応用プログラミングIII [Q2] レポート

C0115114 菅野路哉

2016 年 6 月 9 日

1 目的

デザインパターンの一種である FactoryMethod について、課題を通して理解する。FactoryMethod を用いたプログラムを作成し、どのような場面で利用されているのかを調べ、具体例をあげて説明する。

2 課題

図 1 に示すプログラムを完成させる。このプログラムは、ゲームのキャラクターに名前をつけるときに利用されるといったものである。**リスト 6** では、`RetroFactory` で作成した 2 つのゲームキャラクターに対して `show()` メソッドを呼び出し、キャラクターの名前を表示している。**リスト 8** では、設定した名前の英数字が全て大文字に変わっているのが分かる。

次に、作成した `en06.framework` パッケージを用いて、`en06.front` パッケージに `OldFactory` クラスを作成する。`OldFactory` メソッドを用いた場合の実行結果を**リスト 9** に示す。`OldFactory` クラスを用いたキャラクタは、名前の英数字が全て大文字に変わったうえ、名前の長さが 4 文字に短縮される。

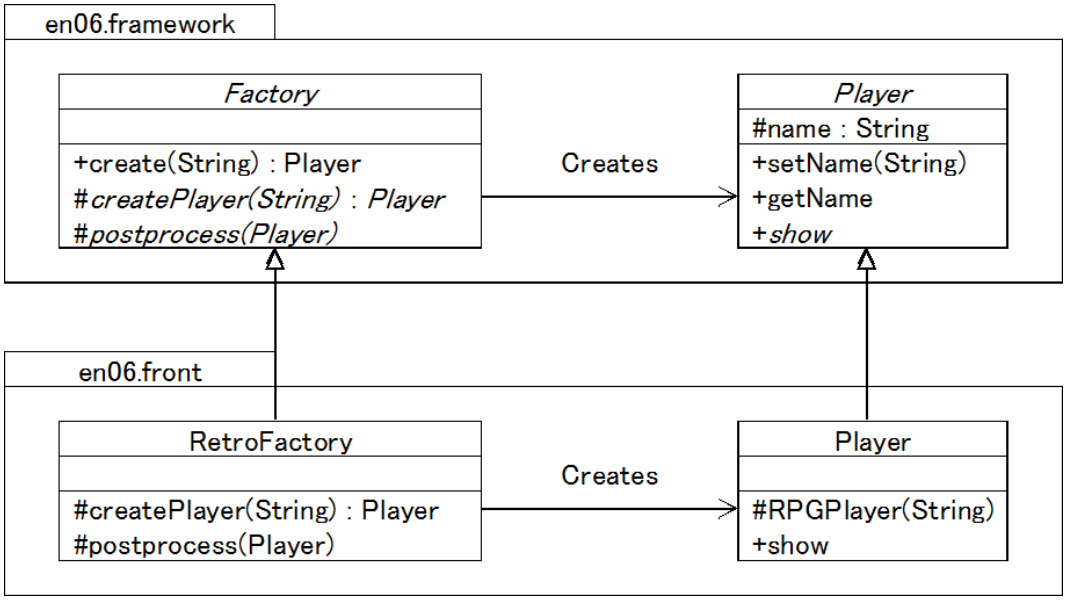

図 1　作成するクラス図

3 ソースコード

3.1 Factory

リスト 1　Factory.java

```
1  package en06.framework;
2
3  public abstract class Factory{
4    public final Player create(String name){
5      Player player = createPlayer(name);
6      postprocess(player);
7      return player;
8    }
9
10   protected abstract Player createPlayer(String name);
11   protected abstract void postprocess(Player player);
12 }
```

3.2 Player

リスト 2　Player.java

```
1  package en06.framework;
2
3  public abstract class Player{
4    protected String name;
5
6    public void setName(String name){
7      this.name = name;
8    }
9
10   public String getName(){
11     return name;
12   }
13
14   public abstract void show();
15 }
```

3.3 RetroFactory

リスト 3　RetroFactory.java

```
1  package en06.front;
2
3  // imports//{{{
4  import en06.framework.Factory;
5  import en06.framework.Player;//}}}
6  public class RetroFactory extends Factory{
7    @Override
8    protected Player createPlayer(String name){
9      return new RPGPlayer(name);
10   }
11
12   @Override
13   protected void postprocess(Player player){
14     player.setName(player.getName().toUpperCase());
15   }
16 }
```

3.4 RPGPlayer

リスト 4　RPGPlayer.java

```
1  package en06.front;
2
3  // imports//{{{
4  import en06.framework.Player;//}}}
5  public class RPGPlayer extends Player{
6    protected RPGPlayer(String name){
7      setName(name);
8    }
9
10   @Override
11   public void show(){
12     System.out.printf("NAME: %s\n", name);
13   }
14 }
```

3.5 OldFactory

リスト 5　OldFactory.java

```
1  package en06.front;
2
3  // imports//{{{
4  import en06.framework.Factory;
5  import en06.framework.Player;//}}}
6  public class OldFactory extends Factory{
7    @Override
8    protected Player createPlayer(String name){
9      return new RPGPlayer(name);
10   }
11
12   @Override
13   protected void postprocess(Player player){
14     player.setName(player.getName().toUpperCase().substring(0, 4));
15   }
16 }
```

3.6 実行ファイル

リスト 6　Main.java

```
1  package en06;
2
3  // imports//{{{
4  import en06.framework.Factory;
5  import en06.framework.Player;
6  import en06.front.RetroFactory;//}}}
7  public class Main{
8    public static void main(String[] args){
9      Factory factory = new RetroFactory();
10     Player p1 = factory.create("いのうえあきふみ");
11     Player p2 = factory.create("akifumi␣inoue");
12     p1.show();
13     p2.show();
14   }
15 }
```

リスト 7　Main2.java

```
1  package en06;
2
3  // imports//{{{
4  import en06.framework.Factory;
5  import en06.framework.Player;
6  import en06.front.OldFactory;//}}}
7  public class Main2{
8    public static void main(String[] args){
9      Factory factory = new OldFactory();
10     Player p1 = factory.create("いのうえあきふみ");
11     Player p2 = factory.create("akifumi␣inoue");
12     p1.show();
13     p2.show();
14   }
15 }
```

4 実行結果

リスト 8　Main.java の実行結果

```
NAME: い の う え あ き ふ み
NAME: AKIFUMI INOUE
```

リスト 9　Main2.java の実行結果

```
NAME: い の う え
NAME: AKIF
```

5 解説

リスト 1 では、Player を生成する Factory の抽象クラスを定義した。このクラスは、createPlayer メソッドによって、Player オブジェクトを生成し、postprocess メソッドによって、生成された Player オブジェクトに対して追加処理を行い、それを生成物として返すという流れを用いてインスタンスの生成を行う。インスタンスの生成を行う際は、create メソッドを用いる。

リスト 2 では、Factory によって生成されるオブジェクトを定義した。このクラスに定義されている setName メソッドは、引数にキャラクターの名前を取り、その名前をフィールドの name に格納する。getName メソッドでは、フィールドの name に格納されている情報を戻り値とする。show メソッドでは、何らかの情報を表示させることを目的に抽象メソッドとして定義した。

リスト 3 では、Factory クラスの実装を行った。このクラスは、createPlayer メソッドで生成するオブジェクトの定義と、postprocess メソッドで行う追加処理の定義を行った。createPlayer メソッドでは、RPGPlayer の生成のみを行う。postprocess メソッドでは、キャラクターの名前を大文字に変換する処理を行う。

リスト 4 では、Player クラスの実装を行った。このクラスは、コンストラクタの引数にキャラクター名を定義してあり、Player クラスに実装してある setName メソッドを用いて、name フィールドに名前を格納する。次に、show メソッドの実装を行った。show メソッドでは、Name:　の表示を付与しつつ、キャラクターの名前を表示する。

リスト 5 では、Factory クラスの実装を行った。このクラスは、createPlayer メソッドで生成するオブジェクトの定義と、postprocess メソッドで行う追加処理の定義を行った。createPlayer メソッドでは、RPGPlayer の生成のみを行う。postprocess メソッドでは、キャラクターの名前を大文字に変換する処理を行い、先頭 4 文字のみにする処理を行う。

リスト 6 では、RetroFactory を用いたプログラムの実行を行った。初めに、RetroFactory のインスタンスを生成し、そのインスタンスから、create メソッドを使用してキャラクターのインスタンスを生成する。最後に、show メソッドを用いてキャラクターの情報を表示する。

リスト 7 では、OldFactory を用いたプログラムの実行を行った。リスト 6 から変更すべき点は、6 行目で import しているものの変更と、9 行目で作成するインスタンスの変更のみである。

6　考察・まとめ

FactoryMethod は、インスタンス生成時に、複雑な同様の処理が必要になる場合に使用される。この複雑な処理の手順を定義しておけば、インスタンス生成時には、この処理を記す必要が無くなり、安全性が保証される。

FactoryMethod が使用されている例を挙げると、java 内で FactoryMethod は、javaScript を使用する際に必要となる ScriptEngine の作成に使用されている。この場合、Factory に当たる機能は、ScriptEngineFactory と、ScriptEngineManager の二種がある。ScriptEngine を生成するための Factory 機能が ScriptEngineFactory であり、ScriptEngineFactory を生成するための Factory 機能が ScriptEngineManager となっている。ScriptEngine のインスタンス生成時には、ScriptEngineManager の引数に言語名を渡して getEngineByName メソッドを呼び出す。この時に、ScriptEngine の Factory の生成や、その Factory の存在確認、null の確認、スコープの設定、Exception の生成が定義されているため、それらを意識する必要が無くなっている。