デザインパターン

オブジェクトの生成に関するパターン

Abstract Factory

関連する部品(オブジェクト群)をまとめて一つのシステムを生成するための手順を抽象化

Builder

元となる専用のクラスを用意し、複数のオブジェクトを生成

Factory Method

インスタンス生成のための枠組みと、インスタンス生成のクラスを分離

クラスを増やし、機能を拡張する際に使用

Prototype

インスタンスをコピーして新しいインスタンスを作成(クローンの作成)

同じクラスで複数のオブジェクトを生成する際に作業を効率化

Singleton

1 つのクラスから 1 つのインスタンスだけを生成するように制限

Factory Method の例

```
1  abstract class Product {
2      public abstract void method ();
3  }
4  abstract class Creator {
5      protected abstract Product factoryMethod(String str);
6      public final Product create(String str) {
7          Product p = factoryMethod(str);
8          return p;
9      }
10 }
11 class ConcreteProduct extends Product {
12     private String str;
13     public ConcreteProduct(String str) {
14         this.str = str;
15     }
16     //@Override
17     public void method() {
18         System.out.println("Hello " + str + "!");
19     }
20 }
```

抽象的な枠組み(製品)

抽象的な枠組み(製品の作成)

具体的な機能拡張(製品)

```
21
22  class ConcreteCreator extends Creator {
23      //@Override
24      protected Product factoryMethod(String str) {
25          return new ConcreteProduct(str);
26      }
27  }
28
29  public class User {
30      public static void main(String[] args) {
31          // 製品の作成者を生成
32          Creator creator = new ConcreteCreator();
33
34          // 製品の作成
35          Product java = creator.create("Java");
36          Product cpp = creator.create("C++");
37          Product cs = creator.create("C#");
38
39          // 処理の呼び出し
40          java.method();
41          cpp.method();
42          cs.method();
43      }
44  }
45
46  // 実行結果
47  Hello Java!
48  Hello C++!
49  Hello C#!
50
```

具体的な機能拡張(製品の作成)

オブジェクトの振る舞いに関するパターン

Chain of Responsibility

あるクラスのオブジェクトが処理可能なら処理を行い、処理不可の場合は他のクラスのオブジェクトに送って処理を実行

Command

「マウスをクリック」「キーを押す」のような命令をインスタンスという「モノ」で管理

Interpreter

プログラムをミニ言語に分け、そのミニ言語を「通訳」するプログラムを作成

変更が必要な場合はミニ言語を書き換える

Iterator

複数のオブジェクトを順番に指し示し、全体をスキャンしていく処理を行う

Mediator

「相談役」が複雑なオブジェクトの状態を把握し、適切な判断と支持を行う

Memento

インスタンスの状態を表す役割を設け、インスタンスの状態の保存と復元を行う

Observer

if 文を利用することなく、状態変化に対応した処理を実行

State

状態をクラスとして表現し、クラスを切り替えることで状態の変化を表す

Strategy

アルゴリズムの実装部分が交換可能で、変更が容易

Template Method

スーパークラスで処理の枠組みを定め、サブクラスでその具体的な内容を定義する

Visitor

データ構造と処理を分離

データ構造を渡り歩く「訪問者」を表すクラスを用意し、そのクラスが処理を実施

新しい処理を追加したい場合は新しい「訪問者」を作成

データ構造側は必要に応じて「訪問者」を受け入れる

Template Method の例

```
51  // 抽象クラス
52  abstract class AbstractClass {
53      protected abstract void method();
54      public final void templateMethod() {
55          method();
56      }
57  }
58  // 具象クラス
59  class ConcreteJavaClass extends AbstractClass {
60      //@Override
61      protected void method() {
62          System.out.println("Hello Java!");
```

```
63      }
64  }
65
66  // 具象クラス
67  class ConcreteCppClass extends AbstractClass {
68      //@Override
69      protected void method() {
70          System.out.println("Hello C++!");
71      }
72  }
73
74  public class User {
75
76      public static void main(String[] args) {
77          // インスタンスの生成
78          AbstractClass java = new ConcreteJavaClass();
79          AbstractClass cpp = new ConcreteCppClass();
80
81          // 処理の呼び出し
82          java.templateMethod();
83          cpp.templateMethod();
84      }
85
86  }
87
88  実行結果
89  Hello Java!
90  Hello C++!
```