オブジェクト指向プログラミング

第12,13回：デザインパターン#1,2

知識科学教育研究センター
金井 秀明

デザインパターン

デザインパターン#1

- プログラミングの熟練者は，
  - プログラミングをしていると，前に同じようなソースを書いたことがあるなと思うことがある．
  - それを「パターン」として蓄積し，そのパターンを使って，次のソースに活かしている．

「パターン」をテンプレートとして整理したものを「デザインパターン」という．

デザインパターン#2

- ソフトウェア設計で，よく使用される特徴的な構造や機能を抽出してパターン化したもの
- GoF
  - Erich Gammaらによって，整理されたデザインパターン
  - 23個のパターン
  - “Design Patterns:Elements of Reusable Object-Oriented Software,” Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
    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（1999年）

使用例

- お手本（再利用）
  - ただし，「どうい手順でまねすればいいのか」についての定石はない
- プログラムの説明
  - 他の人と設計への理解が共有できる.
  - それにより，再利用性の高い設計ができる.

GoFデザインパターン

- 生成に関するパターン
  - Factory Method, Abstract Factory, Builder, 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
  - Cain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor, Template Method

授業では,

- 同一視
  - Composite：容器の中身の同一視
    - Iterator：１つ１つ数え上げる.
- 分けて考える
  - Strategy：アルゴリズムをごっそり切り替える

授業では,

- サブクラスにまかせる
  - Template Method：具体的な処理をサブクラスにまかせる
  - Factory Method: インスタンス作成をサブクラスにまかせる
- 状態を管理する
  - State：状態をクラスとして表現する
    - インスタンスを作る（Singleton：たった１つのインスタンス）

Composite

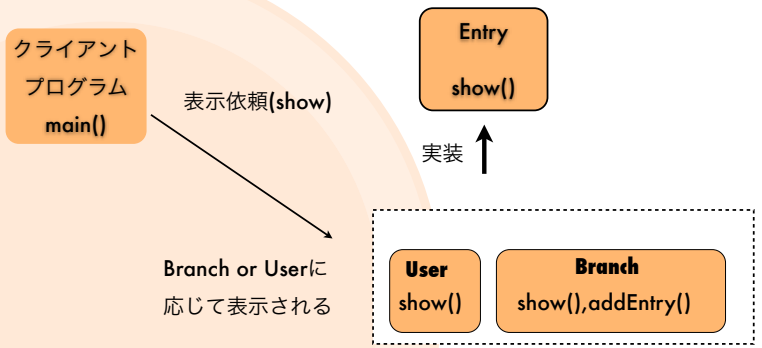
Composite#1

- 目的
  - 階層構造をなすオブジェクト全体に再帰的に一連の操作を行いたい.
- 効果
  - 階層構造をなすオブジェクト全体に，１回のメソッド呼び出しで一連の操作を再帰的に実行できる.
  - １個のオブジェクトにも階層構造全体にも同様にアクセスできる．プログラムがすっきりする.

例

- ある組織を構成している部門とそこに属している人の名前を表示するプログラム
  - 部門や人に同じインタフェースを持たせ，部門（枝），人（葉）なのか区別せずに同じメソッドで呼びだす（Compositeパターン）

例



Iterator

Iterator#1

- 目的
  - ごちゃごちゃと蓄積されたオブジェクト群にアクセスした.
- 効果
  - オブジェクト群がどのような構造で保持されているかに関係なく，シンプルに統一されたインタフェースでアクセスできる.

Iterator#2

- 「順番にオブジェクトを取り出すメソッド」が「Iteratorパターン」である.
  - 多数のオブジェクト群を各々異なる構造で保持しているクラスが，Iteratorをimplementsすれば，オブジェクト群を取り出すには同じメソッドを呼ぶだけですむ.

Iterator#3

- Javaの標準API（java.util.Iterator インタフェース）がある.

```
public interface java.util.Iterator {
    boolean hasNext();
    Object next()
}
```

  - **hasNext():** 次があるかどう確認
  - **next():** 次の要素を返す

例

```
import java.util.*;
class Iterator_sample {
 public static void main(String[] args) {
   Collection list = new ArrayList();
   list.add(”東京”);
   list.add(”ロンドン”);
   list.add(”パリ”);

   Iterator iter = list.iterator();
   while(iter.hasNext()) {
     String item = (String)iter.next();
     System.out.println(item);
   }
 }}
```


Strategy

Strategy#1

- 目的
  - いろいろなアルゴリズム交換しながらプログラムを実行したい
- 効果
  - プログラムの実行中に，その中で使われるアルゴリズムを簡単に切り替えることができる.

Strategy#2

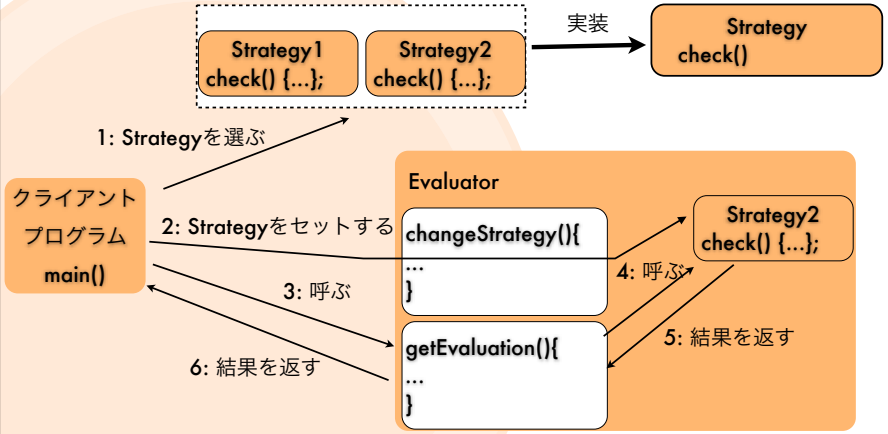
- アルゴリズムの処理部分だけを別のクラスとして，「Strategyクラス」をつくる.
- ストラテジの１つをクライアントプログラムに設定することで，クライアントプログラムのコードを修正することなく，アルゴリズムを交換できる.

例：成績判定

- 中間試験の得点と期末試験の得点から，最終成績の判定をするプログラム
  - 最終成績の判定アルゴリズム（Strategyクラス）
    - Strategy1:中間試験と期末試験の得点の平均得点を評価する．
    - Strategy2:中間試験，期末試験のうち高い得点を評価する．

Strategy/Main.java

例



Template Method

Template Method#1

- 目的
  - 大きな処理を部分的に変更できるようにする．
- 効果
  - ある１つの大きな処理を複数のステップに分解し，ステップの実行順序は守りつつ，各ステップごとに処理内容を変更できる．

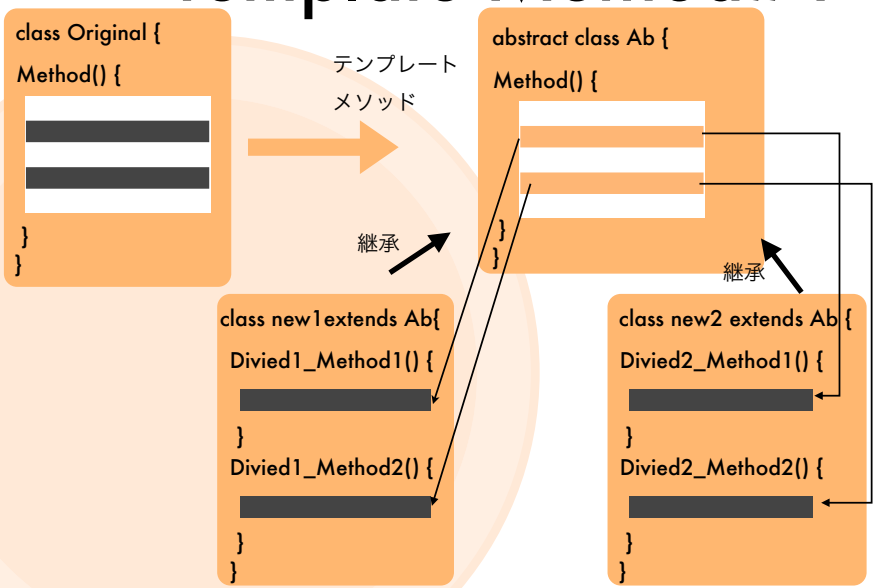
Template Method#2

- 大きなメソッドを複数のメソッド（処理ステップ）に分解する.
- 分解したメソッドを必要に応じて，オーバライドし，一部だけ変更を可能とする.
- 元のメソッドは，分解されたメソッドを順番に呼び出すだけのメソッド（Templateメソッド）になる.

Template Method#3

- 分解されたメソッドのうち，変更することが前提のメソッドは，abstractメソッドする.
  - サブクラス側で必ず実装しなければならない部分を明確にできる.
  - 元のクラスは，abstractクラスとして定義される.

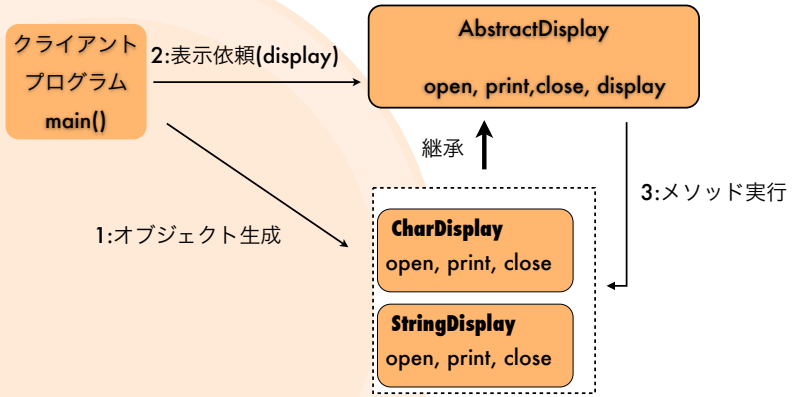
Template Method#4



例

- 文字や文字列を５回繰り返しで表示するプログラム
  - 表示処理の部分Display()（Template Method）
  - 分解されたメソッドの実装が書かれたクラス CharDisplay, StringDisplay

例



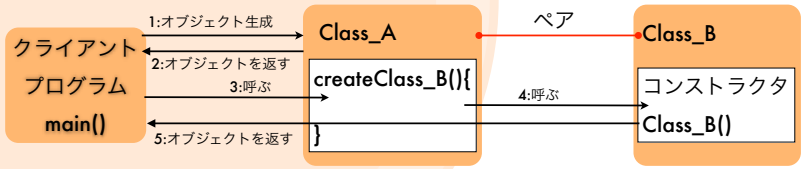
Factory Method

Factory Method#1

- 目的
  - 異なるサブクラスを必ずペアに使うようにしたい.
- 効果
  - どのサブクラスとサブクラスを組合わせて使えばいいのか明確になる.
  - オブジェクト生成の枠組みと，実際のオブジェクト生成のクラスを分けることで，より柔軟に生成するオブジェクトを選択できる.

Factory Method#2

- ペアにして使う必要があるサブクラスのオブジェクトを生成する場合
  - ペアの片方はコンストラクタを呼んでオブジェクトを生成する．生成済みオブジェクトで，もう片方のオブジェクトを生成する.

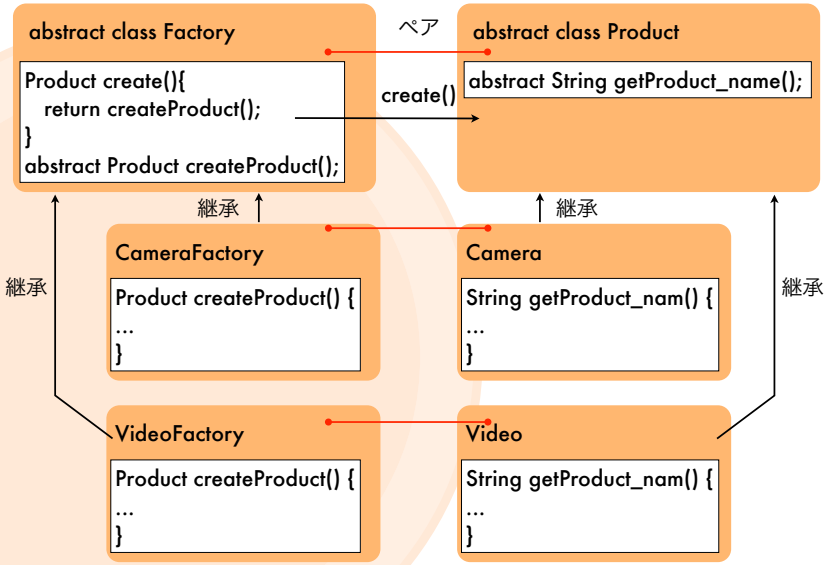


FactoryMethod/rei1/FactoryMethod1.java

Factory Method#3

- Factory Methodは，スーパクラスでオブジェクトの生成の枠組みは決めるが，具体的なクラス名までは決めない．具体的な肉付けはサブクラスで行う．
- ちなみに，Template Methodでは，スーパクラスで処理の流れを記述し，サブクラスで，メッソドの具体的な処理を実装する．

例 FactoryMethod/rei2/Main.java



State

State#1

- 目的
  - オブジェクトの処理内容を状況（状態）に応じて切り替えたい．
- 効果
  - 状況（状態）が複雑に変化しても，処理内容の切り替えがシンプルに行え，コード内での状態遷移の見通しがよくなる．

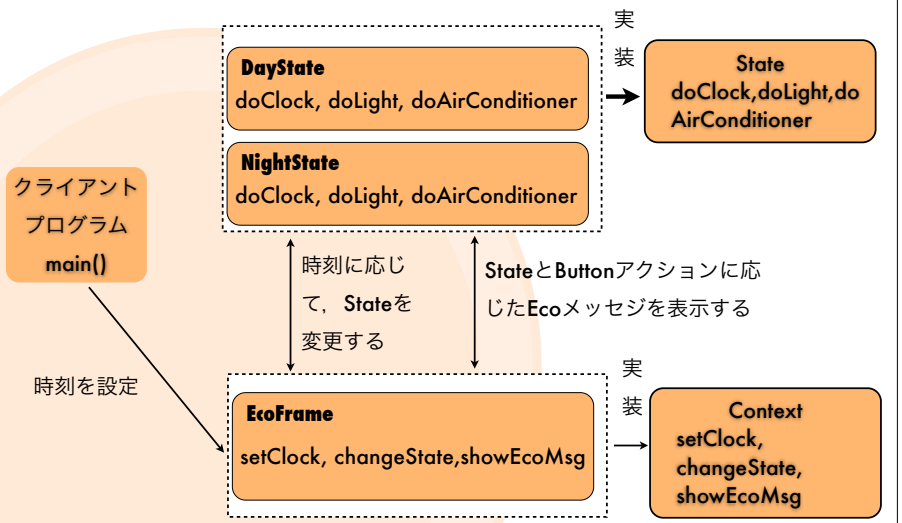
State#2

- 「状況（状態）」をクラスとして表現する．種類ごとに対応した「Stateオブジェクト」を用意する．
- 各ステートオブジェクトに対応する状況の元での処理内容も，ステートオブジェクト自身のメソッドとして実装する方法．

例：

- 時刻ごとで，部屋の照明や空調ONしたときに，エコ的なメッセージを表示するプログラム
  - 「昼間」，「夜間」という状態（Stateクラス）
  - Stateに応じて，照明ON，空調ONに対するメッセージがかわる．

例



Singleton

Singleton#1

- 目的
  - あるクラスのオブジェクトを１個だけ作って共有したい.
- 効果
  - システムの起動から終了までの間に，そのクラスのオブジェクトが１個しか存在しないことを保証する.

Singleton#2

- プログラム全体で１つの情報を共有したい場合，Singletonパターンで，その情報のクラスを定義する.
  - ゲームでの時計
  - ファイルへのアクセス処理で，頻繁にファイルにアクセスする場合，キャッシュにファイルの内容を保持する．そのようなときに，キャ種はシステムで１つあれば十分.

Singleton#3

- Singletonを作る典型的な方法は,
  - Singletonにしたいオブジェクトをクラス変数（static変数）に格納することで，グルーロバルな値にする.
  - コンストラクタをprivateにすることで，不用意に直接コンストラクタを呼び出してオブジェクトを生成することを防ぐ.
  - 唯一のオブジェクトを生成するためのクラスメソッド（getInstance()）を用意する.

Singleton#4

- 例えば，getInstance()としては,

```
if (もうオブジェクトを生成済みであれば) {
 生成済みのオブジェクトを返す
}else {
 新規のオブジェクト生成し,それを返す.
}
```

Singleton/Main.java

まとめ

- デザインパターンの一部について説明をした.
- 残りの大部分のパターンは，説明したパターンを応用，拡張したものである.