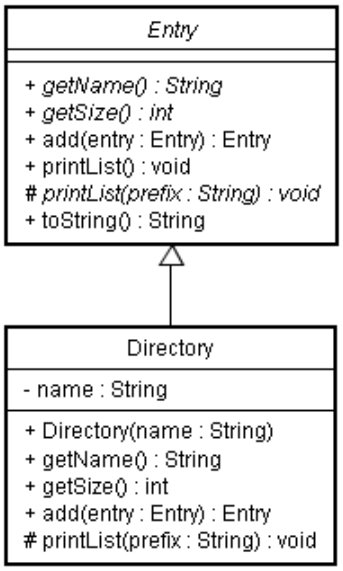

図 2: 継承を表現する図

2 つのクラスの関連は図 3 のように矢印を使って表す。この図では、Class1 が Class2 を使用することを示している。

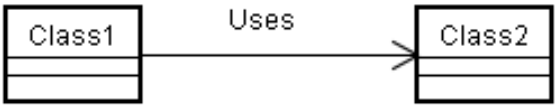

図 3: 関係を表現する図

最後は集約 (aggregation) の表現を図 4 に示す。集約とは「持っている」(has a) 関係のことである。

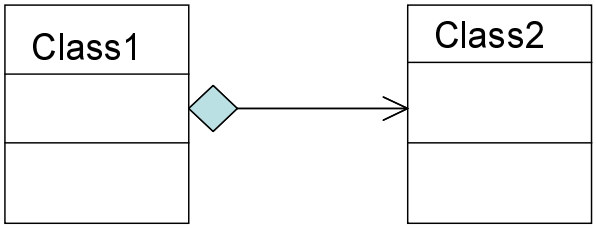

図 4: 集約を表現する図

3 Composite パターン

中身と容れ物を同一視するという考え方を表現するのに Composite パターンがある。コンピュータにおけるファイルシステムは木構造を持つよう設計されている。データが収められているものをファイルと呼ぶ。ここにディレクトリ (あるいはフォルダ) という考えを導入する。すなわち、ファイルを収める特殊なファイルである。違う言い方をすれば、ファイルには 2 種類ある。データが入っているデータファイルと、ファイルが入っているディレクトリファイルである。以下、この例にしたがって Composite パターンを説明していく。

3.1 Composite パターンのクラス図

ファイルを例にとった Composite パターンのクラス図を図 5 に示す。ファイルという概念には上述のようにディレクトリファイルとデータファイルがある。この図ではディレクトリファイルをクラス Directory で、データファイルをクラス File で表現している。その両者を統一的に表すファイルという概念はクラス Entry で表現する。

関連図を見ると集約によって、Directory が Entry を指している。そして Directory は Entry を継承しているので、この図はループになっている。これによって容器と中身を同一視している。容器と中身の同一視とは、すなわち、木構造などの再帰的定義のことである。再帰的な関係はプログラミングの世界ではよく出てくる。そのようなデータ構造をクラスとして設計する際には Composite パターンが利用できるわけである。

3.2 サンプルプログラム

サンプルプログラムを図 6, 図 7, 図 8, 図 9, 図 10 に示す。これら 5 つのクラスは次の意味を持つ。

File データが収められている 1 つのファイル

Directory ディレクトリファイル

Entry File と Directory を同一視するためのクラス

FileTreatmentException ディレクトリにはファイルを追加できるが、File には Entry (File または

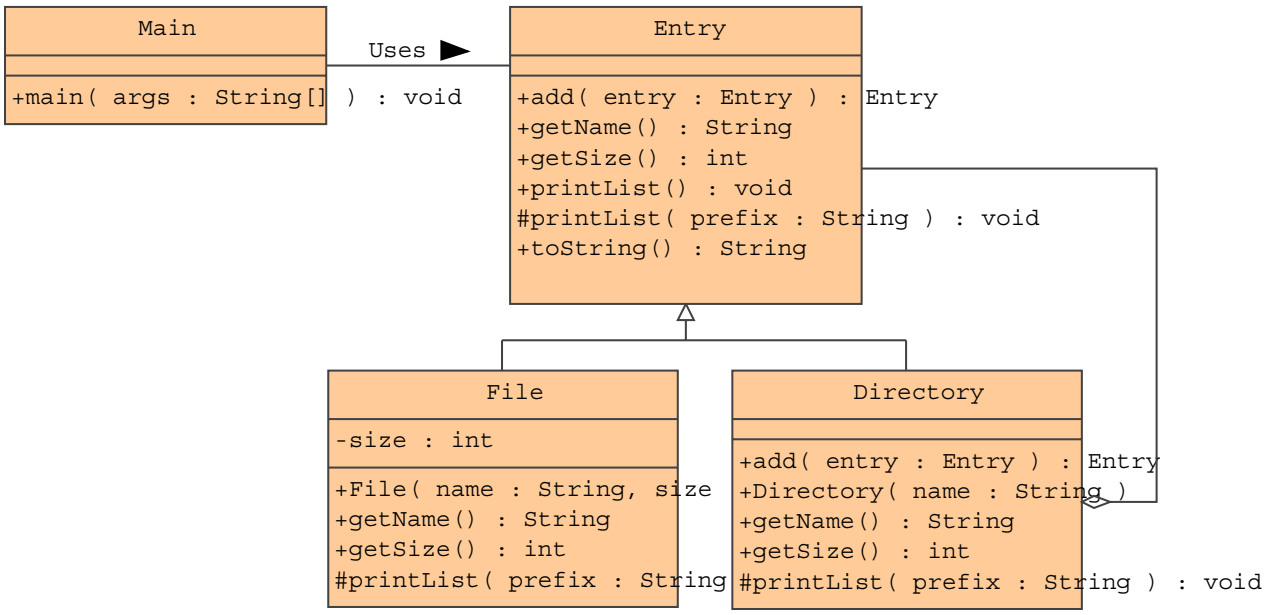

図 5: Composite パターンのクラス図

Directory) を追加することはできない。これを示すための例外

Main サンプル実行用のテストクラス

以下、個々のプログラムを見ていく。

3.2.1 Entry.java

まず、図 6 の `Entry.java` では、add(Entry), printList(), printList(String) というメソッドが定義されている。add はエントリの追加をするためのメソッドである。Directory には File または Directory が追加できるので、そのためのメソッドが必要である。一方、File の場合には add によって Entry を追加することができない。このことを示す方法はいくつかあるが、ここでは Entry クラスで追加を禁じるようにしている。

printList メソッドは 2 つあり、1 つは引数なしのもので、動作としては**何も表示しない**(何もないを表示する)。もう 1 つは抽象メソッドであり、このクラスを継承したクラスで実装を期待されるものである。

また、printList には protected がつけられている。これは継承関係があるクラス以外からのアクセスをさせないためである。継承関係のクラスからはアクセスの必要があるので private にできない。

```
 1 public abstract class Entry {
 2   public abstract String getName();
 3   public abstract int getSize();
 4   public Entry add(Entry entry)
       throws FileTreatmentException {
 5     throw new FileTreatmentException();
 6   }
 7
 8   public void printList(){
 9     printList("");
10   }
11   protected abstract
       void printList(String prefix);
12
13   public String toString(){
14     return getName() + " (" + getSize() + ")";
15   }
16 }
17
```

図 6: `Entry.java`

3.2.2 File.java

File.java は Entry.java を継承し、abstract 宣言されていたものを具象化している。15 行目で System.out.println の引数に this が入っているが、これは、実際に内部では、this.toString() が呼ばれる。File は木でいえば葉に相当する。すなわち、末端であるの

```
 1 public class File extends Entry {
 2   private String name;
 3   private int size;
 4   public File(String name, int size){
 5     this.name = name;
 6     this.size = size;
 7   }
 8   public String getName(){
 9     return name;
10   }
11   public int getSize(){
12     return size;
13   }
14   protected void printList(String prefix){
15     System.out.println(prefix + "/" + this);
16   }
17 }
```

図 7: File.java

で表示としては自身に関することだけ表示すればよい。次の Directory.java と比較してみてほしい。

3.2.3 Directory.java

まず、Vector クラスについて簡単に説明しよう。Vector は可変長な配列と考えられる。すなわち、通常の配列だと使うサイズによってあらかじめ new を施してから使用する。Vector は内部でそのあたりの調整をしてくれると思えば良い。詳しくはマニュアルを参照されたい。

Directory.java も Entry.java を継承し、abstract 宣言されたものを具象化している。こちらは木の節にあたるので、自身には Entry が含まれている可能性がある。自身のサイズは含まれている Entry によって決まる。すなわち、含まれている各 Entry のサイズの合計がこのディレクトリのサイズとなる。自身に含まれている Entry は Vector クラスのオブジェクトに収められている。外部から追加の要求が add メソッドを通してあった場合、内部ではそれを Vector のオブジェクトに追加している。この Vector オブジェクトに追加された要素は、木構造でいえば子ノードである。

さて、getSize ではその Vector (すなわち、各子ノード) を順番に見ていき、それぞれの getSize を呼び出すことでそれらの合計を求めている。各子ノードでは、それが File のオブジェクトであれば、自身の size の値を返し、それが Directory オブジェクトであれば、

```
 1 import java.util.Iterator;
 2 import java.util.Vector;
 3
 4 public class Directory extends Entry {
 5   private String name;
 6   private Vector directory = new Vector();
 7
 8   public Directory(String name){
 9     this.name = name;
10   }
11
12   public String getName(){
13     return name;
14   }
15
16   public int getSize(){
17     int size = 0;
18     Iterator it = directory.iterator();
19     while (it.hasNext()){
20       Entry entry = (Entry)it.next();
21       size += entry.getSize();
22     }
23     return size;
24   }
25
26   public Entry add(Entry entry){
27     directory.add(entry);
28     return this;
29   }
30
31   protected void printList(String prefix){
32     System.out.println(prefix + "/" + this);
33     Iterator it = directory.iterator();
34     while(it.hasNext()){
35       Entry entry = (Entry)it.next();
36       entry.printList(prefix + "/" + name);
37     }
38   }
39 }
```

図 8: Directory.java

自身が持つ各子ノードに対してそれぞれのサイズを計算させ、その合計を返す。

ここで着目すべきは 20 行目で、Entry としてオブジェクトを取り出している。すなわち、その実態が File であるか Directory であるかということを気にする必要がない。実際には Directory か File かのどちらかであるが、この両者は Entry を継承しているため、統一的に Entry で扱い、その getSize メソッドを呼び出すことができる。計算機の内部的には、実際にどちらかであるかによって、適切なメソッドが呼び出される。**Composite パターンの特徴はここにある。**printList もこの仕組みによって再帰的にアクセスされている。実行結果を眺めながら、この printList の動作を追い

かけてみよう。

3.2.4 FileTreatmentException.java

```
1 public class FileTreatmentException
   extends RuntimeException {
2   public FileTreatmentException(){
3   }
4   public FileTreatmentException(String msg){
5     super(msg);
6   }
7 }
```

図 9: FileTreatmentException.java

File に対して Entry の追加処理を行うことはできない。このような処理を行った場合、エラーとして扱いたい。このため、ここでは例外クラスを用意し、そのような扱いがあった場合にその例外を送出するよう Entry.java の 4 行目で扱っている。

これを試すには図 10 の 24 行目あたりに次のようなコードを挿入してみればよい。

```
File err = new File("nakai", 123);
err.add(yuki);
```

このようなコードを挿入してもコンパイラには問題なく通る。しかし、実行時には例外を受け取って終了する。

3.2.5 Main.java

Main.java はあまり説明は要らないと思うが、37 行目で rootdir.printList() という呼び出しをしているが、この main メソッド内で、データ構造に対する繰り返しをしなくてよいということに着目してほしい。

演習 1

式を解析した結果を表す木構造を表現したい。ここでは今回の Composite パターンに合うようにクラスを設計することとする。

- 木の節となる総称的なクラス Node を用意する。Node が持つのは抽象メソッド int eval() のみとする。
- 計算結果を表す木の節として考えられるものは二項演算子を表現するもの (Op) と被演算子 (Num) となるものである。被演算子としては整数しか扱わないとする。
- ここでは具体的な二項演算子としては掛け算 (Mult) と足し算 (Plus) を用意するとする。

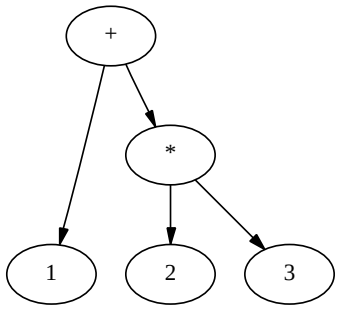

図 11: 1+2*3 に対する木

演習として次を行え。

1. 上述のクラスを具体的に Java で実装せよ。
2. 実装したら、Main クラスを作成し、1+2*3 に対応する木を作成し、そのルートとなる節に対して eval を実行し、結果を求めてみよ。1+2*3 に対応する木を表現すると図 11 のようになる。
3. 別の式を適当に考えてみよう。その式に対応した木を作るように main メソッドを変更し、計算を実行してみよ。(**※注：括弧が入った式も木にしてしまえば、括弧は関係なくなる**)
4. さらに割り算と引き算の演算子も追加し、適当な式を考え、それに対応した木を作るように main メソッドも変更し、実行してみよ。

4 Decorator パターン

decoration はケーキのあの「デコレーション」である。ここで紹介する Decorator パターンは機能を付加するための構造を表している。各クラスの関係は Composite パターンに似ている。両者の違いは、Composite パターンの場合、データ構造を表現するために使われるが、Decorator の場合は機能を表現するために使われる。

```
 1 public class Main {
 2   public static void main(String args[]){
 3     try {
 4       System.out.println
           ("Making root entries ...");
 5       Directory rootdir = new Directory("root");
 6       Directory bindir  = new Directory("bin");
 7       Directory tmpdir  = new Directory("tmp");
 8       Directory usrdir  = new Directory("usr");
 9
10       rootdir.add(bindir);
11       rootdir.add(tmpdir);
12       rootdir.add(usrdir);
13
14       bindir.add(new File("vi", 10000));
15       bindir.add(new File("latex", 20000));
16
17       rootdir.printList();
18
19       System.out.println("");
20       System.out.println
           ("Making user entries...");
21       Directory yuki   = new Directory("yuki");
22       Directory hanako =
                   new Directory("hanako");
23       Directory tomura =
                   new Directory("tomura");
24
25       usrdir.add(yuki);
26       usrdir.add(hanako);
27       usrdir.add(tomura);
28
29       yuki.add(new File("diary.html", 100));
30       yuki.add(new File("Composite.html", 100));
31
32       hanako.add(new File("memo.tex", 300));
33
34       tomura.add(new File("game.doc", 400));
35       tomura.add(new File("junk.mail", 500));
36
37       rootdir.printList();
38     }
39     catch (FileTreatmentException e){
40       e.printStackTrace();
41     }
42   }
43 }
```

図 10: Main.java

Decorator パターンに基づいて実装されたクラスは、基本となるクラスに対し、機能拡張のクラスを用意しておいて、必要に応じて機能を付加するように使われる。

ここではまず、サンプルプログラムを示し、それからイメージをつかんでもらうことにする (図 13、図 14、図 15、図 16、図 17、図 18)。

先に、一連のプログラムをコンパイル、実行するとどうなるか述べておこう。StringDisplay クラスのオブジェクトは文字列を表示する。Border クラスは Display クラスの機能拡張用で、文字列の表示に対し、「飾りをつける」という機能を持っている。ここでは与えられた文字列の前後に (指定した文字で) 飾りをつける SideBorder と与えられた文字列の周りに (指定した文字で) 飾りをつける FullBorder とが用意されている。

基本機能を持っている StringDisplay も拡張機能を持っている Border の継承者たちもどちらも Display の継承者であるため、Java のプログラムとしては Display とみなして同一に扱うことができる。これによって、基本機能のオブジェクトに拡張機能を容易に付加していくことができる。

4.1 Decorator パターンのクラス図

この節の冒頭で述べたように Decorator パターンのクラス図は Composite パターンのクラス図とよく似ている。この形になっていることで、関連する一連のクラス群を同一視できる。

以下、各クラスを個別に見ていこう。

4.1.1 Display.java

```
 1 public abstract class Display {
 2   public abstract int getColumns();
 3   public abstract int getRows();
 4   public abstract String getRowText(int row);
 5   public final void show(){
 6     for(int i = 0; i < getRows(); i++){
 7       System.out.println(getRowText(i));
 8     }
 9   }
10 }
```

図 13: Display.java

このクラスは今回のプログラムにおいて「機能」の抽象化されたものであると考えるとよい。ここでは「表

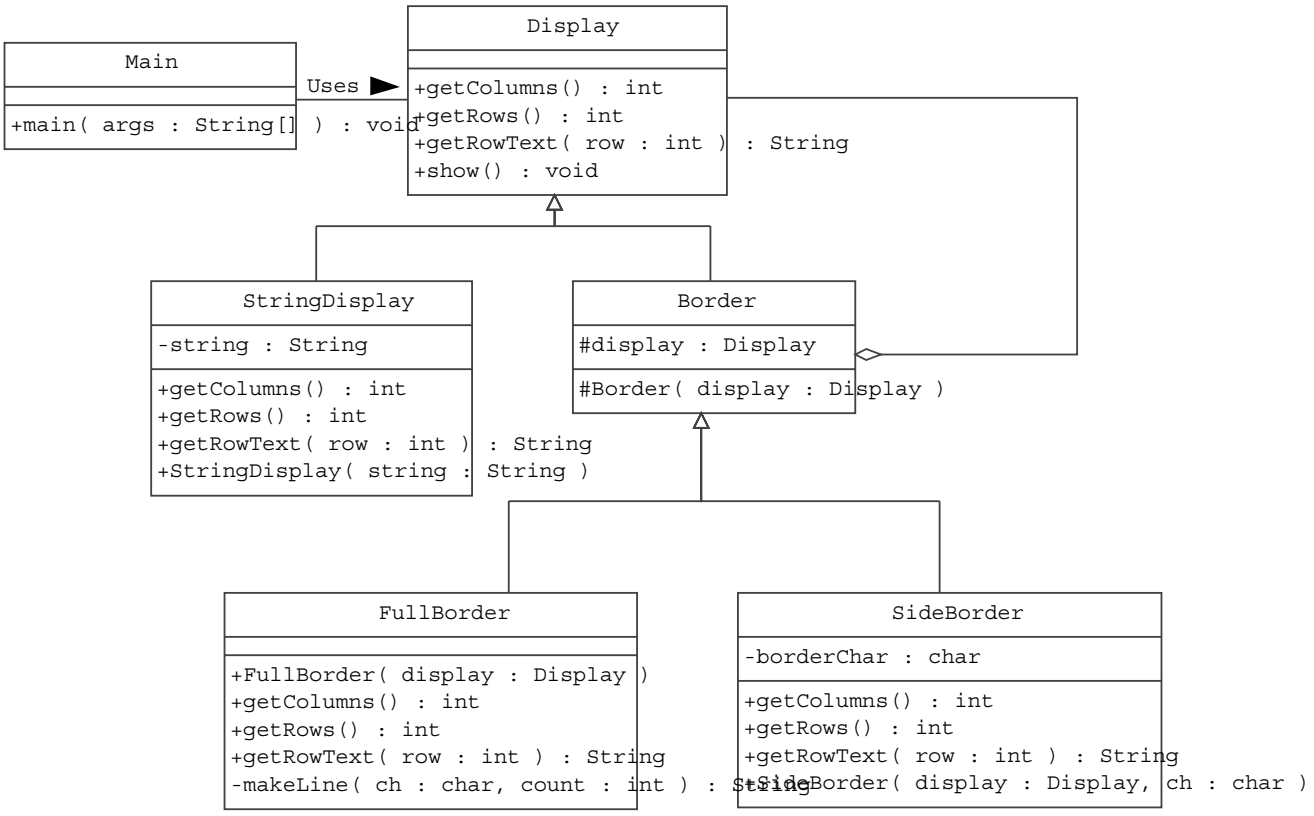

図 12: DecoratorPattern のクラス図

示 (display) する」という機能をクラスにしたわけである。Composite パターンの例における Entry に相当する。

このクラスでは getColumns と getRows が用意されているが、これは機能拡張されることを見越したものである。

4.1.2 StringDisplay.java

StringDisplay クラスは Display クラスを具象化したものであり、今回のプログラムの中では、基本機能を持つクラスとなる。

4.1.3 Border.java

Border クラスは機能拡張用のクラスである。上述したように機能を拡張される側もする側も同じ Display を継承しているところがミソである。

4.1.4 SideBorder.java

このクラスのオブジェクトは、与えられたオブジェクトに対し、前後 (左右というべきか) に飾りをつけるという機能を持っている。

4.1.5 FullBorder.java

FullBorder クラスは、与えられたオブジェクトに対し、その周囲に飾りをつける機能を持っている。

4.1.6 Main.java

さて、ここまでの説明でわかった人にはわかったかもしれないが、わからない人もいたと思われる。とりあえず、この main メソッドの中身を見てから、もう一度、一連のプログラムを眺めて、「なぁるほどぉ」と納得してもらいたい。

```
 1 public class StringDisplay extends Display {
 2   private String string;
 3   public StringDisplay(String string){
 4     this.string = string;
 5   }
 6   public int getColumns(){
 7     return string.getBytes().length;
 8   }
 9
10   public int getRows(){
11     return 1;
12   }
13
14   public String getRowText(int row){
15     if (row == 0){
16       return string;
17     }
18     else {
19       return null;
20     }
21   }
22 }
```

図 14: StringDisplay.java

```
1 public abstract class Border extends Display {
2   protected Display display;
3   protected Border(Display display){
4     this.display = display;
5   }
6 }
```

図 15: Border.java

7 行目では 3 行目で生成したオブジェクトに対し、show メソッドを呼んでいる。この結果、Display の show は StringDisplay の getRowText を使って画面に表示する。ここでは結局与えられた文字列 (Hello, world) が表示される。

4 行目では、SideBorder クラスのオブジェクトを生成しているが、コンストラクタには b1 つまり、StringDisplay のオブジェクトを渡している。これで、StringDisplay に機能の付加をしている。8 行目で show メソッドが呼ばれたとき、まずは SideBorder の getRowText が呼ばれる。SideBorder の getRowText は、コンストラクタで与えられたオブジェクトの getRowText を呼び出し、その結果を加工したものを、呼び出しもとの show メソッドに返す。すなわち、図 16 の 17 行目でコンストラクタでもらったオブジェクトから返ってきた文字列の前後に文字をつけてそれを返している。結果として#Hello, world#が表示される。

```
 1 public class SideBorder extends Border {
 2   private char borderChar;
 3   public SideBorder(Display display, char ch){
 4     super(display);
 5     this.borderChar = ch;
 6   }
 7
 8   public int getColumns(){
 9     return 1 + display.getColumns() + 1;
10   }
11
12   public int getRows(){
13     return display.getRows();
14   }
15
16   public String getRowText(int row){
17     return borderChar + display.getRowText(row)
                         + borderChar;
18   }
19 }
```

図 16: SideBorder.java

演習 2

次のものを作れ ([3] から拝借)。

1. 与えられた文字列の上側と下側に飾り文字をつける UpDownBorder クラス。図 19 に UpDownBorder クラスを用いた main メソッドを示す。その実行結果は次のようになる。

```
Hello, world.
-------------
Hello, world.
-------------
*-------------*
*Hello, world.*
*-------------*
+-----------------+
|/////////////////|
||===============||
||*こんにちは。*||
||===============||
|/////////////////|
+-----------------+
```

2. StringDisplay の代わりに MultiStringDisplay クラスを作れ。MultiStringDisplay クラスは図 20 のように使用されるとする。この実行結果は次のようになる (文字列の後ろ側がそろっている点に注意)。

```
 1 public class FullBorder extends Border {
 2   public FullBorder(Display display){
 3     super(display);
 4   }
 5
 6   public int getColumns(){
 7     return 1 + display.getColumns() + 1;
 8   }
 9
10   public int getRows(){
11     return 1 + display.getRows() + 1;
12   }
13
14   public String getRowText(int row){
15     if (row == 0){
16       return "+" +
              makeLine('-', display.getColumns())
                 + "+";
17     }
18     else if (row == display.getRows() + 1){
19       return "+" +
              makeLine('-', display.getColumns())
                 + "+";
20     }
21     else {
22       return "|" +
               display.getRowText(row - 1)
                + "|";
23     }
24   }
25
26   private String makeLine(char ch, int count){
27     StringBuffer buf = new StringBuffer();
28     for(int i = 0; i<count; i++){
29       buf.append(ch);
30     }
31     return buf.toString();
32   }
33 }
```

図 17: FullBorder.java

```
おはようございます。
こんにちは。
おやすみなさい、また明日。
#おはようございます。       #
#こんにちは。               #
#おやすみなさい、また明日。#
+----------------------+
|おはようございます。       |
|こんにちは。               |
|おやすみなさい、また明日。|
+----------------------+


3
```

[3]がんばったんだけど、LaTeX で全角と半角を混ぜてそろえるのは難しい。画面上の実行結果はちゃんとそろっています。

```
 1 public class Main {
 2   public static void main(String args[]){
 3     Display b1 =
              new StringDisplay("Hello, world.");
 4     Display b2 = new SideBorder(b1, '#');
 5     Display b3 = new FullBorder(b2);
 6
 7     b1.show();
 8     b2.show();
 9     b3.show();
10
11     Display b4 =
12       new SideBorder(
13         new FullBorder(
14           new FullBorder(
15             new SideBorder(
16               new FullBorder(
17                 new StringDisplay
                         ("こんにちは。")
18                                 ),
19               '*'
20               )
21             )
22           ),
23         '/'
24         );
25     b4.show();
26   }
27 }
```

図 18: Main.java

5 Visitor パターン

これまではデータ構造上を渡り歩いてそれぞれの要素 (ノード) で何かをする場合、その要素クラスに行うべき事柄をメソッドとして実装していた。しかし、対象物が持っている構造の表現と、その構造に対する操作を分けておきたい場合が存在する。

わかりやすい (??) 例としてコンパイラの処理の過程を示す。C 言語では、関数を記述するとき次のように書く。

```
int foo(int i){
  変数宣言

  変数の使用を含んだ文
}
```

コンパイラは入力としてソースプログラムを受け取ると、まず、字句解析と構文解析を行う。字句解析とは文字列から意味のある単語を切り出す作業である。上の例では、int, foo, (, int, i,), …というように

```
 1 public class UDMain {
 2   public static void main(String args[]){
 3     Display b1
         = new StringDisplay("Hello, world.");
 4     Display b2 = new UpDownBorder(b1, '-');
 5     Display b3 = new SideBorder(b2, '*');
 6
 7     b1.show();
 8     b2.show();
 9     b3.show();
10
11     Display b4 =
12       new FullBorder(
13         new UpDownBorder(
14           new SideBorder(
15             new UpDownBorder(
16               new SideBorder(
17                 new StringDisplay
                     ("こんにちは。"),
18                 '*'
19                 ),
20               '='
21               ),
22             '|'
23           ),
24         '/'
25           )
26         );
27     b4.show();
28   }
29 }
```

図 19: UDMain.java

```
public class Main {
  public static void main(String args[]){
    MultiStringDisplay md
              = new MultiStringDisplay();
    md.add("おはようございます。")
    md.add("こんにちは。")
    md.add("おやすみなさい、また明日。")
    md.show();

    Display d1 = new SideBorder(md, '#');
    d1.show();

    Display d1 = new FullBorder(md);
    d2.show();
  }
}
```

図 20: Main.java

切り出される。構文解析とは、切り出した単語の並びが文法に合っているかチェックする処理であり、結果として木構造を作り出すことが多い。上のような関数 (func) の宣言に対しては図 21 のような解析結果の木 (解析木という) が得られる。実際には文法に沿って木が作られるのだが、細かいところは省略している (木なんだ、というイメージだけつかんでもらえればよい)。decl(宣言)、stmts(実行文)、の下の三角はサブツリー (部分木) を表している。

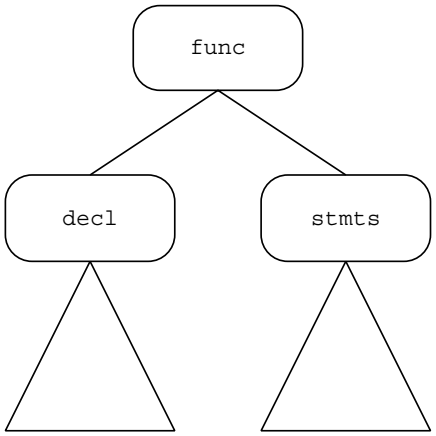

図 21: C 言語の関数宣言の構文解析木 (矢印は訪問順の例)

さて、コンパイラの作業としては、この木をルートからはじめて、深さ優先で左側から探索する。すなわち、関数、宣言、その下の部分木、宣言 (戻り)、関数 (戻り)、実行文、その下の部分木、実行文 (戻り)、関数 (戻り)、という順序である。

構文解析の次には意味解析を行う。意味解析は解析木の各ノードを順番に訪問し、情報を集めながら、二重宣言がないか、未宣言変数を使っていないかなどのチェックを行う。このとき、図 21 のような訪問をすると、まず、宣言の下の部分木の内部を訪問するときには、宣言された変数を記号表に登録していく。その次に実行文のところでは、変数名が現れるたびにその変数が宣言されているか、宣言されたとおりに使用されているかをチェックする。

この一連の訪問が無事に終われば、どんな変数が宣言されていたか、という情報が集まったことになる。これに基づいて次は (中間) コード生成を行う。ここも詳細を省くが、要は機械が実行可能なコードへの変換

を次の訪問で行うわけである。当然、もとのソースプログラムに書いている内容が狂わないように変換を行う必要がある。

さて、ここまでの話をまとめると、コンパイラは次のように処理をする。

1. 字句・構文解析を行い、解析木を作る
2. 解析木を訪問し、意味解析を行う
3. さらに解析木を訪問し、コード生成を行う

ここでは 2 回解析木の訪問を行った。1 回目と 2 回目では処理する内容は異なっている。このため、Composite パターンの例で示したように各要素 (ノード) に処理を記述しておくやり方ではなく、1 回目の処理用クラス、2 回目の処理用クラスというように機能ごとにクラスを用意し、必要に応じてそのクラスを使用して処理をするように設計しておくほうが、あとから 3 回目用のクラスを付け足すなどのことが容易にできる。Composite パターンの例のように各要素クラスに処理を埋め込んでしまった場合、変更をするにはそのクラスのソースファイルを書き換える必要が出てくる。そうではなく、機能ごとにクラスを分け、必要に応じて使えるほうが良い。

このようにデータ構造とそれに対する処理を分離しておくクラスの設計には、Visitor パターンを適用できる。

非常にややこしく、理解するまでに多少時間がかかると思われるが、ある意味、オブジェクト指向らしいクラスの関連をこのパターンから感じ取ってほしい。

6 Visitor パターンの例

さて、具体的な例を示そう。例によって結城さんの本 [3] からサンプルを拝借する。

```
1 public interface Acceptor {
2  public abstract void accept(Visitor v);
3 }
```

図 22: Acceptor.java

今回のサンプルは、Composite パターンのところで紹介した、File と Directory の題材がベースになっている。入力する場合は、それらのファイルをコピーしてきて、変更していくのがよい。

```
 1 import java.util.Iterator;
 2
 3 public abstract class Entry
    implements Acceptor {
 4   public abstract String getName();
 5   public abstract int getSize();
 6   public Entry add(Entry entry)
      throws FileTreatmentException {
 7     throw new FileTreatmentException();
 8   }
 9
10   public Iterator iterator()
      throws FileTreatmentException {
11     throw new FileTreatmentException();
12   }
13
14   public String toString(){
15     return getName() + " (" + getSize() + ")";
16   }
17 }
```

図 23: Entry.java

```
 1 public class File extends Entry {
 2   private String name;
 3   private int size;
 4
 5   public File(String name, int size){
 6     this.name = name;
 7     this.size = size;
 8   }
 9
10   public String getName(){
11     return name;
12   }
13
14   public int getSize(){
15     return size;
16   }
17
18   public void accept(Visitor v){
19     v.visit(this);
20   }
21 }
```

図 24: File.java

Visitor パターンでは、訪問の対象となるデータ構造を表すクラスと訪問者を示すクラスからなる。訪問の対象となるデータ構造を表すクラスは、訪問を受け付ける (受け入れる) という意味で Acceptor というインタフェース (図 22) を実装している (図 23)。このた

```
 1 import java.util.Iterator;
 2 import java.util.Vector;
 3
 4 public class Directory extends Entry {
 5   private String name;
 6   private Vector dir = new Vector();
 7
 8   public Directory(String name){
 9     this.name = name;
10   }
11
12   public String getName(){
13     return name;
14   }
15
16   public int getSize(){
17     int size = 0;
18     Iterator it = dir.iterator();
19     while (it.hasNext()){
20       Entry entry = (Entry)it.next();
21       size += entry.getSize();
22     }
23     return size;
24   }
25
26   public Entry add(Entry entry){
27     dir.add(entry);
28     return this;
29   }
30
31   public Iterator iterator(){
32     return dir.iterator();
33   }
34
35   public void accept(Visitor v){
36     v.visit(this);
37   }
38 }
```

図 25: Directory.java

```
1 public class FileTreatmentException
   extends RuntimeException {
2   public FileTreatmentException(){
3   }
4   public FileTreatmentException(String msg){
5     super(msg);
6   }
7 }
```

図 26: FileTreatmentException.java

め、抽象クラスである Entry を継承したクラスでは、accept メソッドを実装している (図 24, 図 25)

訪問者は訪問するデータ構造を知っている必要がある。この例では、データ構造として具体的にオブジェクトになるのは、File クラスか Directory クラスのインスタンスである。そこで、File クラスのインスタンスだったらこうする、Directory クラスのインスタンスだったらこうする、ということをメソッドとして実装する。Visitor クラスはこのための抽象メソッドである。

```
1 public abstract class Visitor {
2   public abstract void visit(File file);
3   public abstract void visit(Directory directory);
4 }
```

図 27: Visitor.java

```
 1 import java.util.Iterator;
 2
 3 public class ListVisitor extends Visitor {
 4   private String currentdir = "";
 5
 6   public void visit(File file){
 7     System.out.println(currentdir + "/" + file);
 8   }
 9
10   public void visit(Directory directory){
11     System.out.println(currentdir + "/"
                                    + directory);
12     String savedir = currentdir;
13     currentdir = currentdir + "/" + directory;
14     Iterator it = directory.iterator();
15     while(it.hasNext()){
16       Entry entry = (Entry)it.next();
17       entry.accept(this);
18     }
19
20     currentdir = savedir;
21   }
22 }
```

図 28: ListVisitor.java

この例の具体的な訪問時の作業の実装は ListVisitor クラスに実装されている。受け取ったオブジェクトが File クラスのインスタンスなら、visit(File file) のメソッドが起動される。受け取ったオブジェクトが Directory クラスのインスタンスなら、visit(Directory directory) のメソッドが起動される。if 文などを用いなくとも引数の型の違いで起動するメソッドが勝手に選択されるというオブジェクト指向の特性を利用している点に着目しよう。

さて、受け取ったオブジェクトが File クラスのイン

```
 1 public class Main {
 2   public static void main(String args[]){
 3     try {
 4       System.out.println("Making root entries ...");
 5       Directory rootdir = new Directory("root");
 6       Directory bindir  = new Directory("bin");
 7       Directory tmpdir  = new Directory("tmp");
 8       Directory usrdir  = new Directory("usr");
 9
10       rootdir.add(bindir);
11       rootdir.add(tmpdir);
12       rootdir.add(usrdir);
13
14       bindir.add(new File("vi", 10000));
15       bindir.add(new File("latex", 20000));
16
17       rootdir.accept(new ListVisitor());
18
19       System.out.println("");
20       System.out.println("Making user entries...");
21       Directory yuki   = new Directory("yuki");
22       Directory hanako = new Directory("hanako");
23       Directory tomura = new Directory("tomura");
24
25       usrdir.add(yuki);
26       usrdir.add(hanako);
27       usrdir.add(tomura);
28
29       yuki.add(new File("diary.html", 100));
30       yuki.add(new File("Composite.html", 100));
31
32       hanako.add(new File("memo.tex", 300));
33       tomura.add(new File("game.doc" 400));
34       tomura.add(new File("junk.mail'', 500));
35
36       rootdir.accept(new ListVisitor());
37     }
38     catch (FileTreatmentException e){
39       e.printStackTrace();
40     }
41   }
42 }
```

図 29: Main.java

スタンスなら、ファイルのパスを表示して終了する。受け取ったオブジェクトが Directory クラスのインスタンスなら、さらに自身がファイルやディレクトリを持っているので、それらの処理も行う必要がある。実際にはそれらに処理を任せているのだが (図 28 の 14〜18 行目)。

Visitor パターンの実行時のイメージは次のような感じである。あくまでもイメージ。

- まず、一人訪問者がいる。
- やることリストには A ならどうする、B ならどうする、と職種毎にやることが書かれている。
- 訪問される側はたとえば、ある企業の各部屋にいる人である (これから訪問されるのでみんな各自の部屋にいるとする)。人には職種があり、ここでは職種は A, B, C となっているとする。
- 訪問者は廊下の端の部屋を訪問する。
- 訪問された側は訪問者が見せるやることリストに目を通し、自身がやるべき仕事を、それを見ながら始める。(リストには「訪問者に次に行くところを指示せよ」と書いてある場合もある)

これによって、訪問者は、ある部屋を訪問し、その部屋の人に仕事をしてもらい、次にどこに行くか指示を受けて、また別の部屋を訪問し、と繰り返す。訪問の順番が固定の場合、訪問の指示を含めないようなやりかたもある。

7 演習問題の回答

演習 1

1. Node 関連のクラスを図 30 に示す。
2. Main クラスを図 31 に示す。
3. 省略
4. 省略

演習 2

1. 解答例を図 32 に示す。
2. 解答例を図 33 に示す。

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.

```
 1 abstract class Node {
 2   abstract public int eval();
 3 }
 4
 5 abstract class Op extends Node {
 6   Node left, right;
 7
 8   abstract public int eval();
 9 }
10
11 class Plus extends Op {
12   Plus(Node n1, Node n2){
13     left  = n1;
14     right = n2;
15   }
16
17   public int eval(){
18     return left.eval() + right.eval();
19   }
20 }
21
22 class Mult extends Op {
23   Mult(Node n1, Node n2){
24     left  = n1;
25     right = n2;
26   }
27
28   public int eval(){
29     return left.eval() * right.eval();
30   }
31 }
32
33 class Digit extends Node {
34   private int val;
35
36   Digit(int val){
37     this.val = val;
38   }
39
40   public int eval(){
41     return val;
42   }
43 }
```

図 30: Expression.java

```
 1 class ExMain {
 2   public static void main(String args[]){
 3     Node tmp1, tmp2, tmp3, tmp4, tmp5;
 4
 5     tmp1 = new Digit(1);
 6     tmp2 = new Digit(2);
 7     tmp3 = new Digit(3);
 8
 9     tmp4 = new Mult(tmp2, tmp3);
10     tmp5 = new Plus(tmp1, tmp4);
11
12     System.out.print("1+2*3 = ");
13     System.out.println(tmp5.eval());
14   }
15 }
```

図 31: ExMain.java

```
 1 public class UpDownBorder extends Border {
 2   private char borderChar;
 3   public UpDownBorder(Display display,
                         char ch){
 4     super(display);
 5     this.borderChar = ch;
 6   }
 7
 8   public int getColumns(){
 9     return display.getColumns();
10   }
11
12   public int getRows(){
13     return 1 + display.getRows() + 1;
14   }
15
16   public String getRowText(int row){
17     if (row == 0){
18       return makeLine(borderChar,
                         display.getColumns());
19     }
20     else if (row == display.getRows() + 1){
21       return makeLine(borderChar,
                         display.getColumns());
22     }
23     else {
24       return display.getRowText(row - 1);
25     }
26   }
27
28   private String makeLine(char ch, int count){
29     StringBuffer buf = new StringBuffer();
30     for(int i = 0; i<count; i++){
31       buf.append(ch);
32     }
33     return buf.toString();
34   }
35 }
```

図 32: UpDownBorder.java

```
 1 import java.util.Iterator;
 2 import java.util.Vector;
 3
 4 public class MultiStringDisplay
                            extends Display {
 5   private Vector ms = new Vector();
 6   private int maxcol = 0;
 7
 8   public void add(String s){
 9     ms.add(s);
10     if (maxcol < s.getBytes().length){
11       maxcol=s.getBytes().length;
12     }
13   }
14
15   public int getColumns(){
16     return maxcol;
17   }
18
19   public int getRows(){
20     return ms.size();
21   }
22
23   public String getRowText(int row){
24     String s = (String)ms.get(row);
25     int length = s.getBytes().length;
26     StringBuffer sb = new StringBuffer(s);
27     for(int i=0; i<maxcol-length; i++){
28       sb.append(' ');
29     }
30     return sb.toString();
31   }
32 }
```

図 33: MultiStringDisplay.java