

オブジェクト指向の定石：パターン

オブジェクト指向設計：

分析(概念)モデル から 設計(仕様)モデル
への変換

- うまく設計モデルへ変換するには経験が必要
- 設計(デザイン)の典型的パターンを体系化して
カタログ化
 - ⇒ アーキテクチャパターン, デザインパターン
- 適切なクラス構造の設計のためのパターン.
- 「アルゴリズム」(問題の解法)とは違う.
- 新しく考え出したのではなく、既存パターンを体系化



パターンの効能

- 品質のよいソフトウェアが効率的に設計できるようになる
 - 経験者の「経験」を利用する
 - 保守性が向上.
 - 変更容易性: 機能追加が容易になる.
 - 理解容易性: 「○○パターンを利用」ということがわかれば、プログラムが理解しやすくなる.

- ただし、多少コードが長くなったり、実行時の効率が悪くなったりする.
- 1回しか実行しないような簡単なプログラムの開発時には利用する必要性はない.



アーキテクチャパターン

- システム設計における基礎的なアーキテクチャのパターン
- 大規模なパターン
- 一般には、言語には依存しない。
- 抽象度が高い
- MVCモデルが代表的
 - 他に, layers, pipes and filters, blackboard, broker, microkernel, reflection, presentation-abstraction-control(PAC) などがある。
 - 「ソフトウェアアーキテクチャ」という本で提唱された



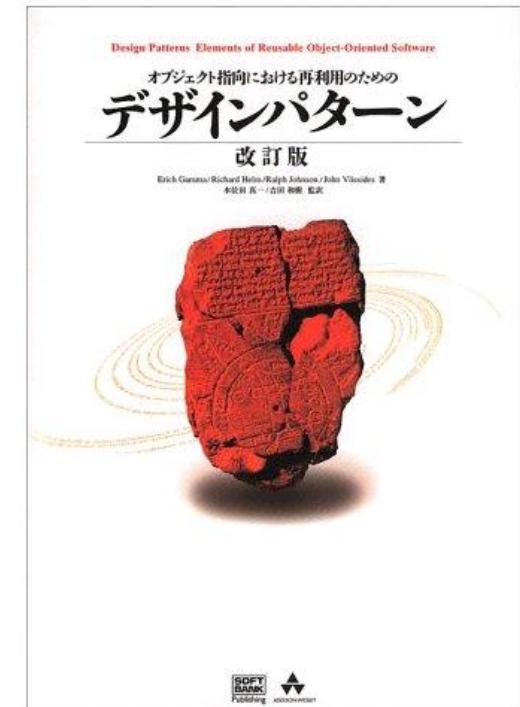
アーキテクチャパターン(8種)

- 混沌から構造へ (From mud to structure)
 - Layers, Pipes and Filters, Blackboard
- 分散システム (Distributed Systems)
 - Broker
- 対話型システム (Interactive Systems)
 - Model-View-Controller (MVC)
 - Presentation-Abstraction-Control (PAC)
- 適合型システム
 - Reflection, Microkernel



GoFのデザインパターン

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995. (邦訳: 本位田真一, 吉田和樹監訳: 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, ソフトバンク, 1999)
 - GoF(Gang of Four) と呼ばれる4人が提唱.
 - 23種類のパターンが収録.
 - 多くのパターンが経験のあるプログラマなら無意識の利用している「当たり前」のパターン.
 - オブジェクト指向設計における経験を容易に共有するために、「デザインパターン」を提唱.
 - 「デザインパターン」は、一般名称ではない！
一般に、この本の23種類のパターンを指す.
- その他のパターン
 - 「アーキテクチャパターン」: MVCモデルを含む
 - 「アナリシスパターン」
 - 「データモデルパターン」
など、他にもあるが、いちばん有名なのは GoFによる「デザインパターン」.



デザインパターンの分類(全23種)

■ 構造に関するパターン(7種類)

- Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

■ 生成に関するパターン(5種類)

- Abstract Factory, Builder, Factory Model, Prototype, Singleton

■ 振る舞いに関するパターン(11種類)

- Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor



生成に関するパターン

■ Factory Method パターン <インスタンス生成をサブクラスにまかせる>

- オブジェクトを生成する時のインターフェースだけを規定して、インスタンス化を子クラスに任せるようにする。

■ Abstract Factory パターン <関連するオブジェクトを組合せて生成>

- 互いに関連しあうオブジェクト群を、そのクラスを明確にせずに生成できるようにする。

■ Builder パターン <複雑なインスタンスを組み立てる>

- オブジェクトの作成過程を表現形式に依存しないようにして、同じ作成過程で異なる表現形式のオブジェクトを生成できるようにする。

■ Prototype パターン <コピーしてインスタンスをつくる>

- 生成すべきオブジェクトの種類の原形となるインスタンスを明確にし、これをコピーすることによって新たなオブジェクトを生成する。

■ Singleton パターン <インスタンスが唯一であることを保証する>

- あるクラスに対してインスタンスが一つしかないことを保証し、それにアクセスするための方法を提供する。

構造に関するパターン(1)

■ Adapter パターン <皮をかぶせて再利用>

- あるクラスにインターフェースを他のインターフェースへ変換し、インターフェースに互換性のないクラス同士を組み合わせることができるようにする。
- Bridge パターン <機能と実装の分離>
 - クラス構造と実装とを分離して、それぞれを独立に変更できるようにする。

■ Composite パターン <容器と中身を統一的に扱う>

- オブジェクトを木構造に組み立てる。
- Decorator パターン <飾り枠を付加>
 - オブジェクトに機能を動的に追加する。



構造に関するパターン(2)

■ Facade パターン <シンプルな窓口>

- 子クラス内に存在する複数のインターフェースに対し、一つの統一的なインターフェースを与える。
- Flyweight パターン <同じものを共有>
 - 多数の細かいオブジェクトを効率よく扱うための共有機構を提供する。

■ Proxy パターン <メソッドのアクセスの制御>

- あるオブジェクトへのアクセスを制御するために、そのオブジェクトの代理を提供する。



振る舞いに関するパターン(1)

■ Template Method パターン <処理をサブクラスにまかせる>

- アルゴリズムの外枠を定義しておき、アルゴリズムの構造を変えずに、その中のいくつかのステップについては、子クラスでの定義に任せるようにする。

■ Observer パターン <状態の変化を通知>

- あるオブジェクトが状態を変えた時に、それに依存するすべてのオブジェクトに自動的にそのことが通知されるような、オブジェクト間の一対多関係を定義する。

■ Iterator パターン <1つ1つを数え上げる>

- オブジェクトが内部表現を公開せずに、要素に順にアクセスする方法を提供する。

■ State パターン <状態をクラスとして表現>

- オブジェクトの内部状態が変化した時に、オブジェクトが振舞いを変えるようにする。



振る舞いに関するパターン (2)

- **Chain of Responsibility パターン <責任の連鎖>**
 - 複数のオブジェクトを鎖状につなぎ、要求をその鎖に沿って次々と渡すようにすることによって、要求送信オブジェクトと受信オブジェクトの直接の結合を避ける。
- **Command パターン <命令のクラス化>**
 - 要求をオブジェクトとしてカプセル化して、クライアントをパラメータ化する
- **Interpreter パターン <文法規則をクラスで表現>**
 - 言語の文法表現と、それを利用して文を解釈するインタプリタを定義する
- **Mediator パターン <調整役のオブジェクトを定義>**
 - オブジェクト群の相互作用をカプセル化するオブジェクトを定義する。



振る舞いに関するパターン (3)

■ Memento パターン <状態を保存する>

- オブジェクトのカプセル化を壊さずに、オブジェクトの内部状態を保存し、後でこの状態に戻すことができるようにする。Undoを実現するためのパターン。

■ Strategy パターン <アルゴリズムを簡単に交換>

- アルゴリズムの集合を定義し、各アルゴリズムをカプセル化してそれらを交換可能にする。

■ Visitor パターン <処理をデータ構造から分離>

- 関連する操作を各クラスから取り出して別オブジェクトにまとめることにより、新しい操作の定義を容易にする。



デザインパターンの分類(全23種)

■ 構造に関するパターン(7種類)

- **Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy**

■ 生成に関するパターン(5種類)

- **Abstract Factory, Builder, Factory Model, Prototype, Singleton**

■ 振る舞いに関するパターン(11種類)

- **Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor**



構造に関するパターン(1)

■ Adapter パターン <皮をかぶせて再利用>

- あるクラスにインターフェースを他のインターフェースへ変換し、インターフェースに互換性のないクラス同士を組み合わせることができるようにする。
- Bridge パターン <機能と実装の分離>
 - クラス構造と実装とを分離して、それぞれを独立に変更できるようにする。

■ Composite パターン <容器と中身を統一的に扱う>

- オブジェクトを木構造に組み立てる。
- Decorator パターン <飾り枠を付加>
 - オブジェクトに機能を動的に追加する。



構造に関するパターン(2)

■ Facade パターン <シンプルな窓口>

- 子クラス内に存在する複数のインターフェースに対し、一つの統一的なインターフェースを与える。
- Flyweight パターン <同じものを共有>
 - 多数の細かいオブジェクトを効率よく扱うための共有機構を提供する。

■ Proxy パターン <メソッドのアクセスの制御>

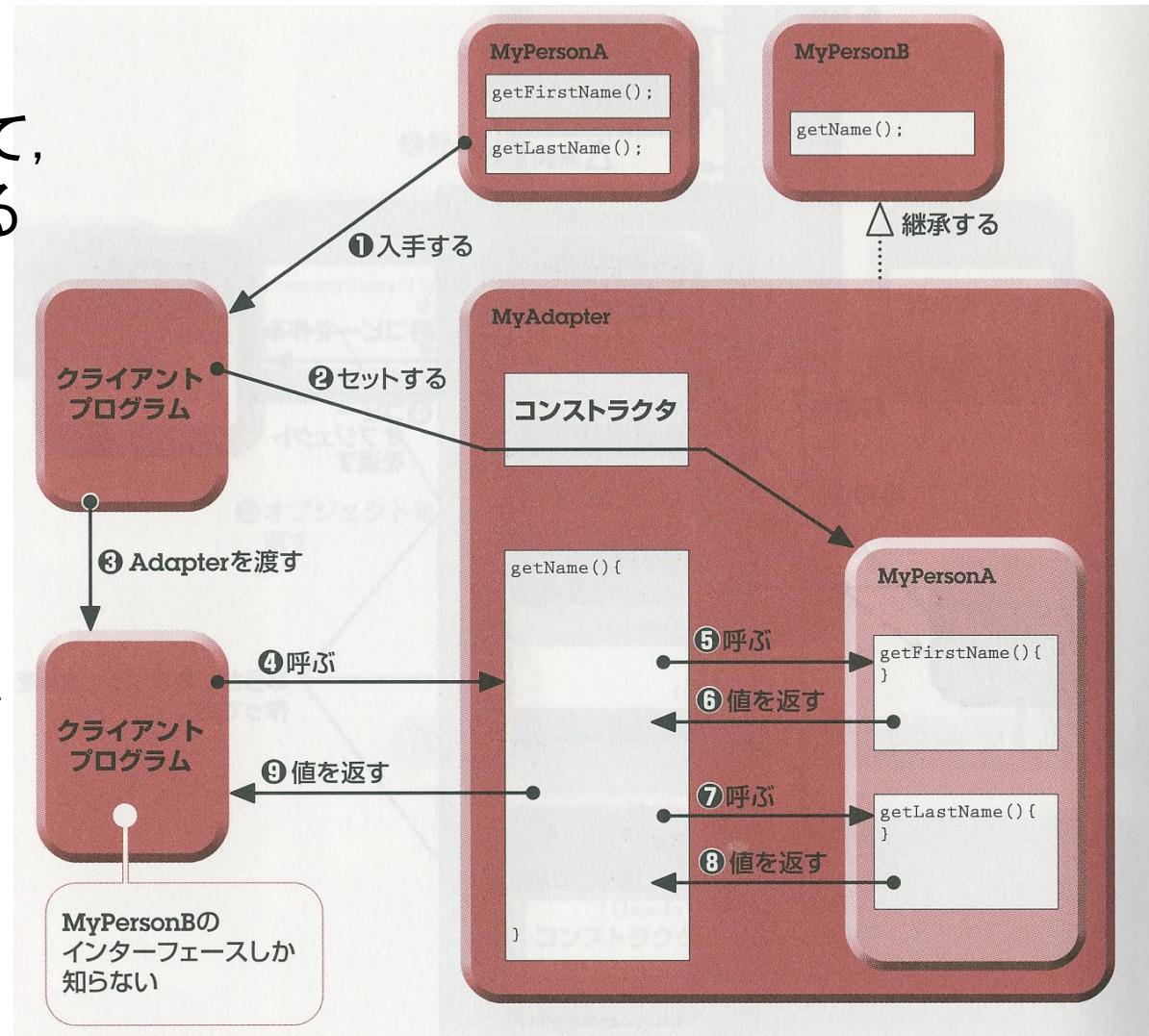
- あるオブジェクトへのアクセスを制御するために、そのオブジェクトの代理を提供する。



Adapter パターン（アダプタ）

＜皮をかぶせて再利用＞

- 既存クラスに別のインターフェースを持たせて、別のクラスとして扱えるようにする。
- オブジェクトコンポジション + 繙承 + ポリモーフィズムを利用して、アダプタークラスを作る。既存クラスのオブジェクトを保持する。
- 既存クラスには、一切手を加えない。



Adapter パターン (2)

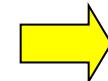
- 既存のクラスに、新しいクラスのインターフェースを持たせる

```
// 既存クラス
class MyPersonA {
    String firstName;
    String lastName;
    MyPersonA(String s1,
              String s2){ firstName=s1;
                         lastName=s2; }
}
```

```
// 新しいクラス
class MyPersonB {
    String Name;
    public getName() { return Name;}
}

// MyPersonBを引数とするメソッド
class PrintName {
    void print(MyPersonB p){
        System.out.println(p.getName());
    }
}
```

PrintNameのメソッドprint() を用いて
既存クラスMyPersonA の中身も表示したい !



Adapter パターンの利用
Adapterクラスを定義する

Adapter パターン (3)

- MyPersonB をextends(インターフェースの場合は implements)したAdapterクラスを用意する

```
// Adapterクラス  
class MyAdapter extends MyPersonB {  
    MyPersonA p;  
    MyAdapter(MyPersonA x) { p=x; }  
    String getName() {  
        return p.firstName+”+p.lastName; }  
}
```

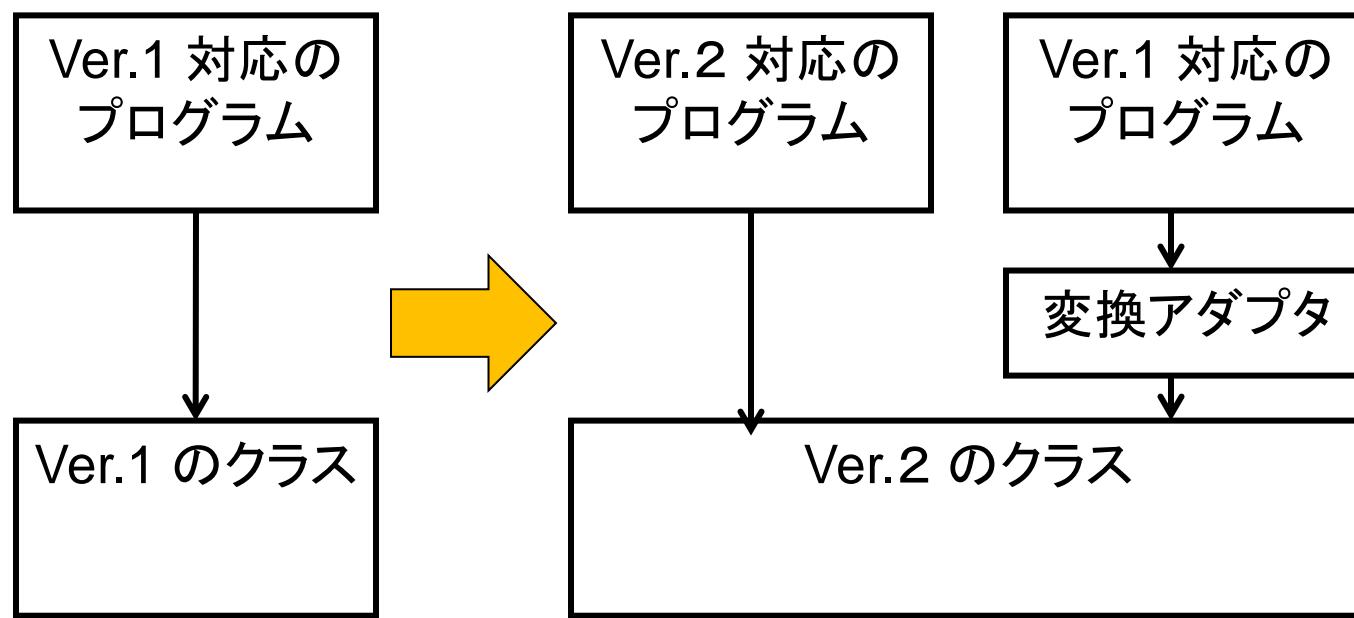
注: MyPersonBがインターフェースなら
オブジェクトコンポジションでなく,
MyPersonAを継承し, さらに
MyPersonBをimplementしてもよい。

以下のように、PrintNameのメソッドprint() を用いて既存クラスMyPersonA
の中身を表示できる。

```
MyPersonA p=new MyPersonA(“Taro”, “Dentsu”);  
MyAdapter a=new MyAdapter(p);  
PrintName pn=new PrintName();  
pn.print(a); //MyPersonB しか対応していないPrintNameが利用できる！
```

Adapter パターン の利用例

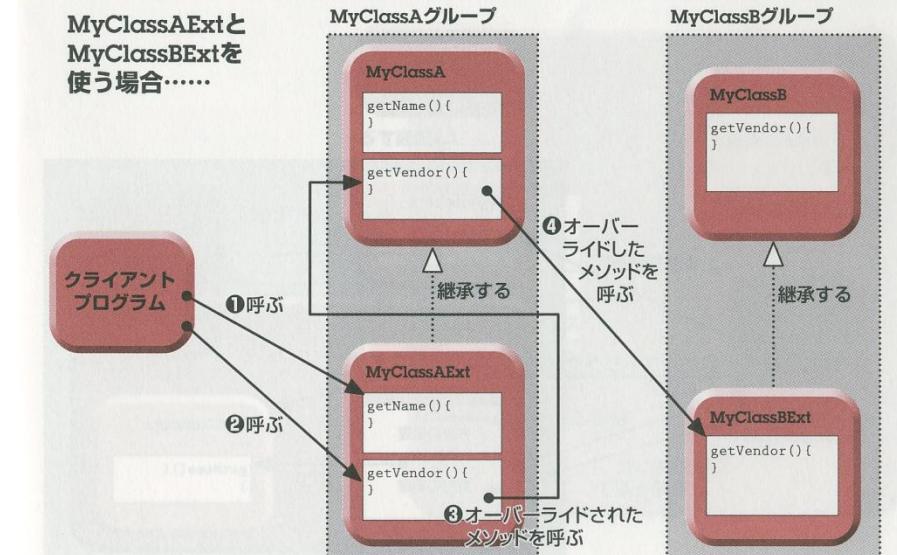
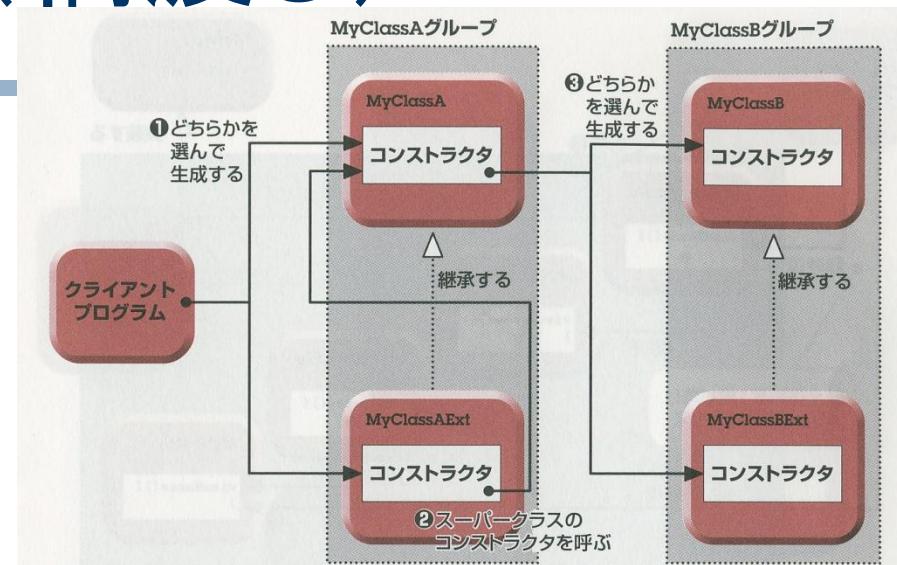
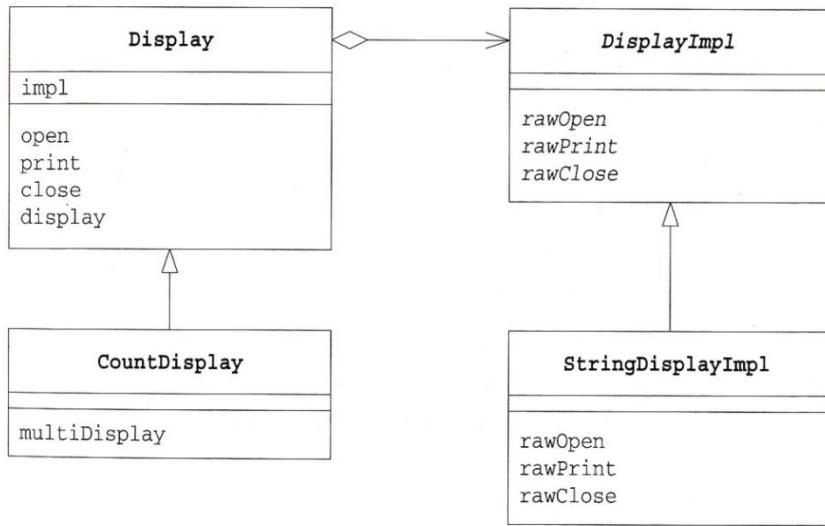
- バージョンアップでインターフェース(メソッド集合)を全面更新した時に、古いバージョンのクラスを利用する



<機能と実装の分離>

Bridgeパターン（橋渡し）

- 機能の階層と 実装の階層を 分離し、独立に変更可能とする。
- 実装の階層は基本機能API を提供。
- 機能の階層はAPI用いて、 まとめた機能を提供。



Bridge パターン (2)

```
// 機能クラス階層
// (Display > CountDisplay)
class Display {
    private DisplayImpl im;
    public Display (DisplayImpl i)
    { im=i; }
    public void open(){im.rawOpen();}
    public void print(){im.rawPrint();}
    public void close(){im.close();}
    public final void disp()
        { open(); print(); close(); }
}
class CountDisplay extends Display{
    public CountDisplay(DisplayImpl i){
        super(i); }
    public void multiDisplay(int n){
        open(); for(int i=0;i<n;i++) print();
        close(); }
}
```

```
// 実装クラス階層 (DisplayImpl > StrDisplImpl)
interface DisplayImpl {
    void rawOpen();
    void rawPrint();
    void rawClose(); }
public class StrDisplImpl
    implements Displayimpl {
    private String str;
    public StrDisplImpl(String s){str=s;}
    public rawOpen() {
        System.out.println("---start---"); }
    public rawPrint() {
        System.out.println(str); }
    public rawClose() {
        System.out.println("---end---"); }
}
```

実装クラスを入れ替えることによって、様々な動作が可能。例えば、異なる環境への対応。

Bridge パターン (3)

■ 実行例:

- 実装クラスを入れ替えることによって、様々な動作が可能。例えば、異なる環境への対応

```
// クラスの利用例
Display dp=new Display(
    new StrDispImpl("test"));
CountDisplay cd=new CountDisplay(
    new StdDispImpl("Count"));

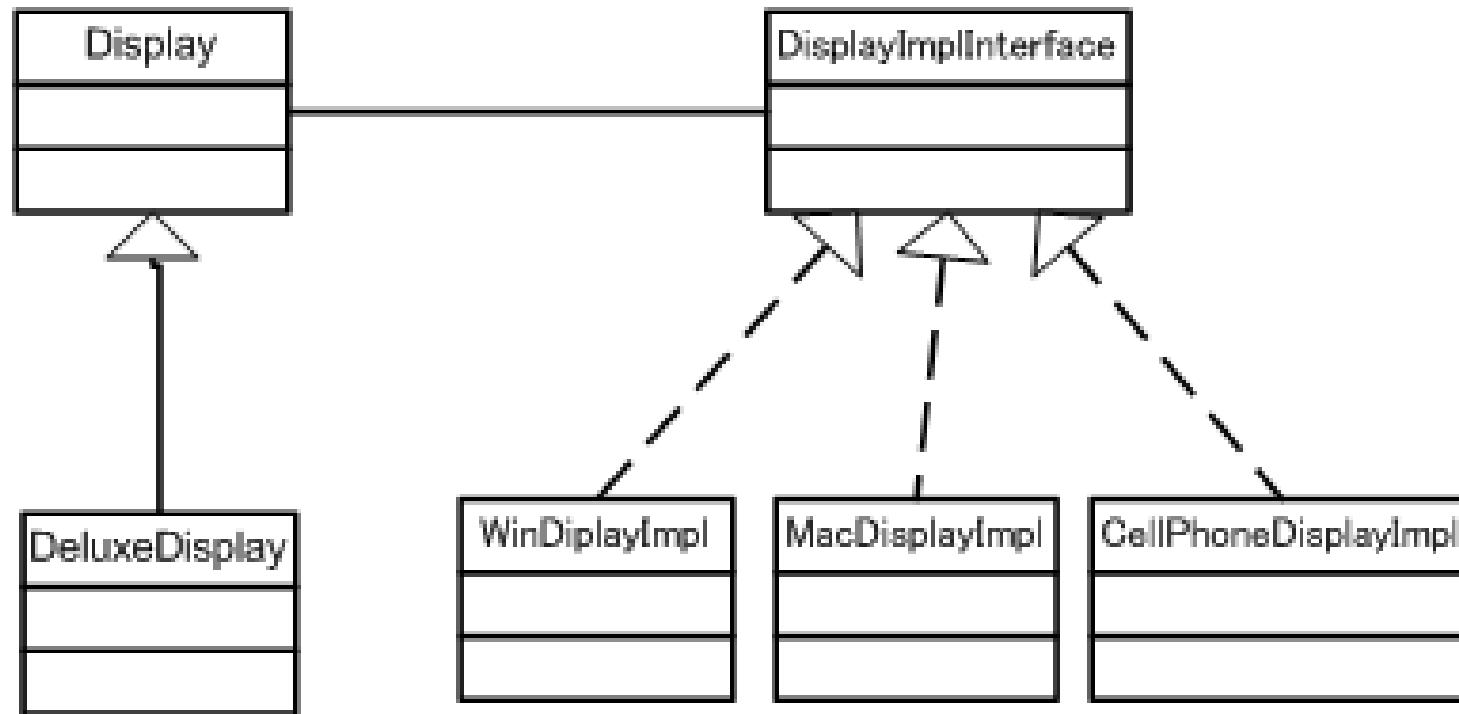
dp.disp();
cd.multiDisplay(3);
```

```
// 実行例
--start--
test
--end--
--start--
Count
Count
Count
--end--
```



Bridgeパターンの利用例

- 環境依存部を実装クラスで実装し、
容易に交換できるようにする。



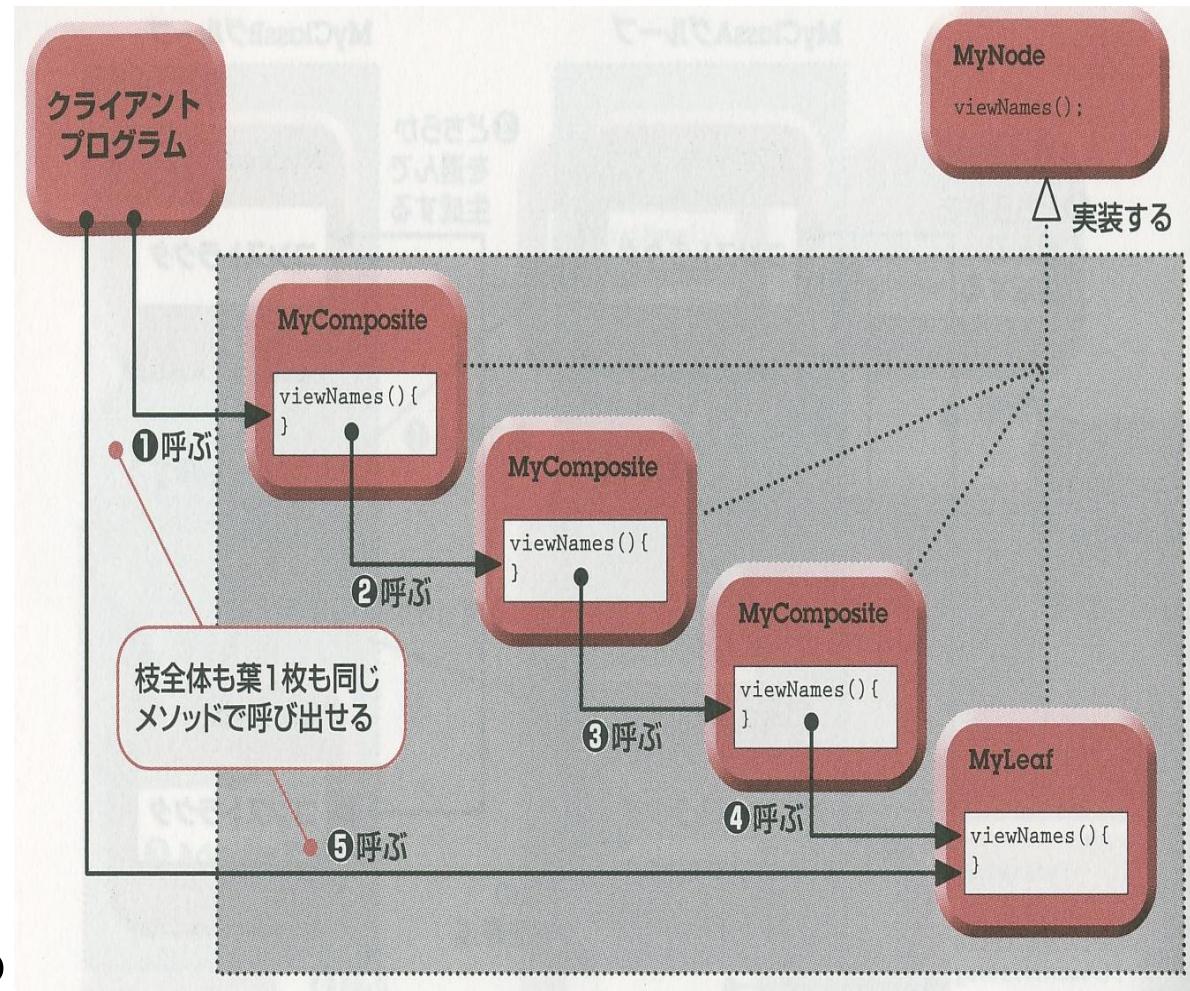
機能部はそのままOK。Windows, Mac, 携帯 に対応。



Composite パターン（合成）

＜木構造を持ったクラス群を生成＞

- 木構造に対して、再帰的に処理を実行する。
左の図は、枝分かれがないのでリスト構造。
- 途中のノード(分岐ノード)と末端のノード(データノード)を同じインターフェースを実装して、定義する
- 例えば、ディレクトリの構造(ディレクトリ(分岐)とファイル(末端))



Composite パターン (2)

```
// 基本のインターフェース
interface MyNode { void viewNames();}

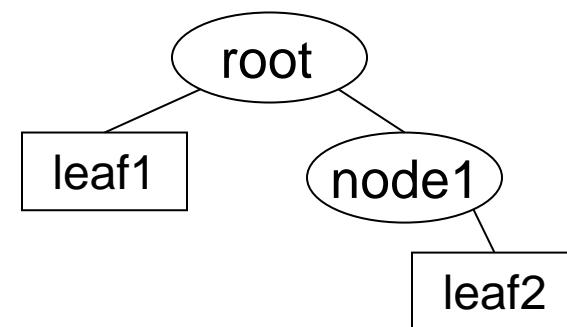
class MyComposite implements
MyNode { // 分岐ノード
private Vector<MyNode> child;
void addNode(MyNode n){
    child.addElement(n); }
public void viewsName(){
    for(int i=0;i<child.size();i++)
        child.elementAt(i).viewNames();
}
}

class MyLeaf implements MyNode {
private String name; // 末端ノード
MyLeaf(String s) { name=s; }
public viewNames(){
    System.out.println(name);
}
}
```

// main 関数の例

```
MyComposite root=new MyComposite();
MyComposite node1=new MyComposite();
root.addNode(new MyLeaf("leaf1"));
root.addNode(node1);
node1.addNode(new MyLeaf("leaf2"));

viewNames(root);
```

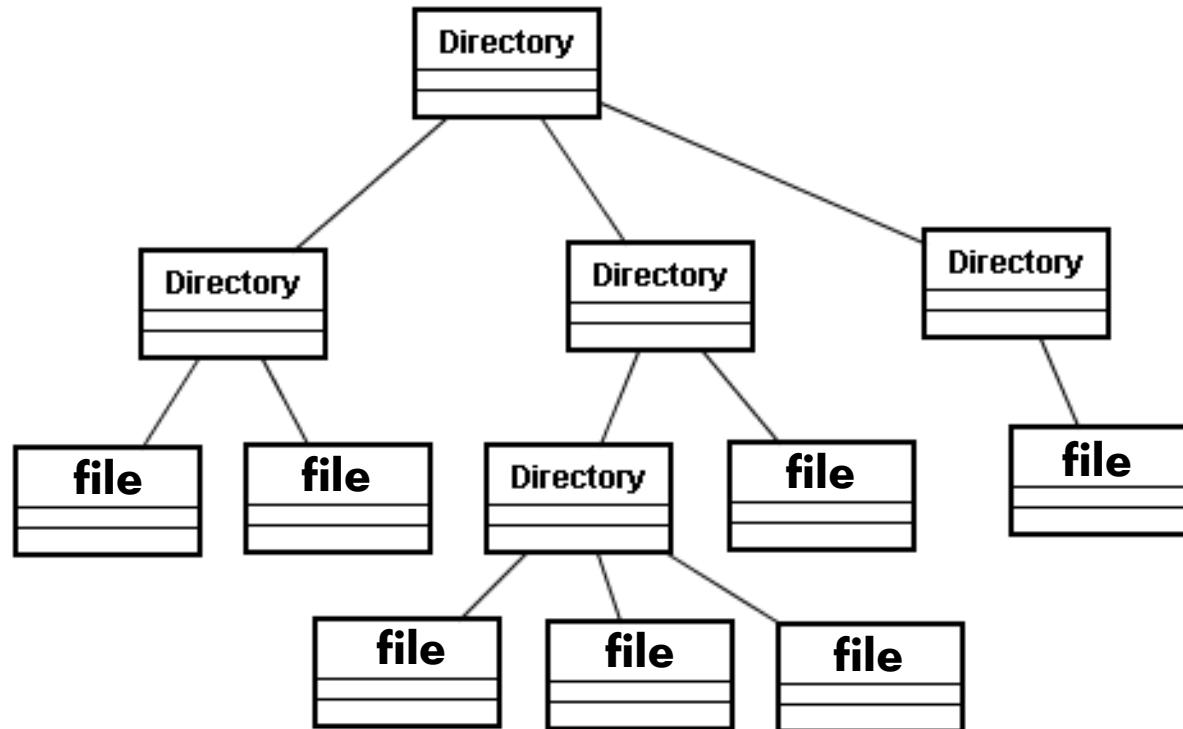


インターフェースの利用によって、複数種類のノードを共存させることが可能。



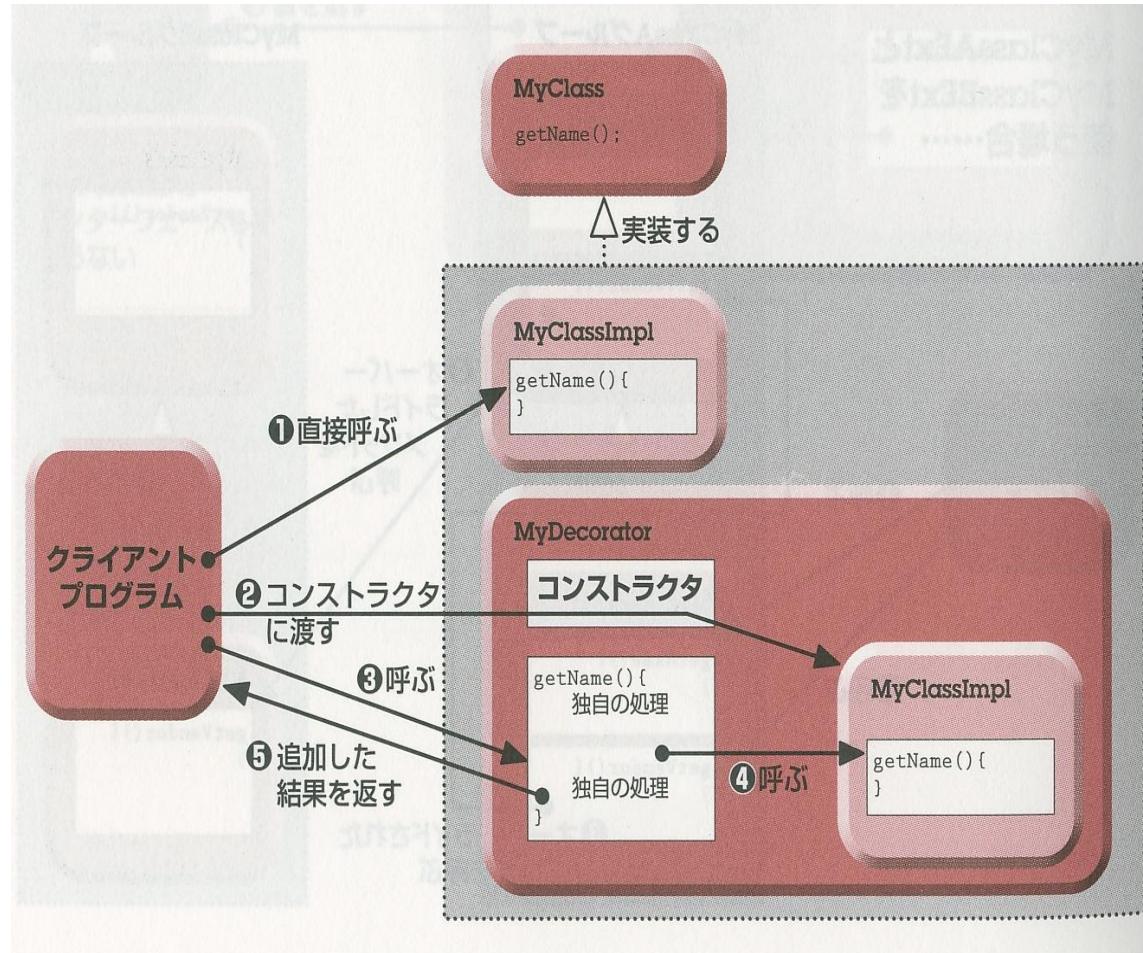
Compositeパターンの利用例

- 木構造のデータの表現
 - ファイルシステムのディレクトリ構造
 - 再帰呼び出しによる全ファイル名の表示



Decorator パターン（飾り付け） ＜飾り枠を付加＞

- 既存のクラスを新しい処理を追加したクラスを作る。
 - 新しいクラスは既存クラスと全く同一に扱うことが可能
 - サブクラスとスーパークラスが同一視できる



Decorator パターン (2)

```
// 基本のクラス（既存クラス）
class MyClass{
    private String name;
    MyClass(String s){ name=s; }
    public String toString() {
        return name;
    }
}

// 拡張クラス
class MyDecorator extends MyClass{
    MyClass target;
    MyDecorator(MyClass t){ target=t; }
    public String toString(){
        return "<" +target+ ">";
    }
}

(MyClass はインターフェースでもよい)
コンストラクタでインスタンスを受け取って、
インスタンス変数に値をコピーして保持する。
```

// main 関数の例

```
MyClass c=new MyClass("software");
MyClass d=new MyDecorator(
    new MyDecorator(new MyDecorator(c)));
```

```
System.out.println(c);
System.out.println(d);
```

予想される実行結果

```
software
<<<software>>>
```

飾り付けを行うフィルターのように
利用できる。



Decorator パターンの利用例

- Javaクラスライブラリの中でも利用されている
 - Java.ioパッケージでの
ファイルストリームからの読み出し時

```
// ファイル読み込み
```

```
Reader reader=new FileReader("data.txt");
```

```
// バッファリング付き
```

```
Reader reader=new BufferedReader(new FileReader("data.txt"));
```

```
// 行番号管理付きバッファリング読み込み
```

```
Reader reader=new LineNumberReader(new BufferedReader(new FileReader("data.txt")));
```

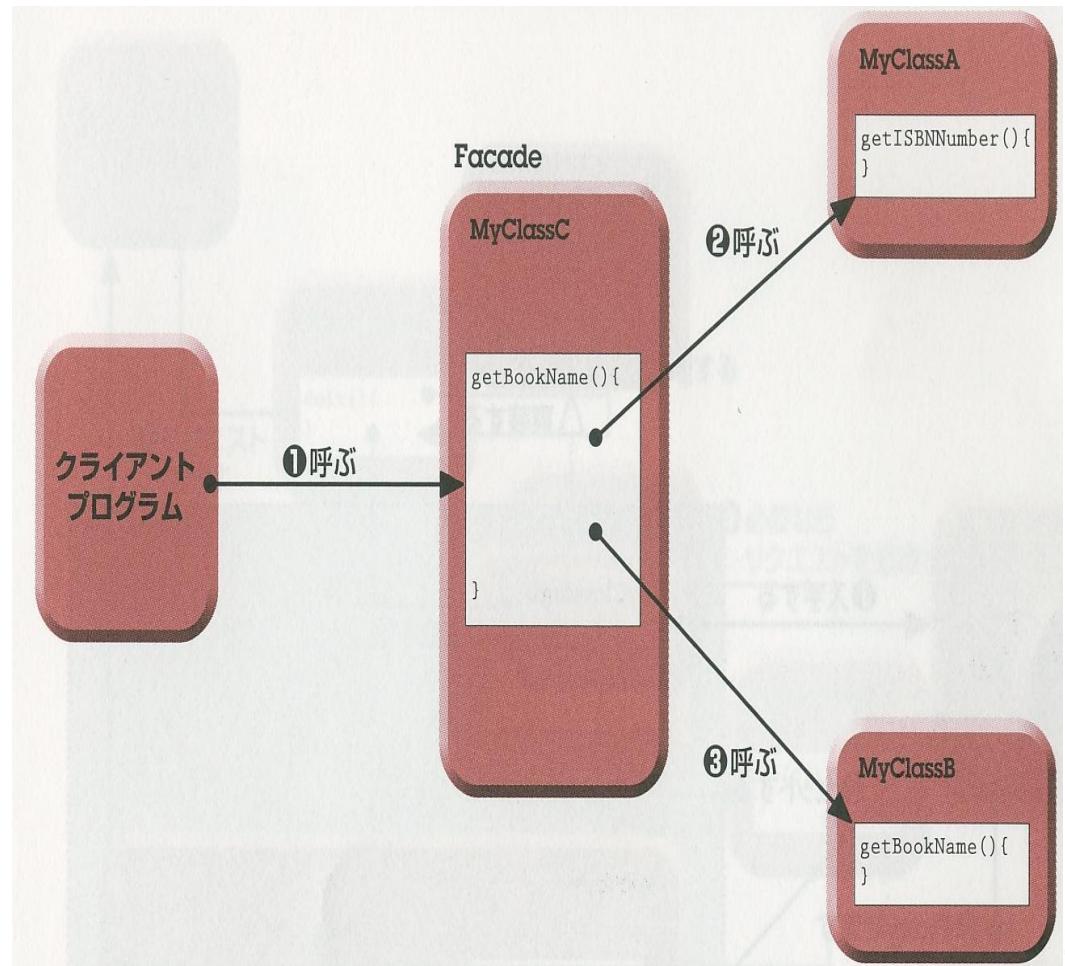
- javax.swing.borderパッケージ: swing部品に飾り
枠を付けるクラス群



Façade(ファサード) パターン

＜シンプルな窓口を提供＞

- 複数のクラスを簡単に利用するためのシンプルな窓口クラスを用意する
- 改めてパターンにするほどでもない、常識的なパターン
- Façade =
フランス語「建物の正面」
➡ 処理の「窓口」



Façade パターン(2)

■ オブジェクトコンポジション(フィールド変数にオブジェクトを保持)の手法のパターン

```
class A {
    void f() { }}
class B {
    void g() { }}
class SUB {
    A a = new A();
    B b = new B();
    void Method(){
        a.f(); b.g();
    }
} // オブジェクトコンポジション版
```

```
class A {
    void f() { }}
class B {
    void g() { }}
class SUB {
    void Method(){
        A a = new A();
        B b = new B();
        a.f(); b.g();
    }
} // ローカル変数版
```

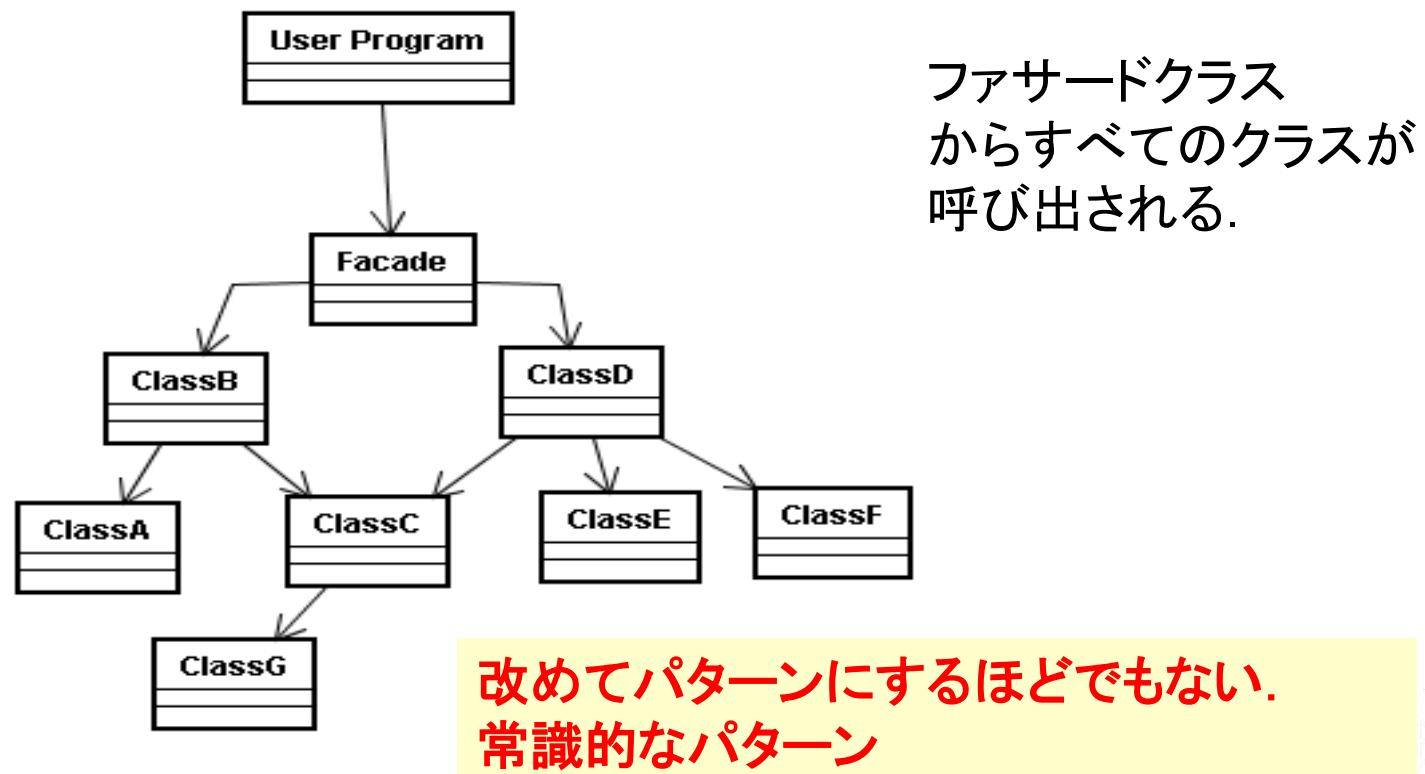
インスタンス変数にオブジェクトを保持して、クラスA, Bのメソッドを利用する。

オブジェクトを保持する必要がない場合は、ローカル変数を利用する場合もある。



Façade パターンの利用例

- 多くのクラスがある時に、呼び出しの窓口を一ヵ所にしてインターフェースをシンプルにする



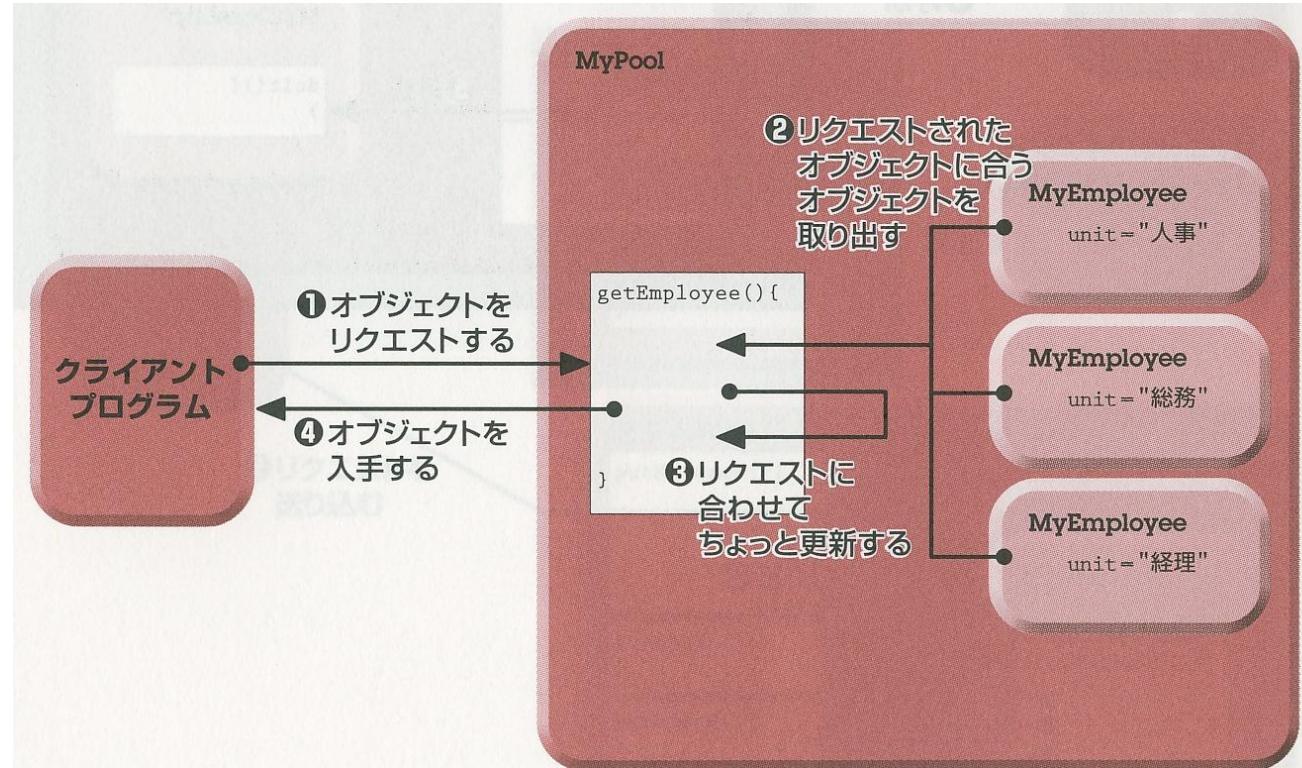
Flyweight パターン

＜同じものを共有し、オブジェクトを軽くする＞

- 同一内容のオブジェクトのインスタンスができるだけ共有させて、無駄を省く

- フライ級 → 動作を軽くする

- オブジェクト生成・保持のコスト削減



Flyweight パターン (2)

```

class MyEmployee{
    public String unit;
    MyEmployee(String u){ unit=u; }
}

class MyPool{
    java.util.HashMap<String,MyEmployee>
        pool= new java.util.HashMap();
    MyEmployee getEmployee(String unit){
        MyEmployee res=pool.get(unit);
        if (res==null){ // なければnewする.
            res=new MyEmployee(unit);
            pool.put(unit, res); // ハッシュ登録
        } // あれば、生成せずにhashから
           // 獲得したオブジェクトを返す.
        return res;
    }
}

```

// main 関数の例

```

Vector<MyEmployee> list;
MyPool pool=new MyPool();
list.add(pool.getEmployee("人事")); // new
list.add(pool.getEmployee("総務")); // new
list.add(pool.getEmployee("総務")); // 自動的に再利用
list.add(pool.getEmployee("経理")); // new
.....

```

Java.util.HashMap クラス(ハッシュ表)を利用して、同一内容のオブジェクトの2個以上の生成を防止する。

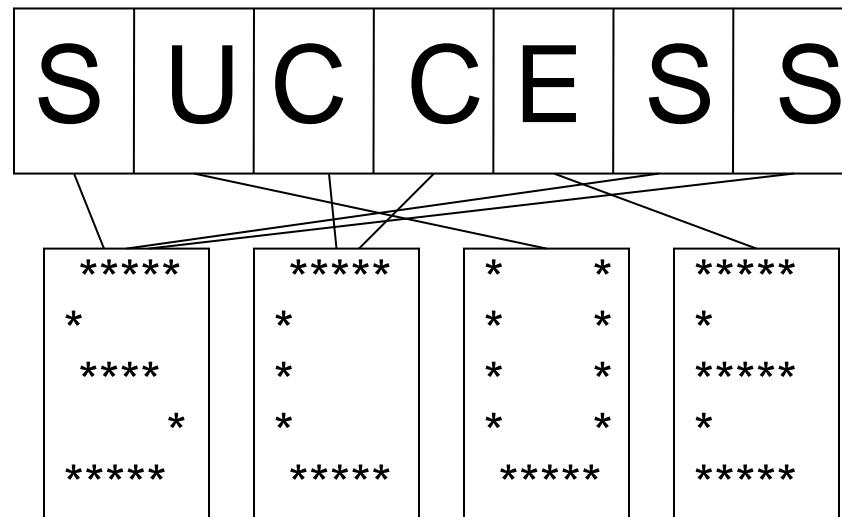
getEmployee メソッドでは、同じunit名のオブジェクトを探して、あれば再利用。なければ、新規に生成する。

中身が変化しないオブジェクトに利用。



Flyweight の利用例

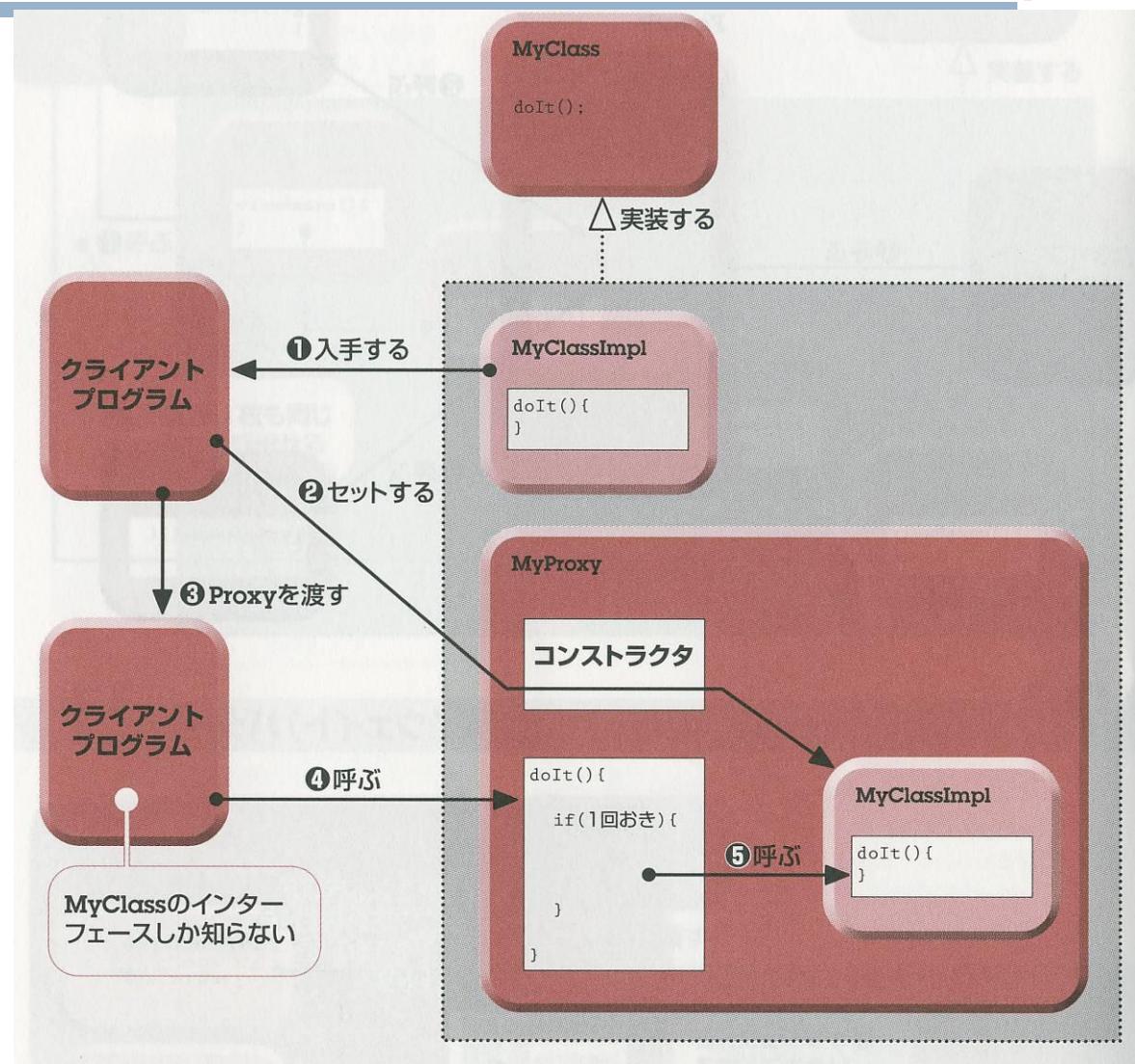
- 生成に手間が掛かって、メモリを消費するようなインスタンスを共有
 - * で文字列を表示する場合、各文字のパターンを1文字ずつインスタンスとして生成して、Vector でそれらを保持する場合.



Proxy パターン（代理人）

＜メソッドのアクセスの制御＞

- オブジェクトが他からアクセスされた時に、実行をフックして、処理を追加する



Proxy パターン (2)

```
// 基本のインターフェース
interface MyClass {
    void doIt();
}
class MyClassImpl implements MyClass{
    public void doIt() {
        System.out.println("Do it!!");
    }
}
// Proxyクラス
class MyProxy implements MyClass{
    MyClass mc;
    boolean flag=true;
    MyProxy(MyClass m){ mc=m; }
    public void doIt(){
        if (flag){ mc.doIt(); }
        else{ // do nothing
        }
        flag= !flag;
    } // 2回に1回しか実行しない.
}
```

// main 関数の例 (中身のみ示す)

```
MyClassImpl imp=new MyClassImpl();
MyProxy proxy=new MyProxy(imp);
for(int i=0;i<10;i++){
    System.out.print(i+” [direct]: “);
    Imp.doIt();
    System.out.print(i+” [proxy]: “);
    Proxt.doIt();
}
```

予想される実行結果

```
0 [direct]: Do it!!
0 [proxy]: Do it!!
1 [direct]: Do it!!
1 [proxy]: 2 [direct]: Do it!!
...
...
```



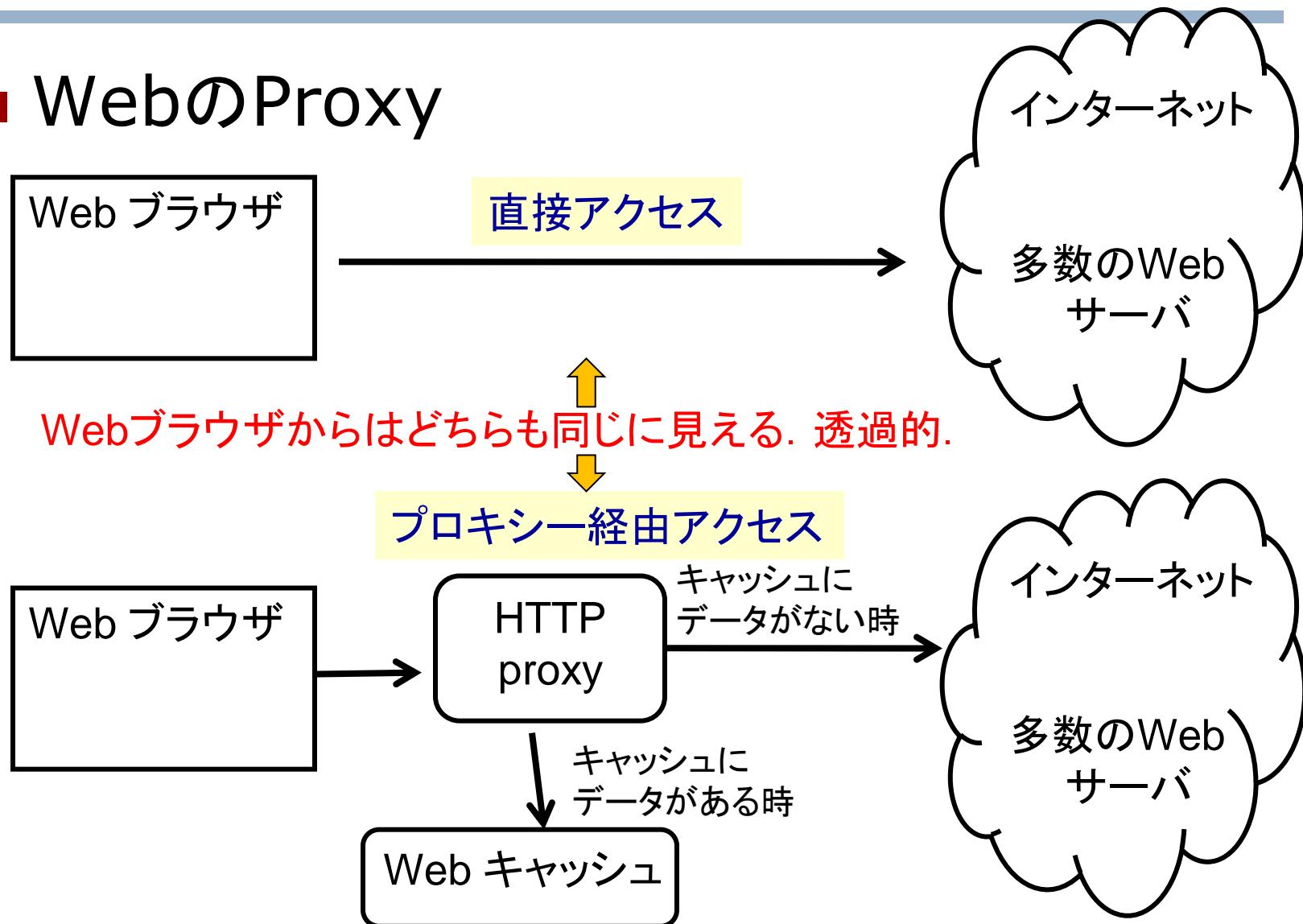
Proxy パターン (3)

- 他にもProxyパターンとして、
 - Virtual Proxy:
インスタンスが本当に必要になった時点で生成。
 - 例えば、プリンターバッファなど。印刷時点で生成。
 - Remote Proxy:
オブジェクトがネットワーク越しに別の場所(計算機)にあるにもかかわらず、メソッド呼び出しができる
(Remote Method Invocation: Java RMIなど)
 - Access Proxy: 今回説明したタイプ



Proxyパターンの使用例

■ WebのProxy



構造に関するパターン(1)

■ Adapter パターン <皮をかぶせて再利用>

- あるクラスにインターフェースを他のインターフェースへ変換し、インターフェースに互換性のないクラス同士を組み合わせることができるようにする。
- Bridge パターン <機能と実装の分離>
 - クラス構造と実装とを分離して、それぞれを独立に変更できるようにする。

■ Composite パターン <容器と中身を統一的に扱う>

- オブジェクトを木構造に組み立てる。
- Decorator パターン <飾り枠を付加>
 - オブジェクトに機能を動的に追加する。



構造に関するパターン(2)

■ Facade パターン <シンプルな窓口>

- 子クラス内に存在する複数のインターフェースに対し、一つの統一的なインターフェースを与える。
- Flyweight パターン <同じものを共有>
 - 多数の細かいオブジェクトを効率よく扱うための共有機構を提供する。

■ Proxy パターン <メソッドのアクセスの制御>

- あるオブジェクトへのアクセスを制御するために、そのオブジェクトの代理を提供する。



Javaクラスライブラリ中で利用されているデザインパターン

- 入出力のストリームは, **Decorator**パターン
- SwingのJComponent, JPanelは, **Composite**パターン
 - (再)描画時に, paintComponent() が再帰的に呼び出される
- Swingが各プラットフォームの部品を生成する方法は, **Abstract Factory**パターン
- java.util.Iterator は, **Iterator**パターン
- java.util.Observer は, **Observer**パターン



デザインパターンの分類(全23種)

■ 構造に関するパターン(7種類) 前回説明

- Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

■ 生成に関するパターン(5種類) 本日説明

- **Factory Method, Abstract Factory, Builder, Prototype, Singleton**

■ 振る舞いに関するパターン(11種類)

- **Template Method, Observer, Iterator, State,**
Chain of Responsibility, Command, Interpreter,
Mediator, Memento, Strategy , Visitor



オブジェクト生成に関するパターン

■ Factory Method パターン <インスタンス生成をサブクラスにまかせる>

- オブジェクトを生成する時のインターフェースだけを規定して、インスタンス化を子クラスに任せるようにする。

■ Abstract Factory パターン <関連するオブジェクトを組合せて生成>

- 互いに関連しあうオブジェクト群を、そのクラスを明確にせずに生成できるようにする。

■ Builder パターン <複雑なインスタンスを組み立てる>

- オブジェクトの作成過程を表現形式に依存しないようにして、同じ作成過程で異なる表現形式のオブジェクトを生成できるようにする。

■ Prototype パターン <コピーしてインスタンスをつくる>

- 生成すべきオブジェクトの種類の原形となるインスタンスを明確にし、これをコピーすることによって新たなオブジェクトを生成する。

■ Singleton パターン <インスタンスが唯一であることを保証する>

- あるクラスに対してインスタンスが一つしかないことを保証し、それにアクセスするための方法を提供する。

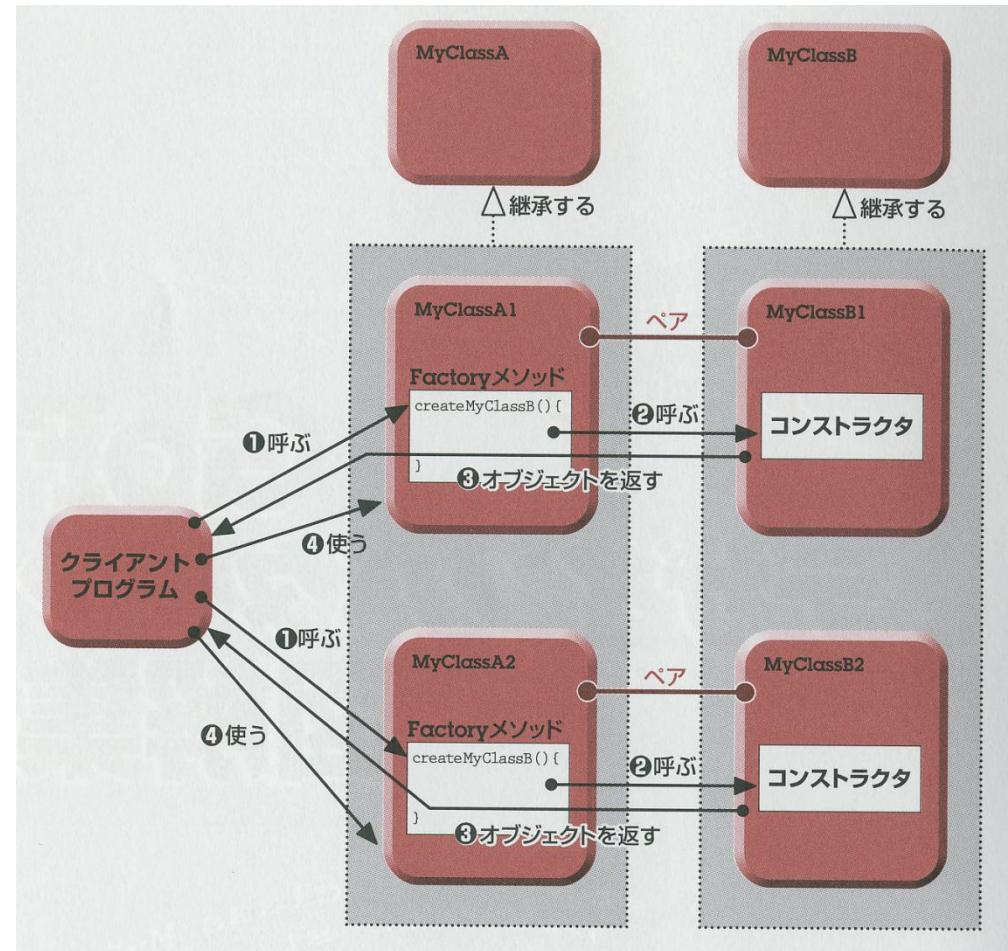
Factory Method パターン

＜インスタンス生成をサブクラスにまかせる＞

■ オブジェクト生成をサブクラスにまかせる

オブジェクトを生成する
メソッド
を「ファクトリ(factory)
メソッド」という。

複数のサブクラスがあって、
それぞれに対応するクラスが
あるとする。



Factory Method パターン (2)

```
// 抽象クラス (interfaceでも可)
abstract class Factory {
    abstract String getName();
    abstract int getPrice();
    abstract Product create();
}

class TvFactory extends Factory{
    String getName(){ return "TV";}
    int getPrice(){ return 100000; }
    Product create() {
        return new TV(); }
}

class PcFactory extends Factory{
    String getName(){ return "PC";}
    int getPrice(){ return 50000; }
    Product create() {
        return new PC(); }
}
```

```
interface Product{
    void PlayGame();
}

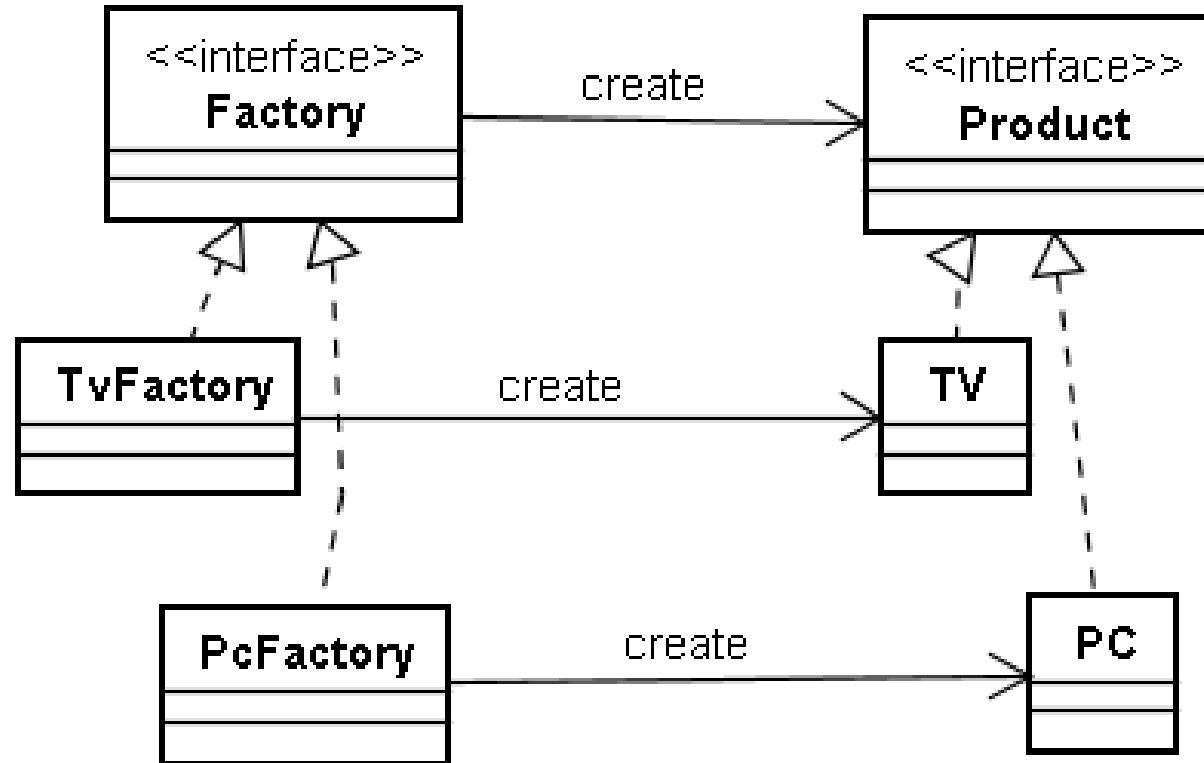
class TV implements Product{
    void PlayGame() { .... }
}

class PC implements Product{
    void PlayGame() { .... }
}

// main関数の中身
Factory f1=new TvFactory();
Factory f2=new PcFactory();
Product a=f1.create(), b=f2.create();
a.PlayGame();

// 実際に生成されるオブジェクトは
// f1 に入っているオブジェクトに依存
// Factory の変更のみで全体が変化
// OOFactory... と容易に拡張可能
```

Factory Method パターン のクラス図



対応するFactoryクラス(TV->TvFactory,PC->PcFactory)で、
オブジェクトを生成する。Factoryクラスは、オブジェクト生成用のクラス。

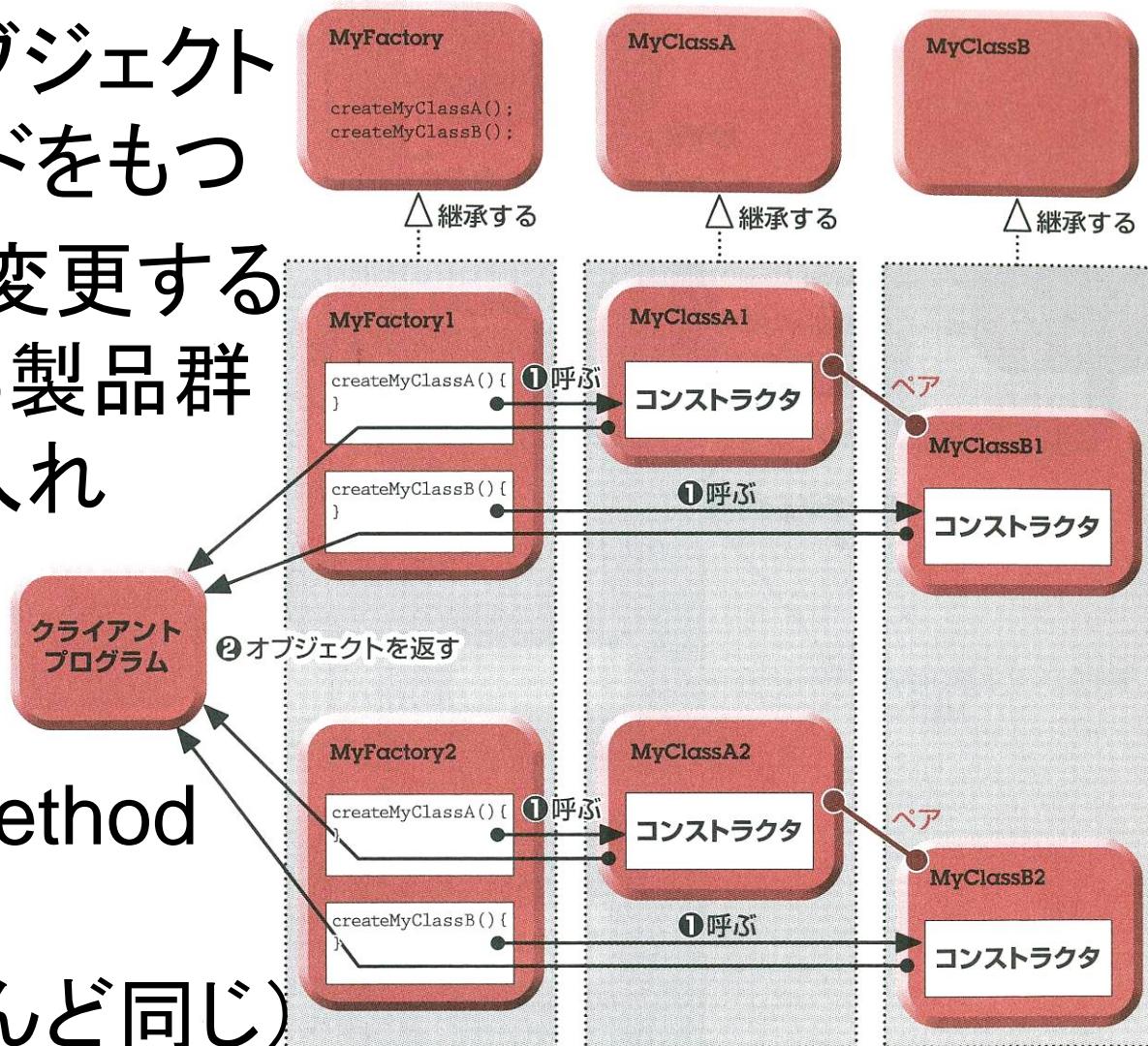
Abstract Factory パターン

〈関連するオブジェクトを組合せて生成〉

- 複数のオブジェクト生成メソッドをもつ

- Factoryを変更する
生産される製品群
がすべて入れ
替わる

- Factory Method
の発展形
(実はほとんど同じ)



Abstract Factory パターン (2)

```

abstract class Factory {
    private String str;
    abstract MyClassA createA();
    abstract MyClassB createB();
}

class Factory1 extends Factory{
    MyClassA createA() {
        return new MyClassA1(); }
    MyClassB createB() {
        return new MyClassB1(); }
}

class Factory2 extends Factory{
    MyClassA createA() {
        return new MyClassA2(); }
    MyClassB createB() {
        return new MyClassB2(); }
}

```

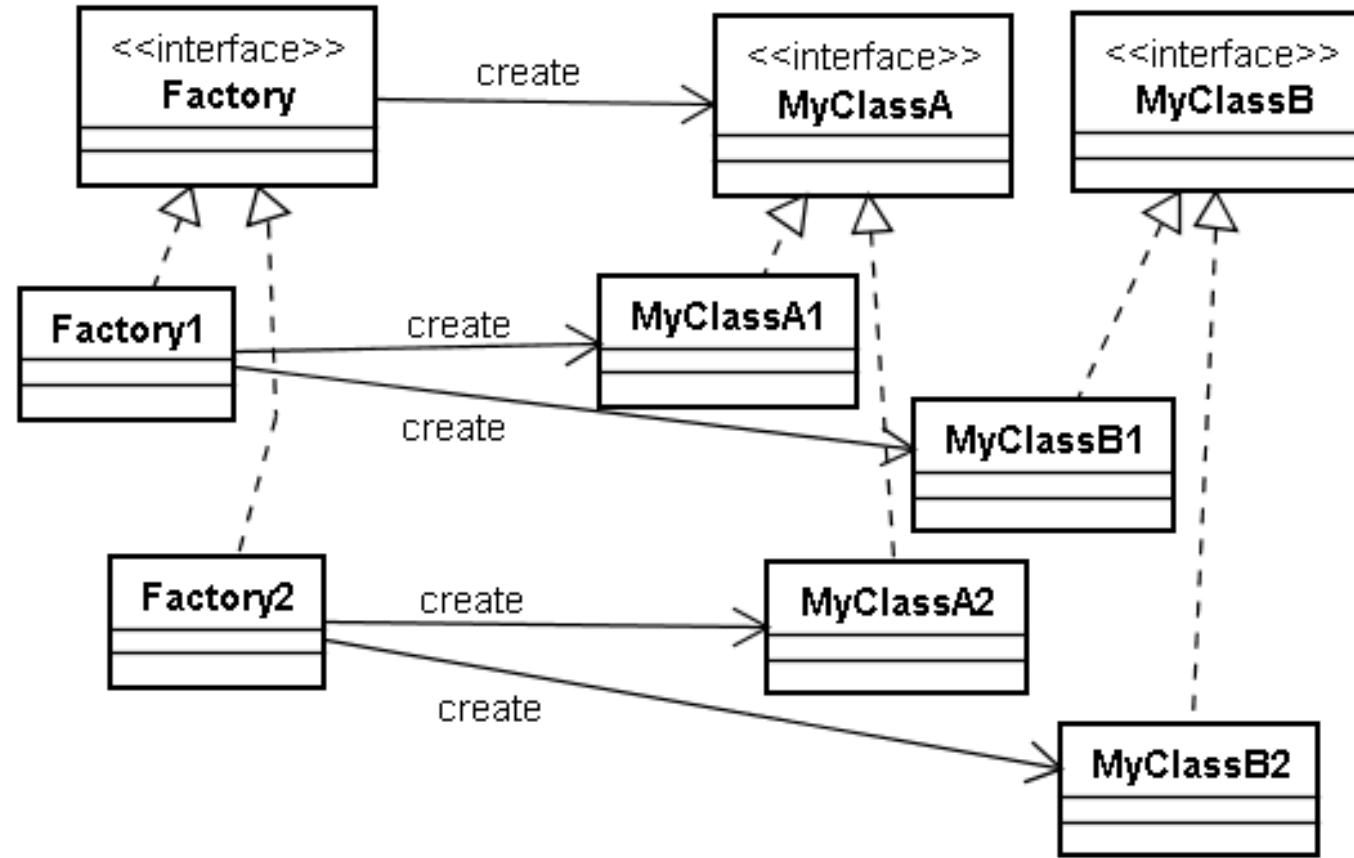
```

interface MyClassA{ String getAddress(); }
class MyClassA1 implements MyClassA{
    String getAddress() { return "Tokyo"; } }
class MyClassA2 implements MyClassA{
    String getAddress() { return "Osaka"; } }
interface MyClassB{ String getAddress(); }
class MyClassB1 implements MyClassB{
    String getAddress() { return "New York"; } }
class MyClassB2 implements MyClassB{
    String getAddress() { return "Chicago"; } }

// main関数の中身
Factory f=new Factory2();
MyClassA a=f.createA();
MyClassB b=f.createB();
System.out.println(a.getAddress());
// 実際に生成されるオブジェクトは
fに入っているオブジェクトに依存
fの中身を変えると、全体の動作も変わる

```

Abstract Factory のクラス図



Factory Method とは、单一クラスのオブジェクトを生成していたが、
Abstract Factory では、関連するオブジェクトをまとめて生成する。

Factory Method と Abstract Factory

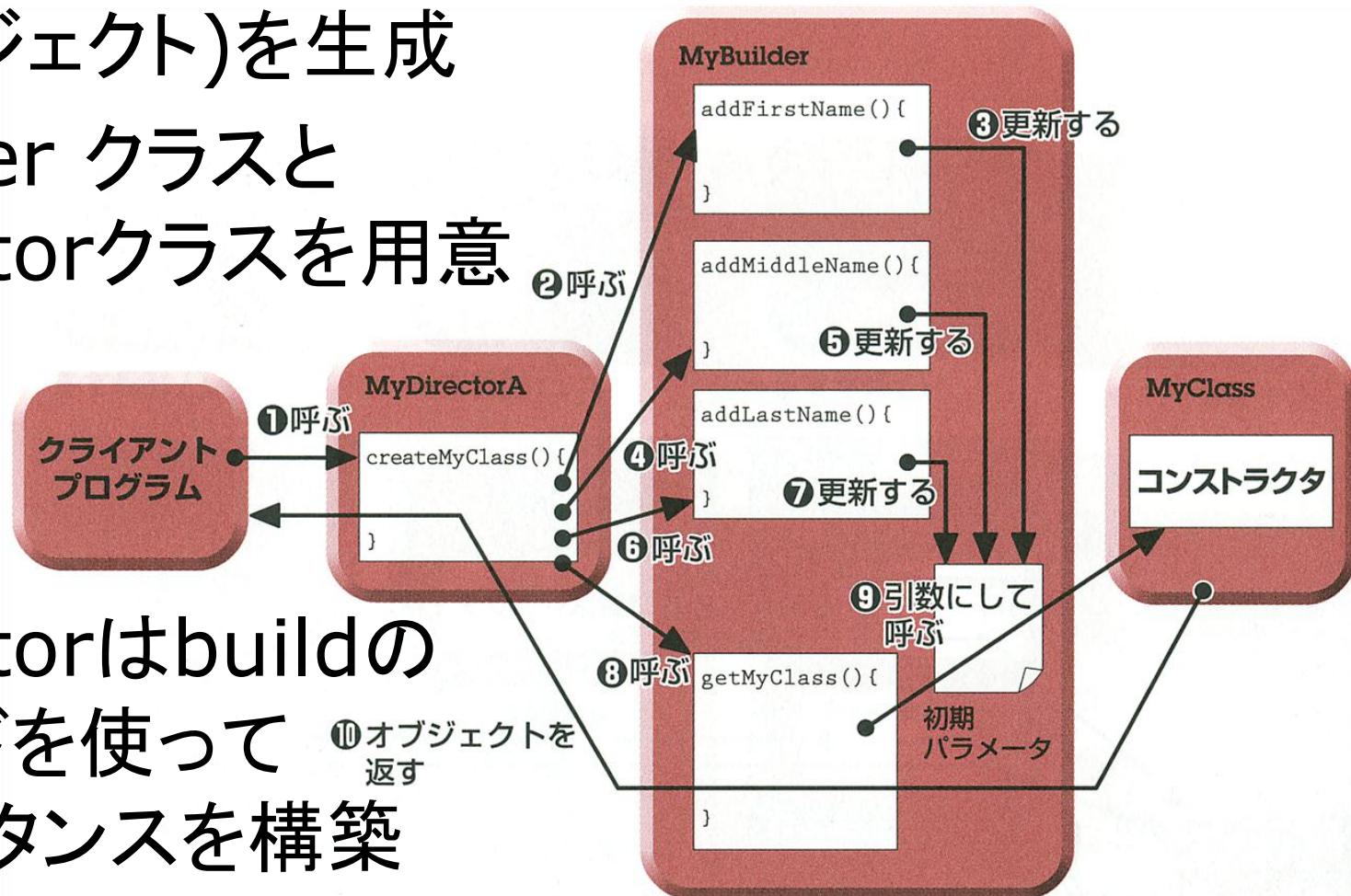
- どちらもオブジェクトを生成するメソッド(Factory)を利用したパターン
- Factoryでオブジェクトを生成することによってポリモーフィズムを利用したオブジェクト生成が可能となる。
 - Factoryメソッドを持ったオブジェクトを変更するだけで、他に生成されるオブジェクトの種類も自動的に変更される。
 - それによって、関連するオブジェクト同士を組み合わせて生成することも可能となる。 (abstract factory)



Builderパターン

〈複雑なインスタンスを組み立てる〉

- 複雑なインスタンス(オブジェクト)を生成
- Builder クラスと Directorクラスを用意



Builder パターン (2)

```
// 抽象クラス
abstract class Builder {
    String name;
    abstract void addFirstName();
    abstract void addLastName();
    String getName() { return name; }
}

// Director にメソッドを提供
class Builder1 extends Builder{
    void addFirstName(String s){
        name = name + s; }
    void addLastName(String s){
        name = name + " " + s; }
}
```

```
class Builder2 extends Builder{
    void addFirstName(String s){
        name = name + "[First name:" +s+"]"; }
    void addLastName(String s){
        name = name + " [Last name:" +s+"]"; }
}

class Director{ // インスタンスの構築
    String create(Builder b){
        b.addFirstName("Taro");
        b.addLastName("Dentsu");
        return b.getName();
    }
}

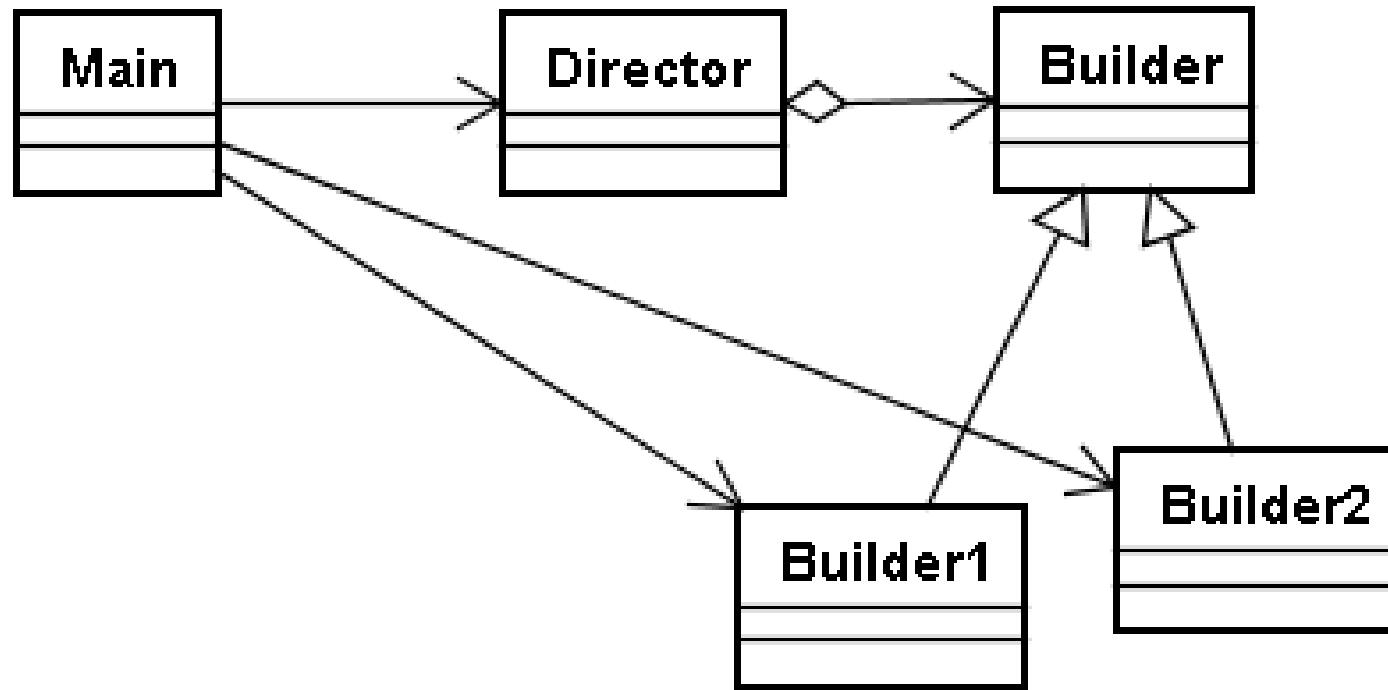
// main関数の中身
Director d=new Director();
String c=d.create(new Builder1());
String c=d.create(new Builder2());
// Builder1 を Builder2 に変えることで
// 生成されるオブジェクト(文字列)が変わる
```

Bulider パターン の まとめ

- 複雑なデータを生成するためのパターン
- 実際に作るBuilderクラス と
Builderクラスのメソッドを組み合わせて呼び出してデータ生成の指示をするDirectorクラスから成る.
- Builderクラスは抽象クラスとして定義され,
実際のbuildは継承したサブクラスが行う.
- 一方, Directorクラスは, 抽象クラスBuilderで宣言されたメソッドを組み合わせて呼び出す. サブクラスのことは知らないので, 容易にサブクラスを交換可能.
- 後述するTemplate Methodと似ているが, Builderパターンでは, DirectorがBuilderをコントロールする. 一方, Template Methodでは, 親の抽象クラスが子クラスをコントロールする.



Builder パターン のクラス図



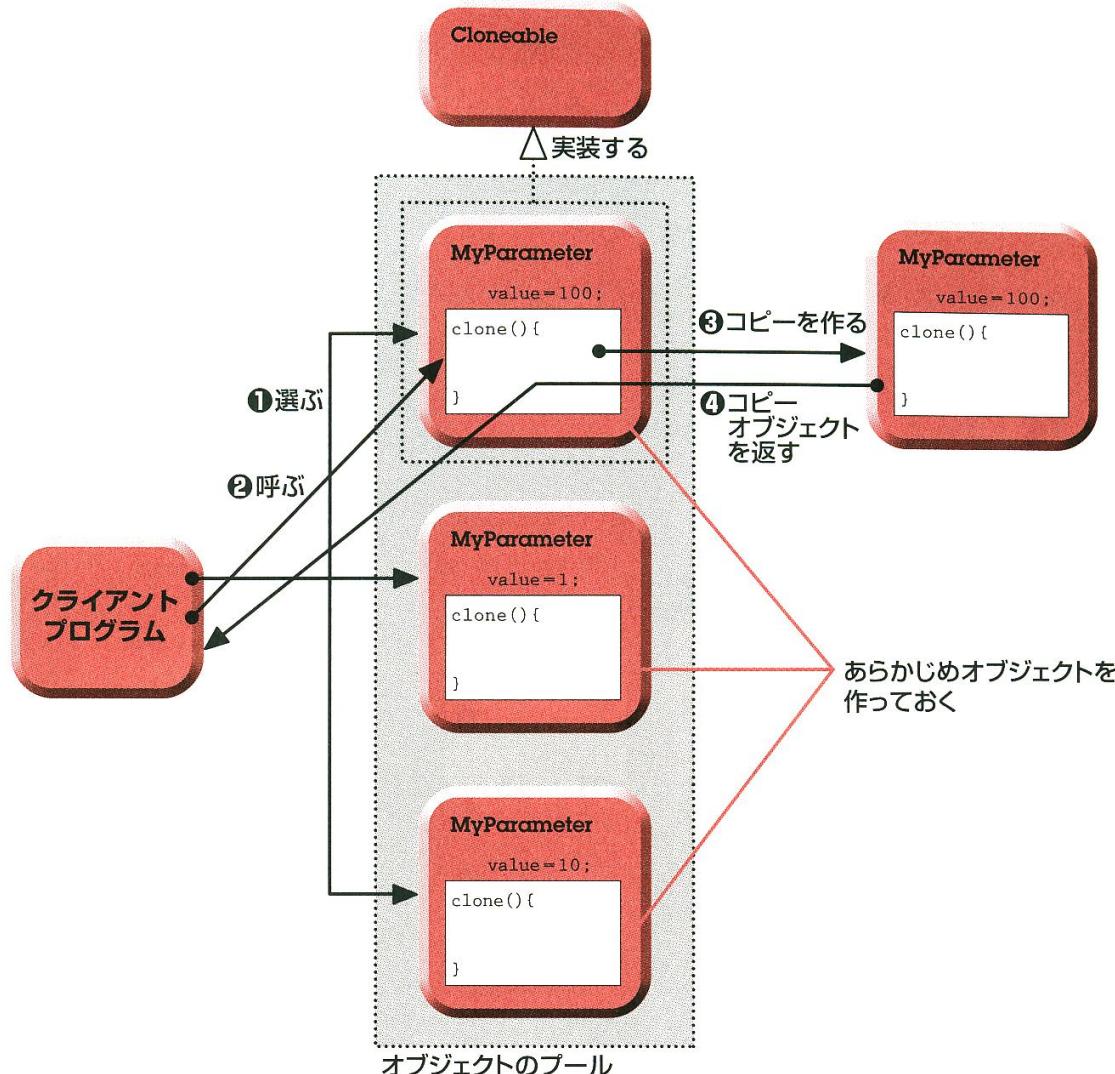
Directorは、Builderを呼び出して、必要なデータを作り上げて、Main(呼び出し元)に渡す。



Prototypeパターン

〈コピーしてインスタンスをつくる〉

- 既存のオブジェクトをコピーして新しいオブジェクトを生成
- Javaの場合は、ライブラリのObject.cloneメソッド, cloneableインターフェースを利用する。



Prototypeパターン(2)

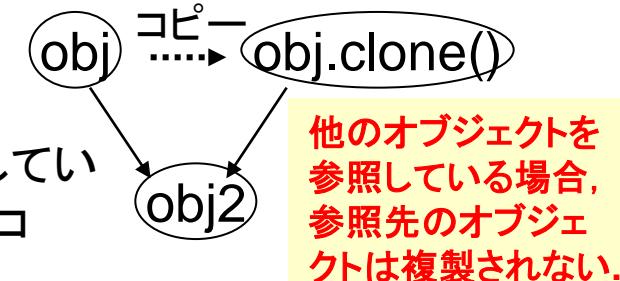
```
// コピーされるクラス
// clone()を利用する場合は、
// java.lang.Cloneable の実装必須。
class MyParam
    implements Cloneable{
private int value;
MyParam(int x){ value=x; }
public Object clone() throws
    CloneNotSupportedException{
    return super.clone();
} // clone()は java.lang.Objectでprotected
} //として定義されているので、利用するには、
//privateとしてオーバライドする必要がある。
```

【注意】Clonableインターフェースはメソッドを持たない。
cloneメソッドは、java.lang.Objectで定義されている。

オブジェクトコンポジションの場合は、cloneメソッドでは、保持しているオブジェクト自体はコピーされない。Clone()の中で明示的にコピーする必要がある。

```
// main関数の中身
// プロトタイプを用意
try{
    MyParam a1 =new MyParam(1);
    MyParam a100=new MyParam(100);

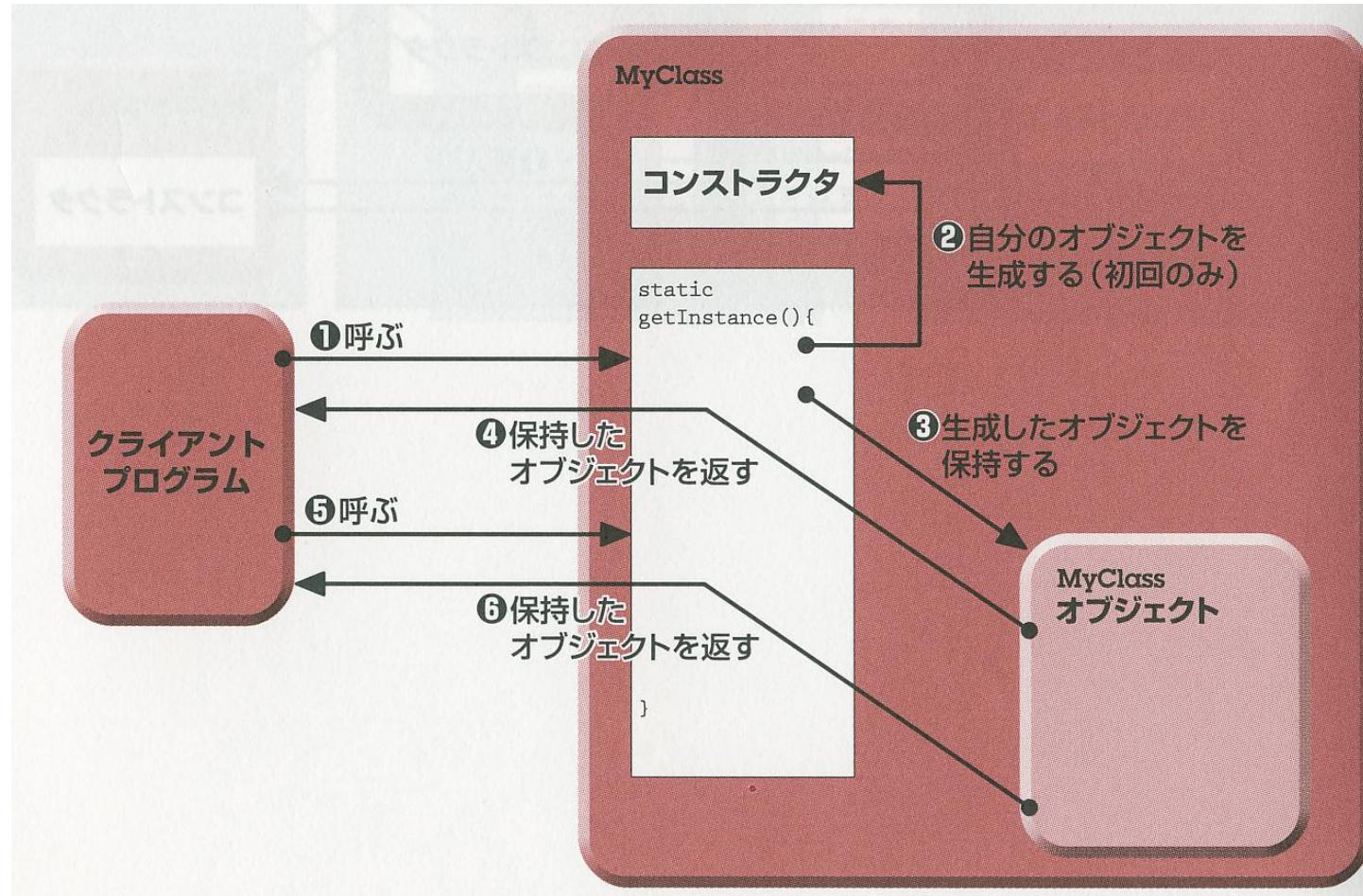
    // クローンを作る
    // (プロトタイプをコピーする)
    MyParam b1 =(MyParam)a1.clone();
    MyParam b100=(MyParam)a100.clone();
}
catch(CloneNotSupportedException e){ }
```



Singletonパターン

＜インスタンスが唯一であることを保証する＞

- オブジェクトが1つしか作れないクラス.



Singleton パターン (2)

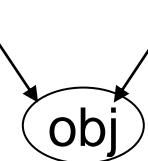
- オブジェクトをstaticで保持。2回目以降は、生成済みオブジェクトへの参照を返す。

- たとえば、大量のメモリが必要なオブジェクト、巨大画像、動画像など
- または、システムに1つしかないもの（ウィンドウ）などを表現する場合

```
class MyClass {
    static MyClass unique;
    private MyClass(){
        // private としてコンストラクタの
        // 外部呼出しを禁止にする
    } // 外部でのnewの実行の禁止
    static MyClass getInstance(){
        if (unique==null){
            unique=new MyClass();
        }
        return unique;
    }
}
```

```
// main
MyClass c1=MyClass.getInstance();
MyClass c2=MyClass.getInstance();
```

c1とc2は同じオブジェクトを参照することになる。 c1 c2



Flyweightパターンに似ているが、別にPool classがある訳ではなく、クラス自体が一度生成したインスタンスを保持している。



Prototype と Singleton のまとめ

- Prototypeは、内部状態をそのままコピーするためのパターン。Javaの言語仕様の一部として実現されている。
 - フィールドのデータを個別にコピーする必要がない。手間の節約。
- Singletonは、static classを実現するためのパターン。
 - コンストラクタをprivateにして、外部でのnewの実行を禁止し、システム内で1つだけしかインスタンスが存在しないことを保証する。



振る舞いに関するパターン(1)

■ Template Method パターン <処理をサブクラスにまかせる>

- アルゴリズムの外枠を定義しておき、アルゴリズムの構造を変えずに、その中のいくつかのステップについては、子クラスでの定義に任せるようにする。

■ Observer パターン <状態の変化を通知>

- あるオブジェクトが状態を変えた時に、それに依存するすべてのオブジェクトに自動的にそのことが通知されるような、オブジェクト間の一対多関係を定義する。

■ Iterator パターン <1つ1つを数え上げる>

- オブジェクトが内部表現を公開せずに、要素に順にアクセスする方法を提供する。

■ State パターン <状態をクラスとして表現>

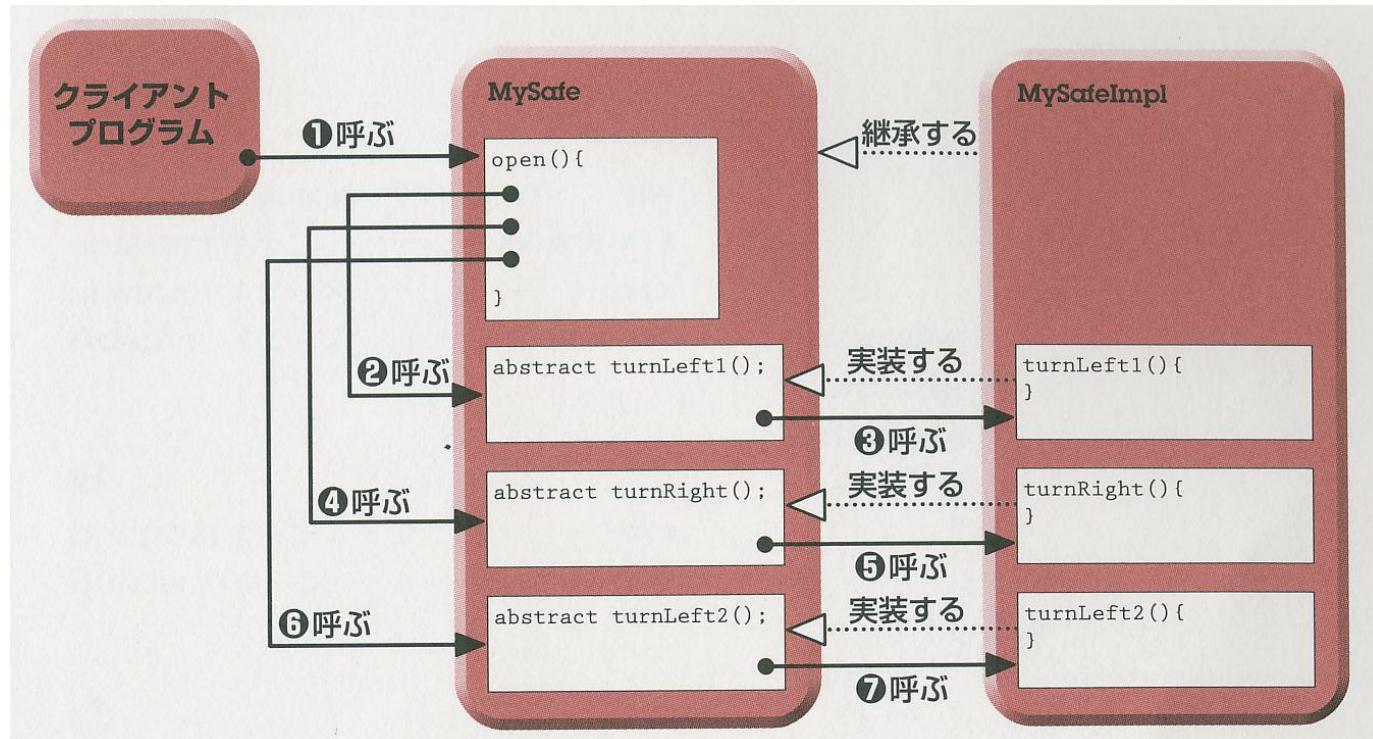
- オブジェクトの内部状態が変化した時に、オブジェクトが振舞いを変えるようにする。



Template Method

＜処理をサブクラスにまかせる＞

- 処理を細かいステップに分割して、
それぞれのステップをサブクラスで実装
サブクラスの変更によって、動作を変えられる



Template Method パターン (2)

金庫のダイヤルを右, 左, 右に回す
場面をシミュレートした例.

```
// 抽象クラスをテンプレートして準備
abstract class MySafe {
    boolean open(int pos,int key1,int
        key2,int key3){
        int result=turn1(pos, key1);
        result=turn2(result,key2);
        result=turn3(result,key3);
        if (result==pos) return true;
        else return false;
    }
    abstract protected int turn1(int p,int n);
    abstract protected int turn2(int p,int n);
    abstract protected int turn3(int p,int n);
}
```

個々の金庫の回し方はサブクラスで定義

```
// サブクラスで、抽象メソッドを実装
class MySafeImpl1 extends MySafe {
    protected int turn1(int p,int n){
        return p += n; }
    protected int turn2(int p,int n){
        return p -= n/2; }
    protected int turn3(int p,int n){
        return p += n*4; }
}
```

// main関数

```
MySafeImpl1 safe=new MySafeImpl1();
boolean result=safe.open(2,2,12,1);
if (result) System.out.println("OK !");
else System.out.println("Failed !");
```

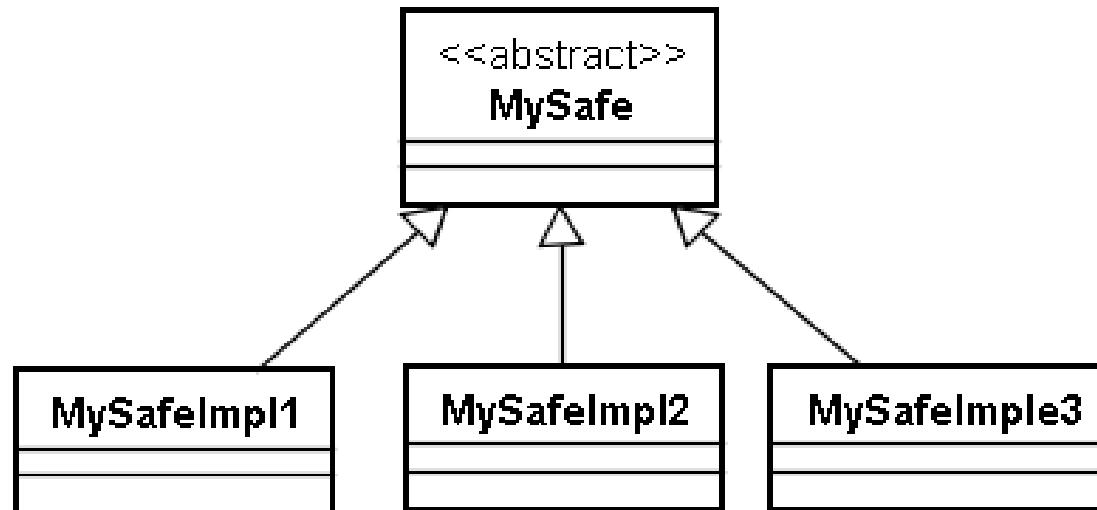


Template Method の まとめ

- 抽象クラスに、抽象メソッドを組み合わせて実現するテンプレート処理を記述.
- テンプレート処理を実際に実行するには、抽象メソッドをサブクラスで実装することが必要.
- サブクラスでの抽象メソッドの実装を変更することによって、テンプレート処理の実行結果が変化
- Bridgeパターンの実装クラス階層で利用されている.



Template Method のクラス図



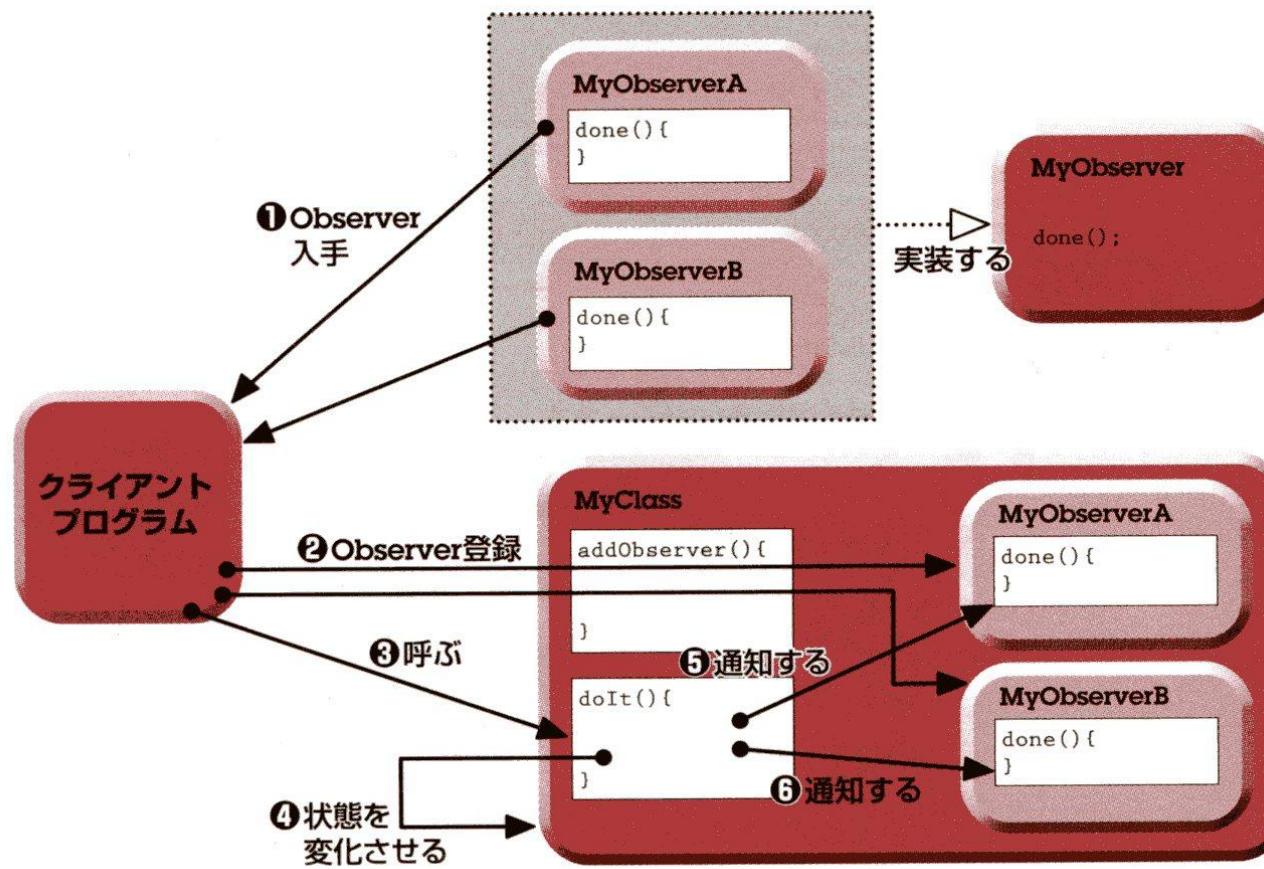
金庫を開けるための openメソッドは、子クラスで実装された turn1, tunr2, turn3メソッドを呼び出して実際には動作する。

MySafeクラスは abstract型(抽象クラス型)なので、必ず子クラスを作成しないと、利用できない。



Observer パターン <状態の変化を通知>

あるオブジェクトが状態を変えた時に、それに依存するすべてのオブジェクトに自動的にそのことが通知されるようなパターン。



Observerパターン(2)

```
// 監視する(状態変化を通知  
// される)オブジェクトのクラス  
interface Observer {  
    void update(int u);  
} // doneは更新時に呼ばれるメソッド  
class ObserverA  
    implements Observer{  
    public void update(int u){  
        System.out.println(  
            "A: updated : "+u); }  
}  
class ObserverB  
    implements Observer{  
    public void update(int u){  
        System.out.println(  
            "B: updated : "+u); }  
}
```

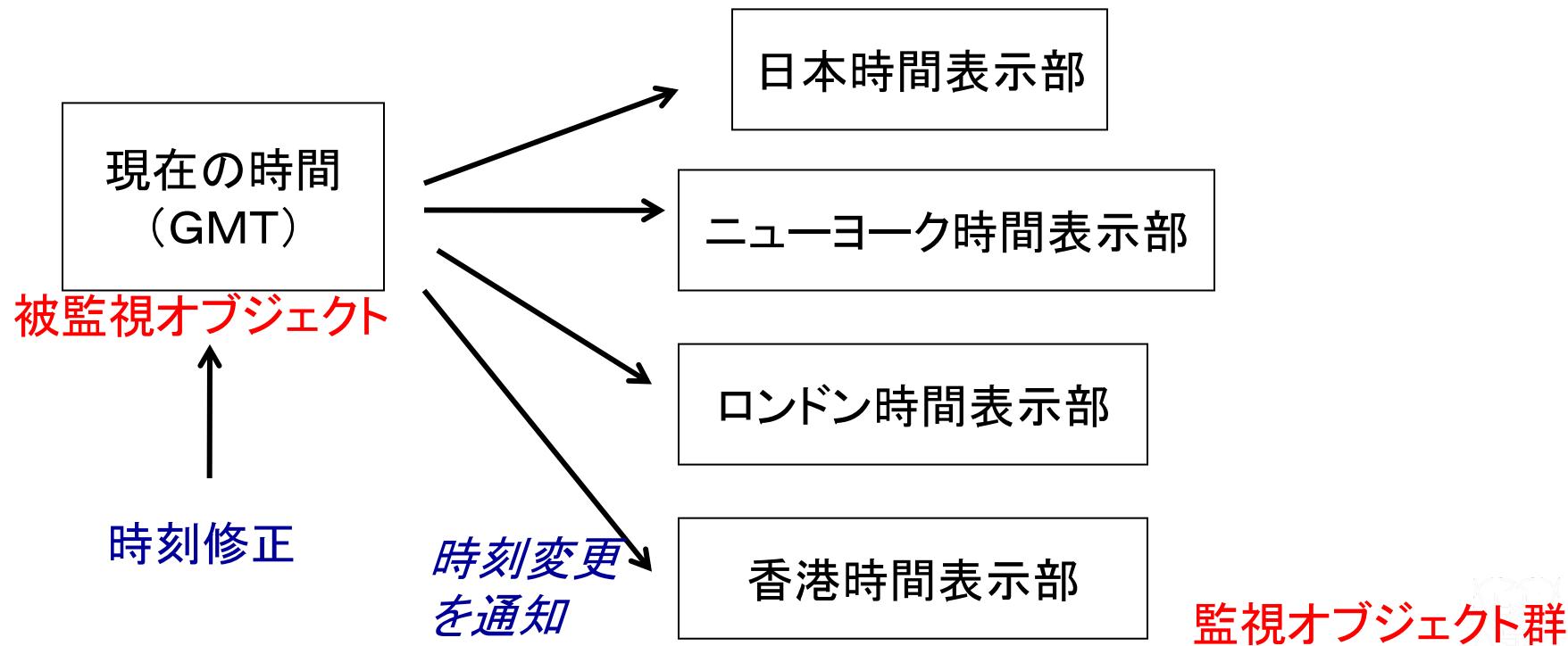
```
// 状態変化を通知するクラス(被監視)  
class MyClass{  
    int value=0;  
    Vector<Observer> observers=  
        new Vector<Observer>();  
    // observerの登録メソッド  
    void addObserver(Observer ob){  
        observers.add(ob); }  
    // observerへの更新通知メソッド  
    void doIt() {  
        value++;  
        for(int i=0;i<observers.size();i++){  
            observers.get(i).update(value); }  
    }  
}
```

Observerを実装したクラスを最初に登録し、更新時に登録クラスのメソッドdone()を呼ぶ



Observerパターンの例

- あるオブジェクトの状態変化に伴って、他のオブジェクト(observer)も変化する
 - 例) 世界時計プログラム



Observerパターン(3)

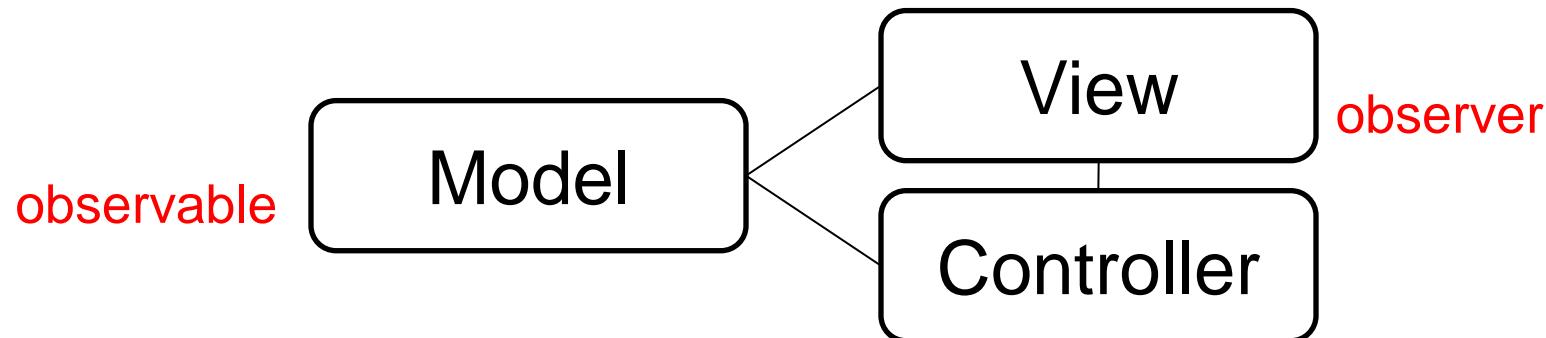
- Javaの標準ライブラリ Java.utilに
Java.util.Observerインターフェース と
Java.util.Observableクラスがある.
 - 考え方は同じ. 更新を通知する方が, Observable
を継承する.
 - ただし, Observableがクラスであるので,
更新を通知するクラスは他のクラスを継承できな
い. (欠点)(継承は1つのクラスのみなので)
- Observerパターンは, アーキテクチャパター
ンのMVCモデルと関係が深い



MVCモデル(1)

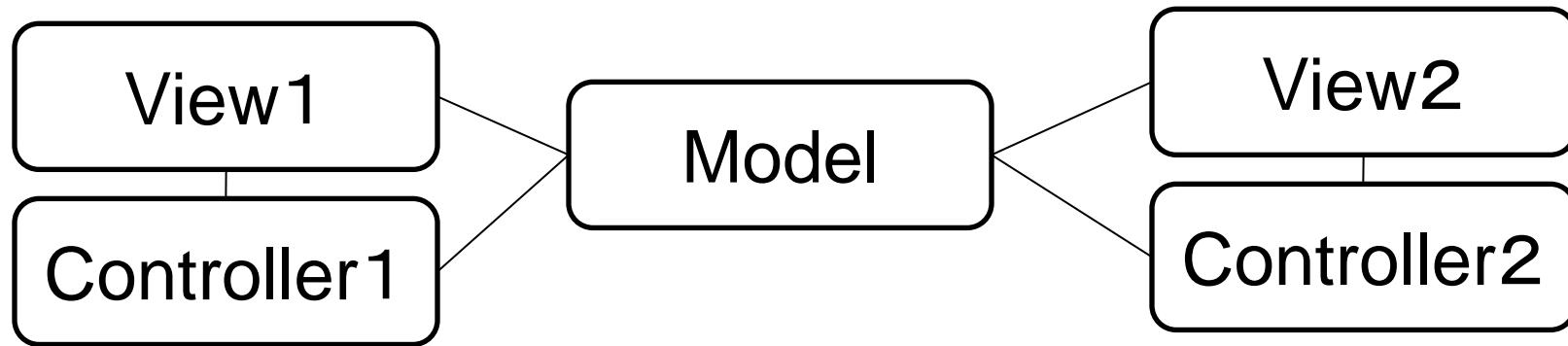
■ Model, View, Controllerの3つに分けて、システムを設計する方法

- できるだけ、独立に設計する
 - Model: データ操作
 - View: 出力. モデルの状態に応じて出力を変化させる.
 - Controller: 入力



MVCモデル(2)

■ 例:スクリーンエディターの作成



- Model(データ操作のクラス)は同一で、VCを交換することによって、全く見た目の異なる(例えば、GUIとCUI)エディタを実現可能。

<http://jikken.cs.uec.ac.jp/soft/oop/conv.cgi?URL=oop3/kadai.html>

<http://jikken.cs.uec.ac.jp/soft/oop/conv.cgi?URL=oop3/observer.html>

MVCモデル と observerパターン

■ MVCモデル:

Model, View, Controllerを分離して設計する設計モデル

C, Vがboundary, Mがcontrolに相当する。

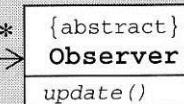
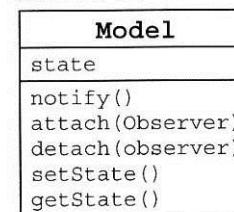
- SmallTalkで用いられた
- Javaの設計思想にも取り入れられている。

■ Observerパターン:

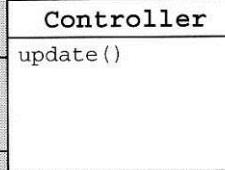
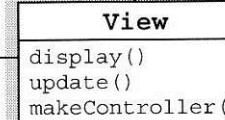
デザインパターンの一種
observer がsubjectの状態変化を監視して、変化があれば、何らかのアクションを起こす。

図 C-1 MVC モデルと Observer パターン

(a)MVC モデル



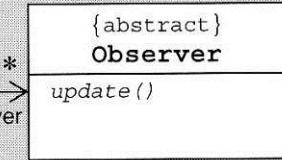
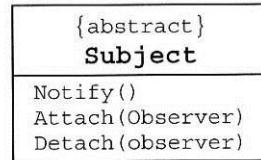
*



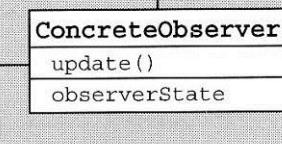
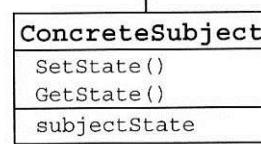
生成する▶
◀表示を指示する

*

(b)Observer パターン



observer



subject

MVCモデル と observerパターン

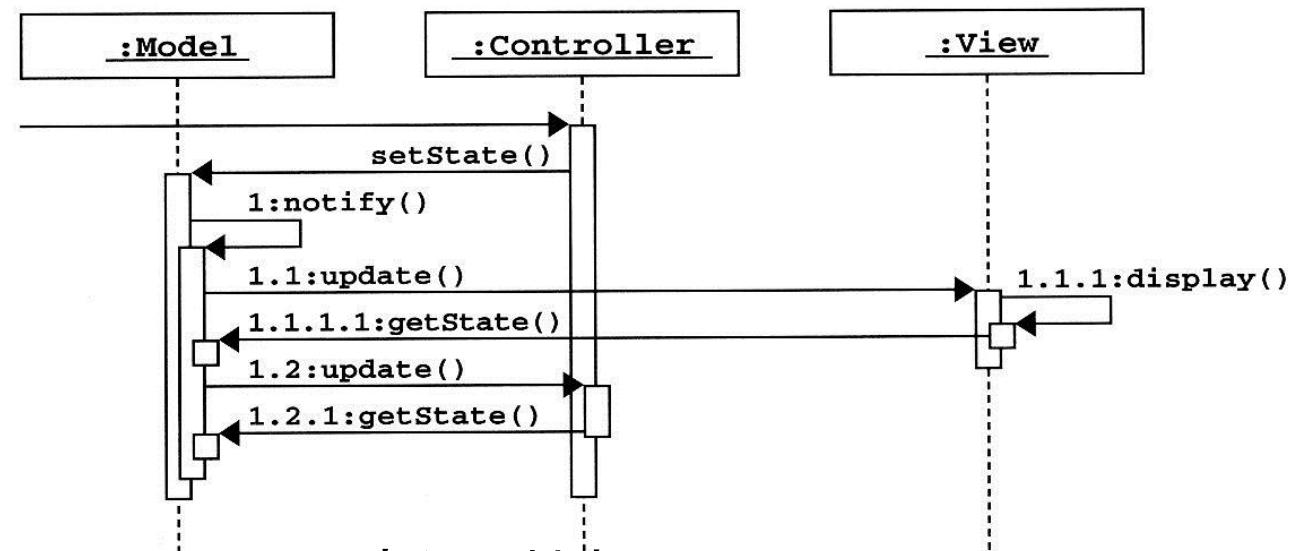
Model: データ構造の中心（基本データ構造 と その操作）

View: 表示のクラス（GUIの場合は GUI部品のクラス）

Controller: 入力のクラス（GUIの場合はイベントハンドリングのクラス）

図 C-2

MVC モデルにおけるデータ変化通知のシーケンス図



Observer パターンによって Model の変化に対応して
View を更新(update)する



振る舞いに関するパターン (2)

- **Chain of Responsibility パターン <責任の連鎖>**
 - 複数のオブジェクトを鎖状につなぎ、要求をその鎖に沿って次々と渡すようにすることによって、要求送信オブジェクトと受信オブジェクトの直接の結合を避ける。
- **Command パターン <命令のクラス化>**
 - 要求をオブジェクトとしてカプセル化して、クライアントをパラメータ化する
- **Interpreter パターン <文法規則をクラスで表現>**
 - 言語の文法表現と、それを利用して文を解釈するインタプリタを定義する
- **Mediator パターン <調整役のオブジェクトを定義>**
 - オブジェクト群の相互作用をカプセル化するオブジェクトを定義する。



振る舞いに関するパターン (3)

■ Memento パターン <状態を保存する>

- オブジェクトのカプセル化を壊さずに、オブジェクトの内部状態を保存し、後でこの状態に戻すことができるようにする。Undoを実現するためのパターン。

■ Strategy パターン <アルゴリズムを簡単に交換>

- アルゴリズムの集合を定義し、各アルゴリズムをカプセル化してそれらを交換可能にする。

■ Visitor パターン <処理をデータ構造から分離>

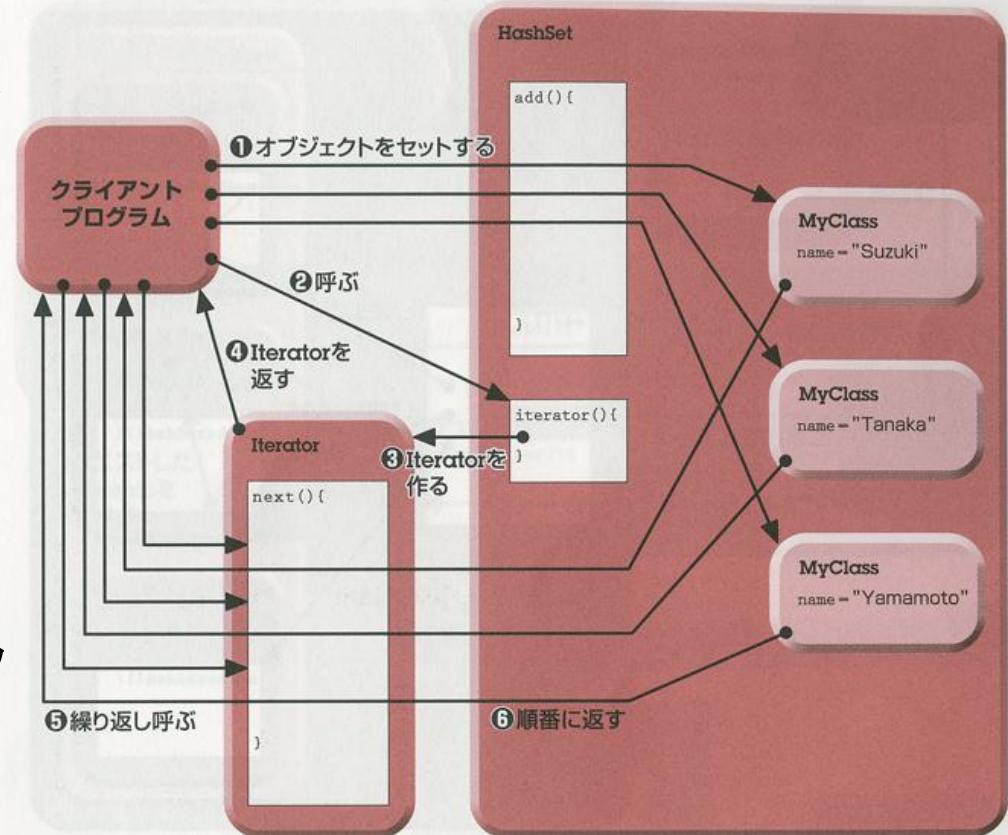
- 関連する操作を各クラスから取り出して別オブジェクトにまとめることにより、新しい操作の定義を容易にする。



Iterator パターン

<1つ1つを数え上げる>

- データ構造にかかわらず、蓄積されたオブジェクトを1つ1つ順番にアクセスする。



- `Java.util.Iterator` インターフェースがある。`Java.util`のデータ型クラスは、ほぼすべて、`iterator` オブジェクトを返す`iterator`メソッドを実装している。



Iteratorパターン(2)

```
// Javaの標準インターフェース
interface java.util.Iterator<E>{
    boolean hasNext();
    E next();
    void remove();
    //removeは最後の要素を削除。
    //実際には何もしないメソッドでよい。
}

class MyIterator()
    implements Iterator<String>{
    int idx=0;
    vector<String> v; // Vectorの参照を保持
    MyIterator(vector<String> vs){
        v=vs; idx=v.size();}
    boolean hasNext(){return (idx>0);}
    String next() { return v.get(--idx);}
    void remove() {} }
```

```
class MyClass{
    vector<String> v=new vector<String>();
    void add(String s){ v.add(s); }
    // MyIteratorオブジェクトを返すメソッド
    MyIterator iterator(){
        return new MyIterator(v); }
} // Iteratorオブジェクトは再利用不可。
// 一度スキャンしたら、後は捨てる

// main関数
MyClass c=new MyClass();
c.add("Suzuki");
c.add("Tanaka");
MyIterator it=c.iterator();
While(it.hasNext()){
    System.out.println(it.next());
}
```

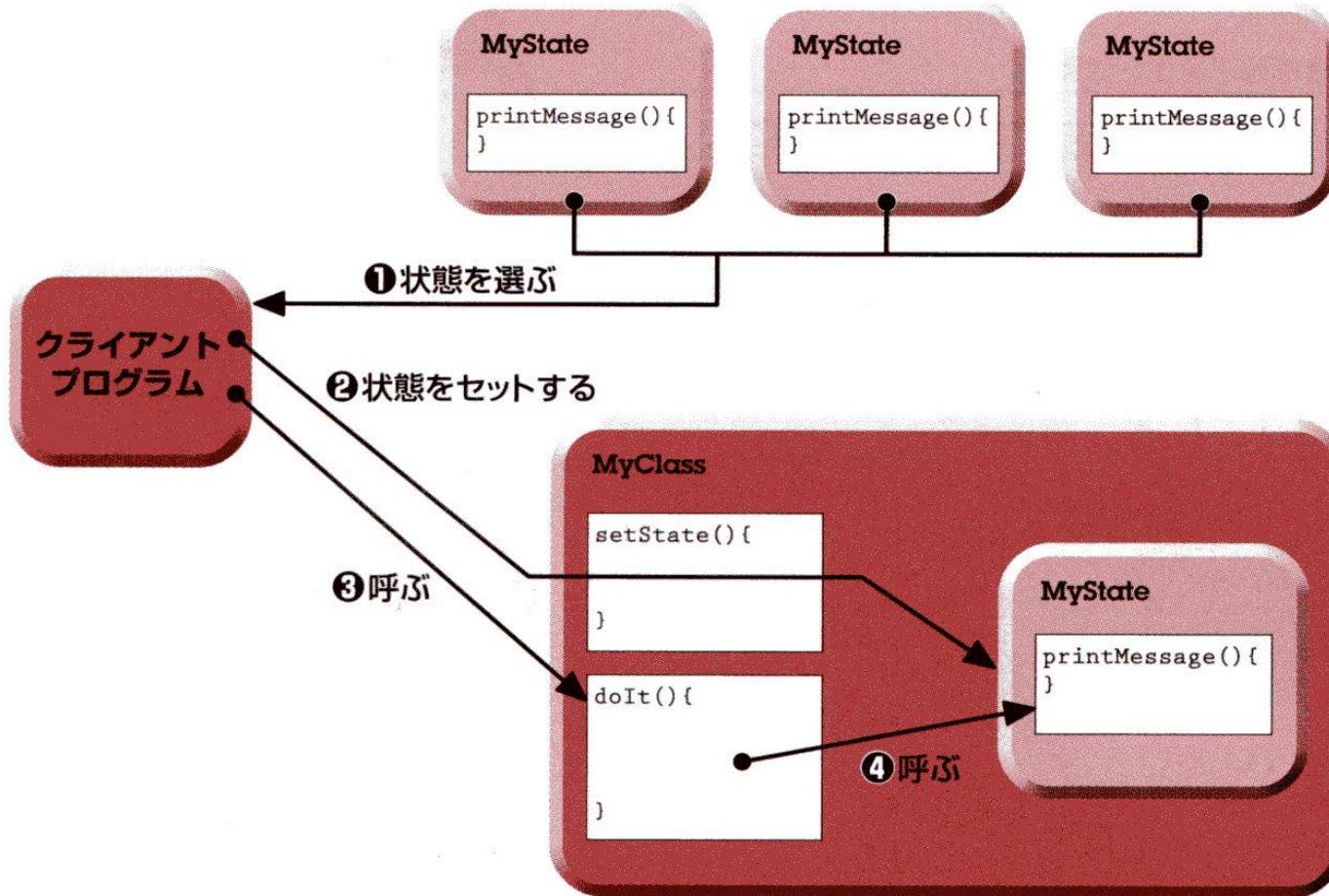
Iteratorパターン(3)

- Java.util.Iteratorインターフェースがある。
 - Javaのライブラリに取り込まれている。
- Java.util内の多くのデータ構造クラス(vector, Hashmap, LinkedListなど)は、Iteratorオブジェクトを返すiterator()メソッドを実装している。
 - ハッシュのような通常は逐次アクセス不可能なデータ構造も1つ1つデータをアクセスすることが可能となる。



Stateパターン (1)

＜状態をクラスとして表現＞



オブジェクトの状態をオブジェクトを使って表す。ポリモーフィズムを利用して、IF文を排除。状態の追加が容易に行える。

Stateパターン(2)

```

class Book{
    String name;
    State currentState;
    Book(String n,State s){
        name=n; currentState=s; }
    void setState(State s){
        currentState=s; }
    void printState(){
        System.out.println("現在の"+
            name+"の状況:"+currentState);
    }
}

interface State{ }
class StateIn implements State{
    public String toString () {
        return "図書館にあります. ";
    }
}

```

```

class StateOut implements State{
    public String toString () {
        return "貸し出し中です. ";
    }
}

// main関数
Book b1=new Book("ソフトウェア工学",
    new StateIn());
Book b2=new Book("デザインパターン",
    new StateOut());

```

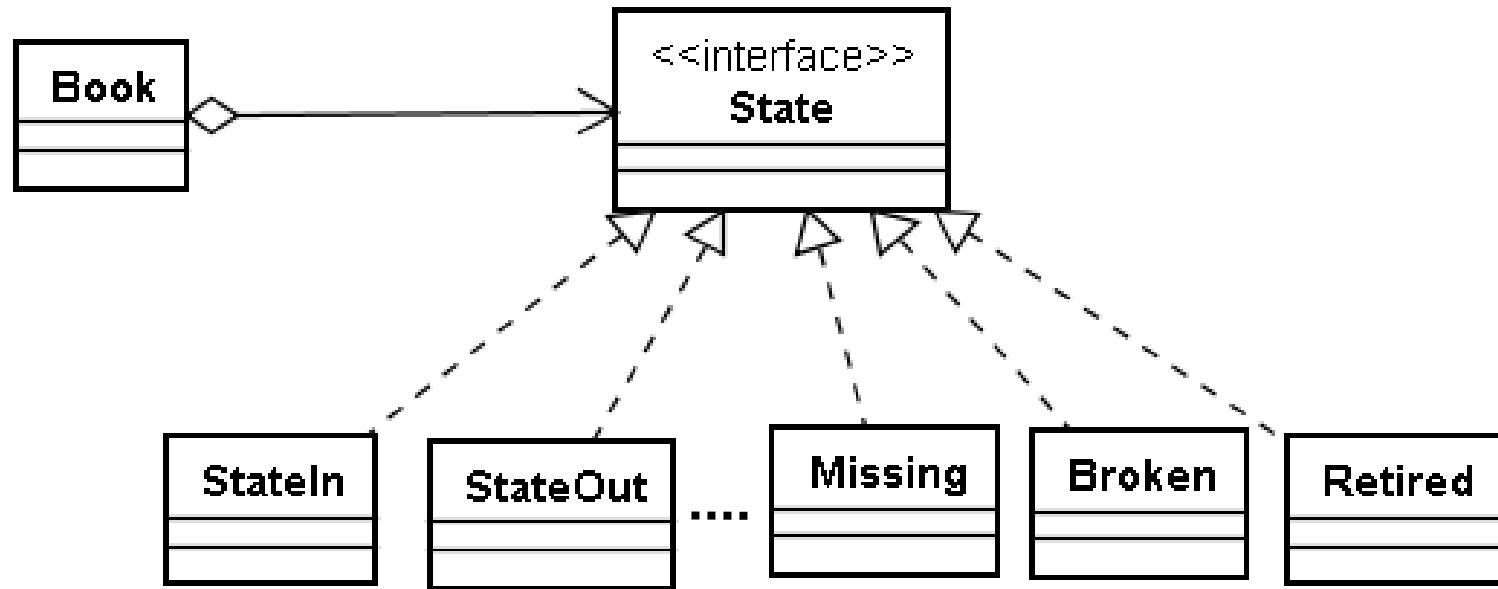
```

b1.printState();
b1.setState(new StateOut());
b1.printState();

```

この貸し出し状態をStateIn, StateOutのオブジェクトで表現. Bookクラスは、Stateのみに依存し、その実装クラスに依存しないので、状態のクラスが増えても変更する必要がない。IF文を用いる必要がない。

State パターン のクラス図



状態を他のクラスを書き換えることなく、簡単に追加できる。

追加した状態の固有の動作は、ポリモーフィズムによって実現。



オブジェクト生成に関するパターン

■ Factory Method パターン <インスタンス生成をサブクラスにまかせる>

- オブジェクトを生成する時のインターフェースだけを規定して、インスタンス化を子クラスに任せるようにする。

■ Abstract Factory パターン <関連するオブジェクトを組合せて生成>

- 互いに関連しあうオブジェクト群を、そのクラスを明確にせずに生成できるようにする。

■ Builder パターン <複雑なインスタンスを組み立てる>

- オブジェクトの作成過程を表現形式に依存しないようにして、同じ作成過程で異なる表現形式のオブジェクトを生成できるようにする。

■ Prototype パターン <コピーしてインスタンスをつくる>

- 生成すべきオブジェクトの種類の原形となるインスタンスを明確にし、これをコピーすることによって新たなオブジェクトを生成する。

■ Singleton パターン <インスタンスが唯一であることを保証する>

- あるクラスに対してインスタンスが一つしかないことを保証し、それにアクセスするための方法を提供する。



振る舞いに関するパターン(1)

■ Template Method パターン <処理をサブクラスにまかせる>

- アルゴリズムの外枠を定義しておき、アルゴリズムの構造を変えずに、その中のいくつかのステップについては、子クラスでの定義に任せるようにする。

■ Observer パターン <状態の変化を通知>

- あるオブジェクトが状態を変えた時に、それに依存するすべてのオブジェクトに自動的にそのことが通知されるような、オブジェクト間の一対多関係を定義する。

■ Iterator パターン <1つ1つを数え上げる>

- オブジェクトが内部表現を公開せずに、要素に順にアクセスする方法を提供する。

■ State パターン <状態をクラスとして表現>

- オブジェクトの内部状態が変化した時に、オブジェクトが振舞いを変えるようにする。



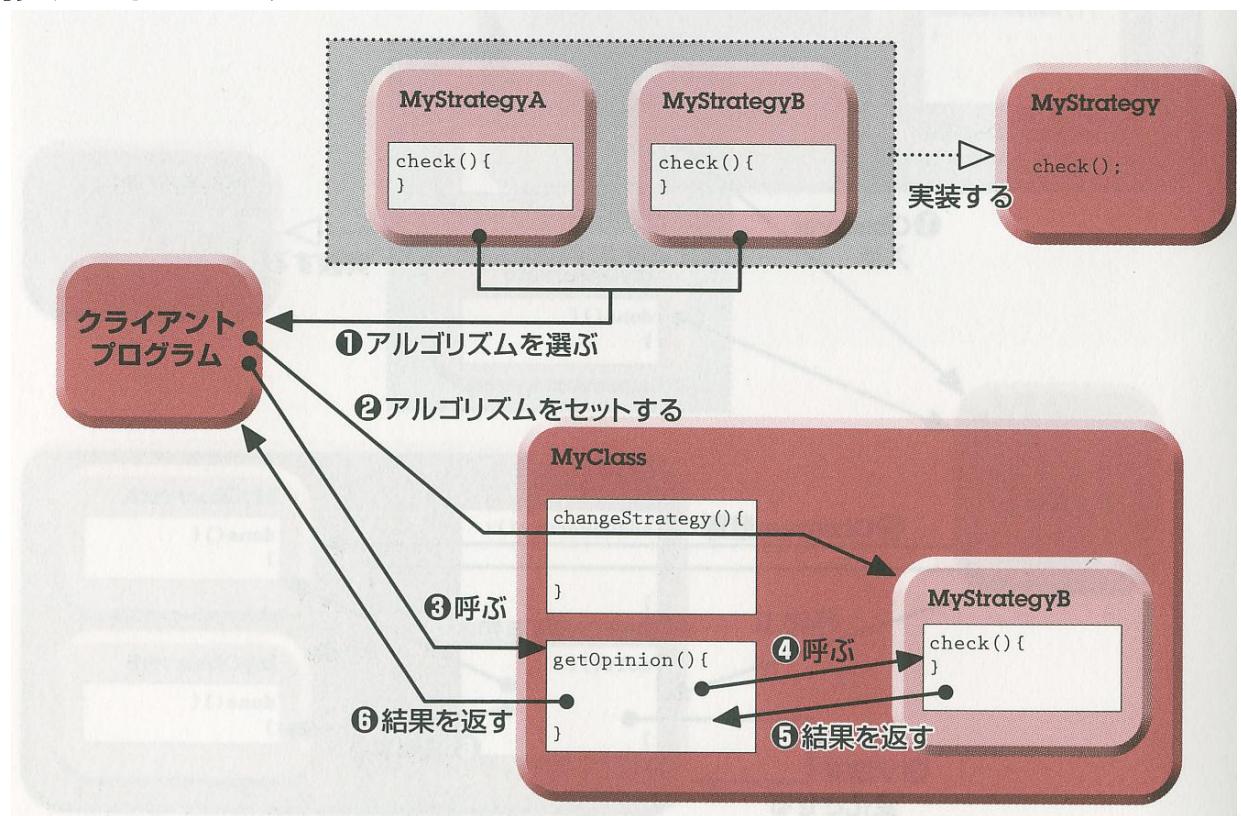
Strategy パターン

<アルゴリズムを簡単に交換>

- アルゴリズムの集合を定義し、各アルゴリズムをカプセル化してそれらを交換可能にする。

各アルゴリズムを同一のインターフェースを実装したクラスとして準備し、ポリモーフィズムによって簡単に切り替えられるようとする

If文による切り替えを用いない。



Strategy パターン(2)

```

class MyClass {
    Strategy currentStrategy;
    MyClass(MyStrategy m)
    { currentStratedy=m; }
    void changeStrategy(Strategy m)
    { currentStrategy=m; }
    String getOpinion(int p){
        int result=currentStrategy.check(p)
        if (result>0) return "good";
        else if (result==0) return "so-so";
        else return "bad!";
    }
}
interface Strategy {
    int check(int p);
}

```

Strategyの切り替えによって、動作を変更する。Stateパターンと同様に、Strategyの追加が容易。

```

class StrategyA implement Strategy{
    public int check(int p){
        if (p>90) return 1; else return -1;}
}

```

```

class StrategyB implement Strategy{
    public int check(int p){
        if (p>50) return 1;
        else if (p>30) return 0;
        else return -1;}
}

```

```

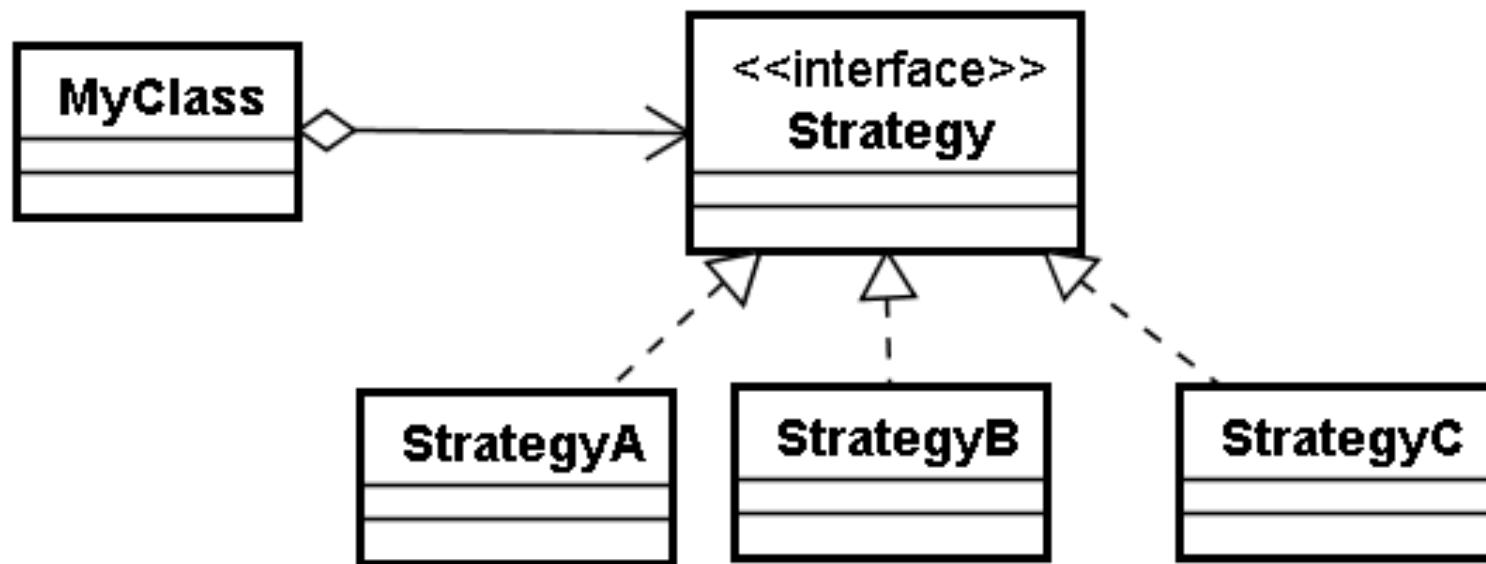
// main
MyClass m= new MyClass(new StrategyA());
System.out.println(m.getOpinion(80));
m.changeStrategy(new StrategyB());
System.out.println(m.getOpinion(80));

```



Strategyパターンのクラス図

- Stateパターンと全く同じになる。
- Strategyの交換で、振る舞いが変わる



Command パターン

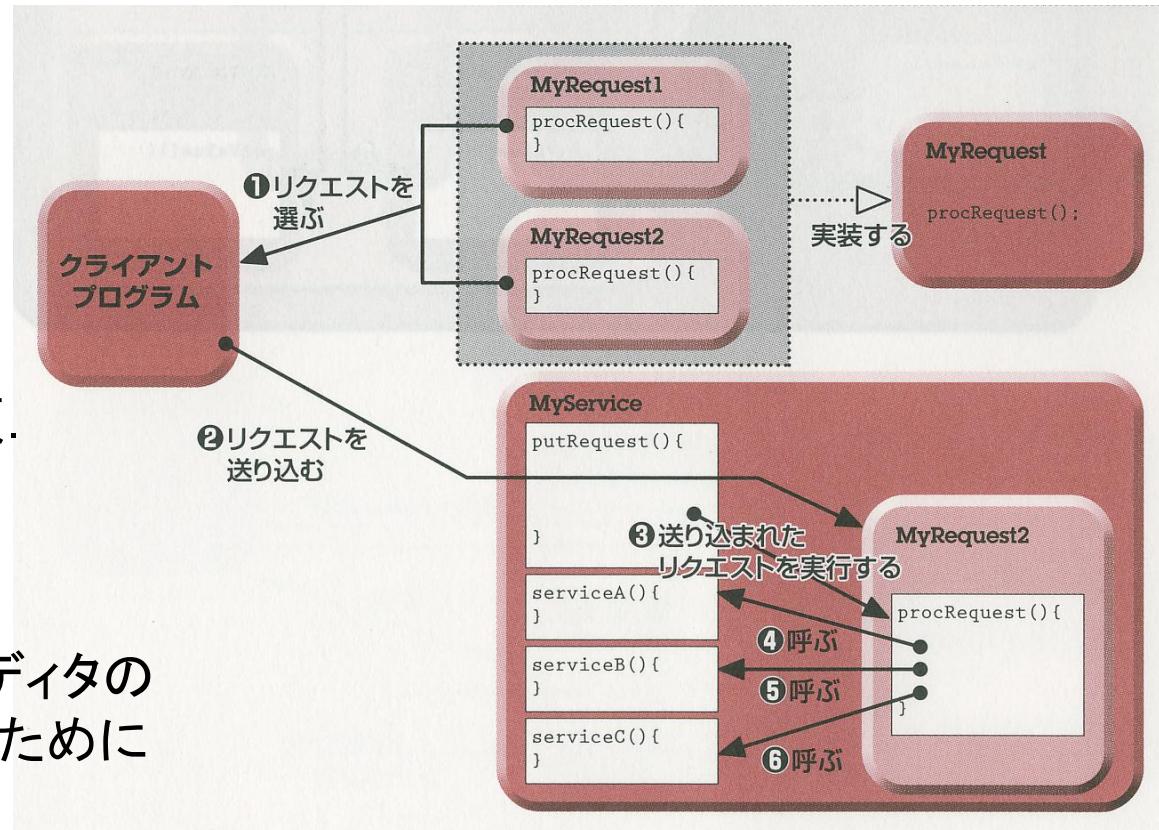
＜命令のクラス化＞

- 命令(要求)をオブジェクトとして表現する。
オブジェクトを保存することによって、命令の履歴を
容易に保存可能。

実行手順を保存し、
再現可能とする。

例1) ラーメンを作る手順。
対象のどんぶりのみ変更
して、再現する。

例2) J3課題のドローエディタの
paint() メソッド。再描画のために
履歴を記録している。



Commandパターン(2)

// コマンドを表現する抽象クラス

```
abstract class MyRequest{
```

```
    MyService service;
```

```
    void setService(MyService s){
```

```
        service=s; }
```

```
    abstract void procRequest(); } // 命令(要求)を実行する抽象メソッド
```

```
class MyRequest1 //コマンド実行クラス
```

```
    extends MyRequest {
```

```
        void procRequest(){ //  
            service.serviceA(); } 味噌ラーメン  
            を作る手順
```

} // 実際には色々なMyRequest を用意する

```
class HistRequest //コマンド履歴クラス
```

```
    extends MyRequest {
```

```
        Vector<MyRequest> vec =  
            new Vector<MyRequest>();
```

```
        void Add(MyRequest req){
```

```
            vec.addElement(req); }
```

```
        void procRequest(){
```

```
            for(int i=0;i<vec.size();i++)
```

```
                vec.elementAt(i).procRequest();}}
```

```
class myService{ // 実行対象で,  
    int num; // 対象に応じたメソッドを持ったクラス  
    MyService(int i){ num=i; }  
    void putRequest(MyRequest req){  
        req.setService(this);  
        req.procRequest(); } 丼(numが大きさ)  
    void serviceA(){ // 基本動作メソッド(通常複数用意する)  
        System.out.println("A:"+num); } ラーメンを作る基本動作メソッドを用意する
```

// main関数

```
MyService s=new MyService(10);
```

// コマンドオブジェクトrを生成, 実行

```
MyRequest r=new MyRequest1();
```

```
s.putRequest(r);
```

// 実行履歴を保存

```
HistRequest hist=new HistRequest();
```

```
hist.Add(r);
```

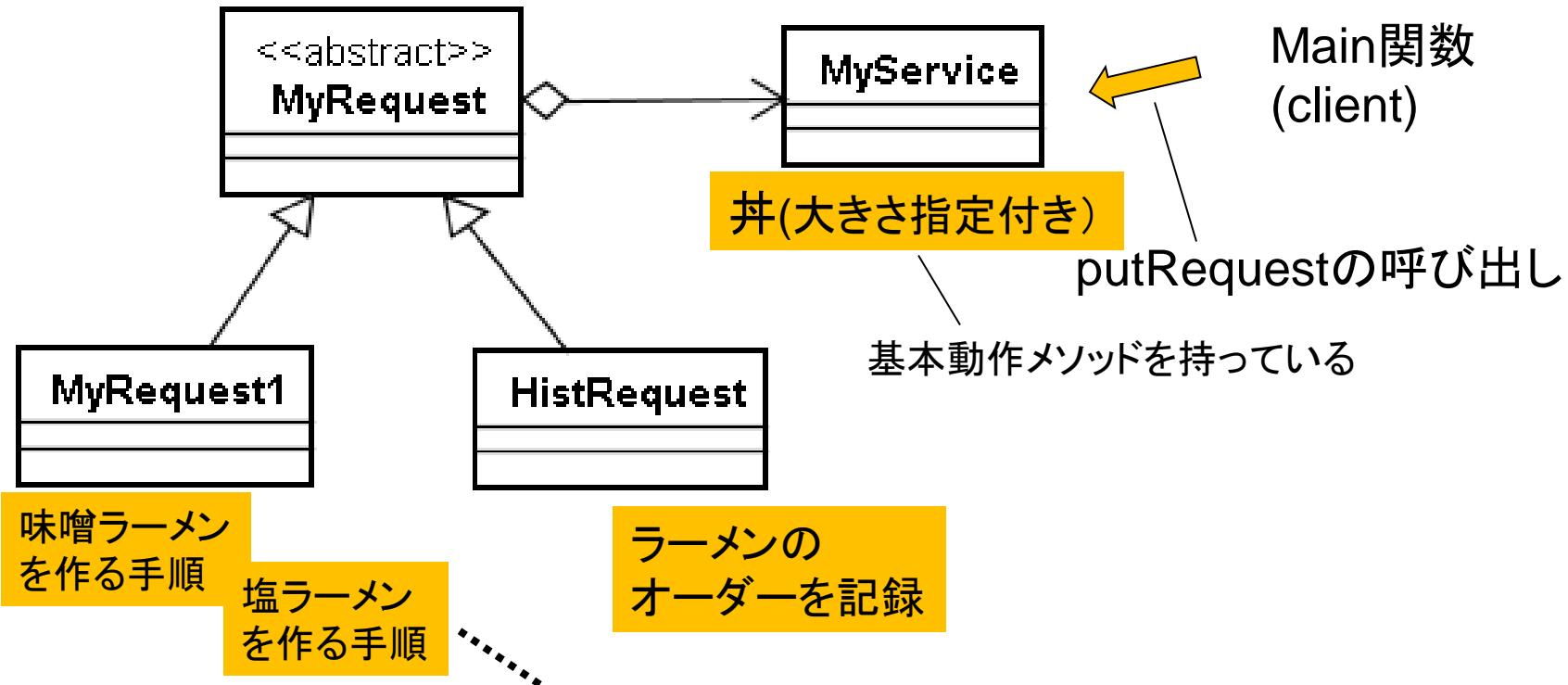
// 履歴を別のオブジェクトに対して実行する

```
MyService s2=new MyService(100);
```

```
s2.putRequest(hist);
```

Commandパターンのクラス図

- MyServiceは、動作対象であり、
対象に応じた処理メソッドも持つ。



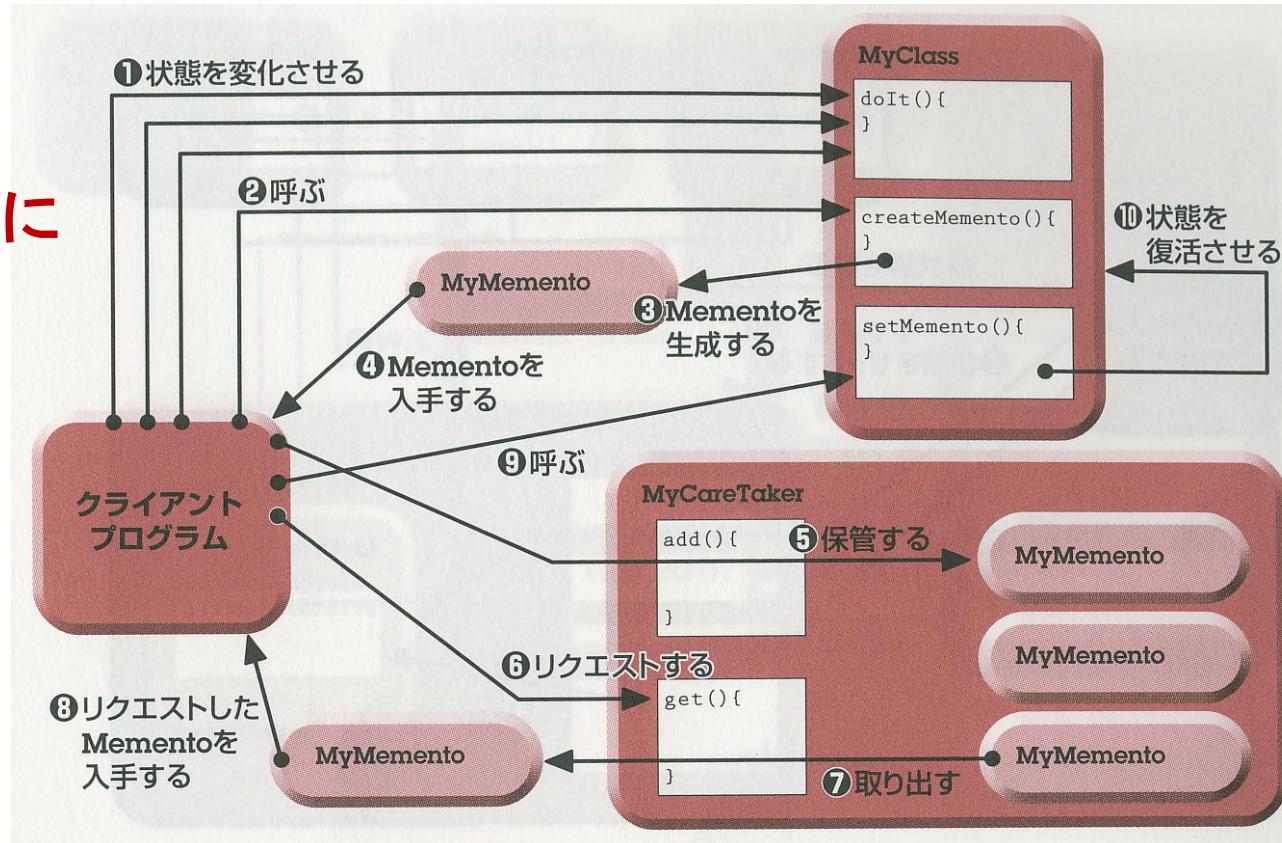
Memento パターン（形見、記念品）

＜状態の保存＞

- オブジェクトのカプセル化を壊さずに、オブジェクトの内部状態を保存し、後でこの状態に戻すことができるようにする

ミメントオブジェクトにある瞬間の状態を保持して、すべて保存しておく

Undo のためのパターン



Memento パターン(2)

```

class MyClass {
    int val1=0;
    String val2="";
    void doIt() { val1++; val2+="★"; }
    MyMemento createMemento(){
        return new MyMemento(val1,val2);}
    void setMemento(MyMemento mem){
        val1=mem.val1;
        val2=new String(mem.val2); }
    }
class MyMemento {
    int val1;
    String val2;
    MyMemento(int v1,String v2){
        val1=v1; val2=new String(v2); }
    }
// ミメントオブジェクトをプール
class MyCareTaker {

```

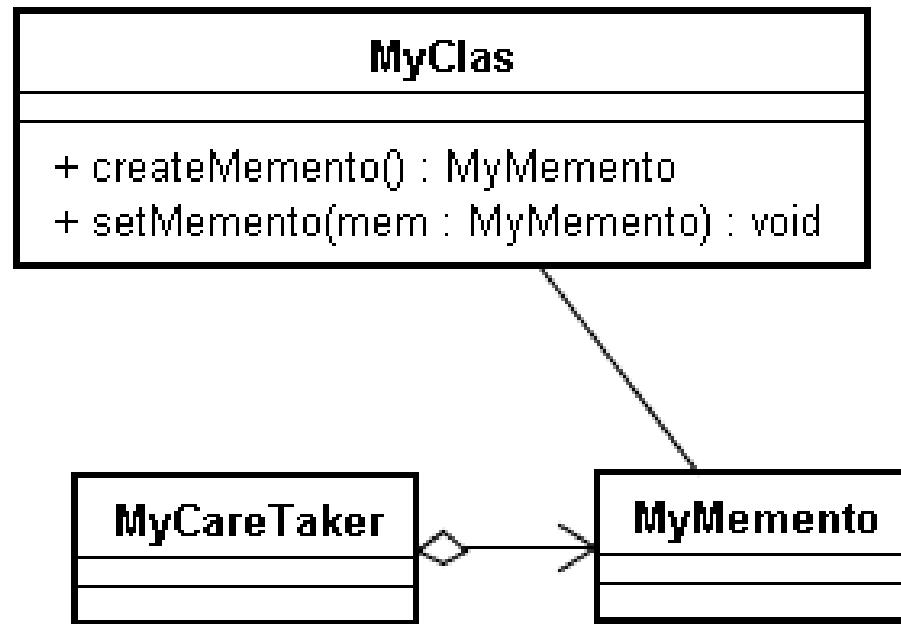
```

Vector<MyMemento> pool=new
    Vector<MyMemento>();
void add(MyMemento m){
    pool.add(m); }
// n回前を取り出し.
MyMemento get(int n){
    return pool.elementAt(pool.size()-n); }
}
// main
MyCareTaker ct=new MyCareTaker();
MyClass m=new MyClass();
m.doIt();
ct.add(m.createMemento());
m.doIt();
ct.add(m.createMemento());
// 2回前に戻す
m.setMemento(ct.get(2));

```

Mementoパターンのクラス図

- 状態を保持したいクラスは、createMementoとsetMementoメソッドを実装する。



Command と Memento パターン

■ Commandで履歴を保存する場合

- 命令(要求)をオブジェクトで表現し,
オブジェクトをそのまま履歴として保存する.
そのまま保持するだけなので、実装が簡単.
- 余計なデータまで保持する可能性がある.
メモリの無駄.

■ Memento:

- 必要な情報のみをコピーできる.
容易に復元できるデータや作業用変数は
保存しないようにできる. メモリの節約.

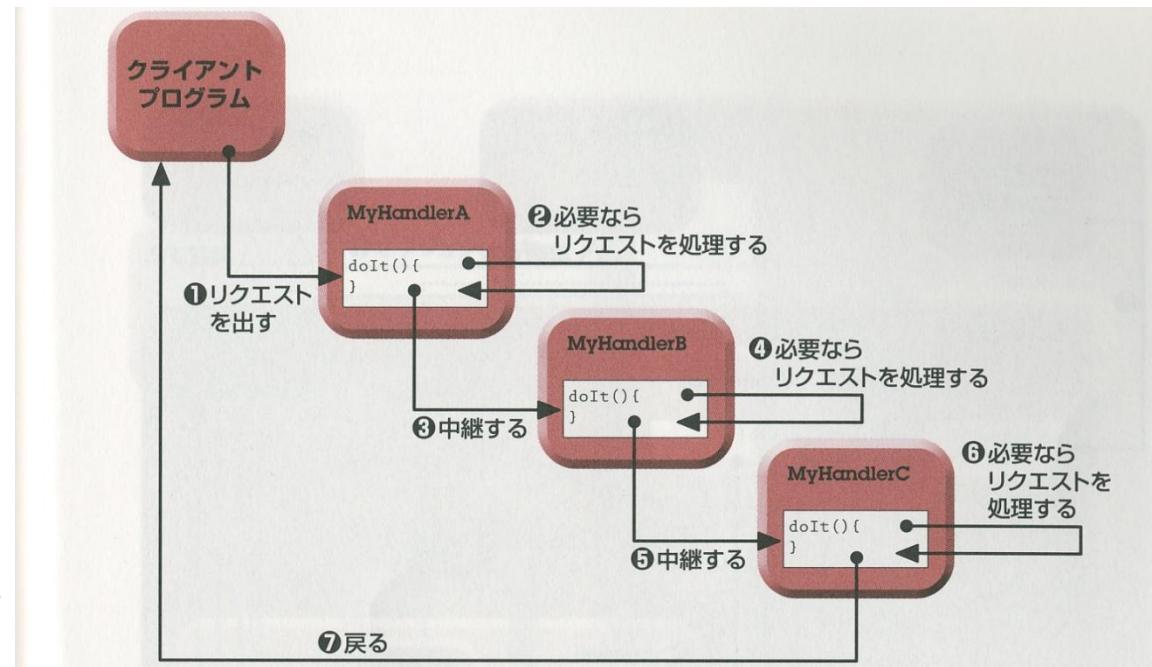


Chain of Responsibility パターン(1)

＜責任の連鎖＞

- 複数のオブジェクトを鎖状につなぎ、要求をその鎖に沿って次々と渡すようにすることによって、最終的に目的のオブジェクトを決める方法。

- たらい回し。
- 下請けに出す
- 利点：
handlerの追加が容易
if 条件の追加が不要



Composite と似ているがこちらは振る舞いのパターン。
Compositeは、(データ)構造のパターン。



Chain of Responsibility パターン(2)⁹⁷

```
interface Handler{
```

```
    void doIt(int parm); }
```

```
class MyHandlerA
```

```
    implements Handler {
```

```
    Handler next;
```

```
    MyHandlerA(Handler n) {next=n;}
```

```
    void doIt(int parm){
```

```
        if (parm%2==0)
```

```
            System.out.println("Do it by A");
```

```
        else next.doIt(parm); }
```

```
}
```

```
class MyHandlerB
```

```
    implements Handler {
```

```
    Handler next;
```

```
    MyHandlerB(Handler n) {next=n;}
```

```
    void doIt(int parm){
```

```
        if (parm>1)
```

```
            System.out.println("Do it by B");
```

```
        else next.doIt(parm); } }
```

```
class MyHandlerC implements Handler{
```

```
    void doIt(int parm){
```

```
        if (parm<=0)
```

```
            System.out.println("Do it by C");
```

```
        else System.out.println("No one can do");}
```

```
}
```

```
// main関数
```

```
Handler h=new MyHandlerA(new
```

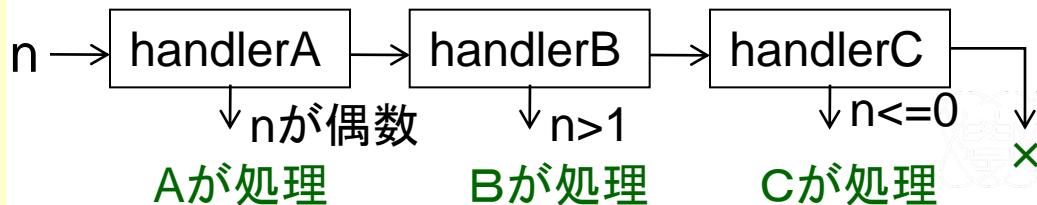
```
MyHandlerB(new MyHandlerC()));
```

```
h.doIt(3); // Bで処理される
```

```
h.doIt(2); // Aで処理される
```

```
h.doIt(-1); // Cで処理される
```

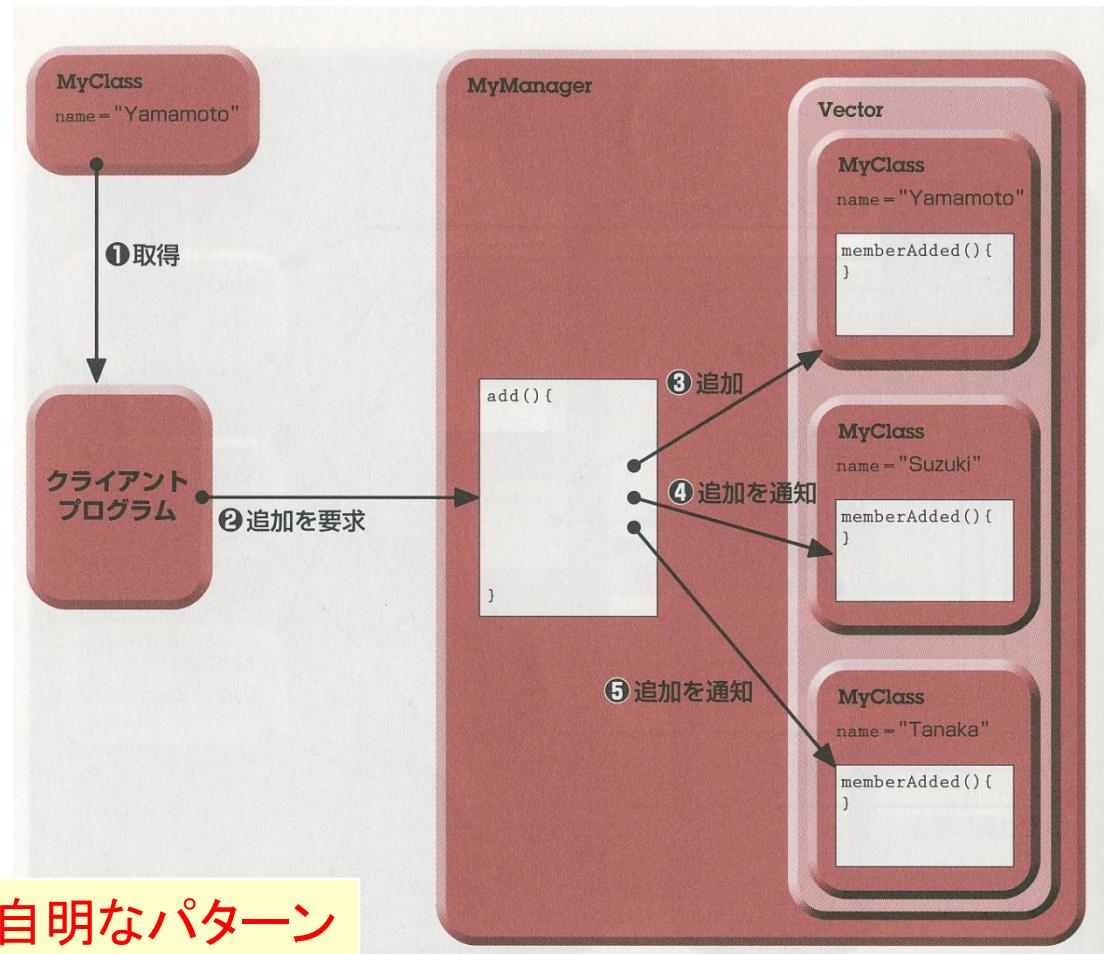
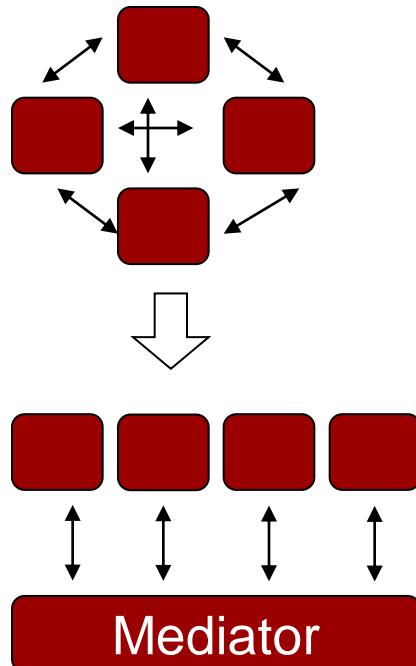
呼び出しは同一であるが、値によって
実際に処理されるオブジェクトが異なる。



Mediator パターン

＜調整役のオブジェクトを定義＞

- オブジェクト群の相互作用をカプセル化するオブジェクトを定義する。



n対n が 1対n になる。
互いを知らなくてよい。



Mediator パターン(2)

```

class MyClass{
    String name;
    MyClass(String n){name=n;}
    // 新メンバが追加されると呼ばれる
    void memberAdded(String n){
        System.out.println(name+
            ": Welcome, "+n); }
}

class MyManager {
    Vector<MyClass> mem=
        new Vector<MyClass>();
    void add(MyClass m){
        for(int i=0;i<mem.size();i++)
            mem.elementAt(i).
                memberAdded(m.name);
        mem.add(m);
    }
}

```

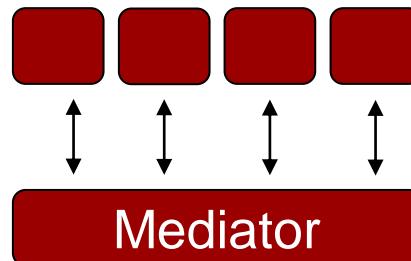
// main

```

MyManager maneger=new MyManager();
MyClass m1=new MyClass("Onai");
MyClass m2=new MyClass("Yanai");
MyClass m3=new MyClass("Numao");

manager.add(m1);
manager.add(m2);
manager.add(m3);

```



実行結果)

Onai: Welcome, Yanai
 Onai: Welcome, Numao
 Yanai: Welcome, Numao

新メンバーが加わると, mediatorが従来からのメンバー全員に連絡する



Visitor パターン

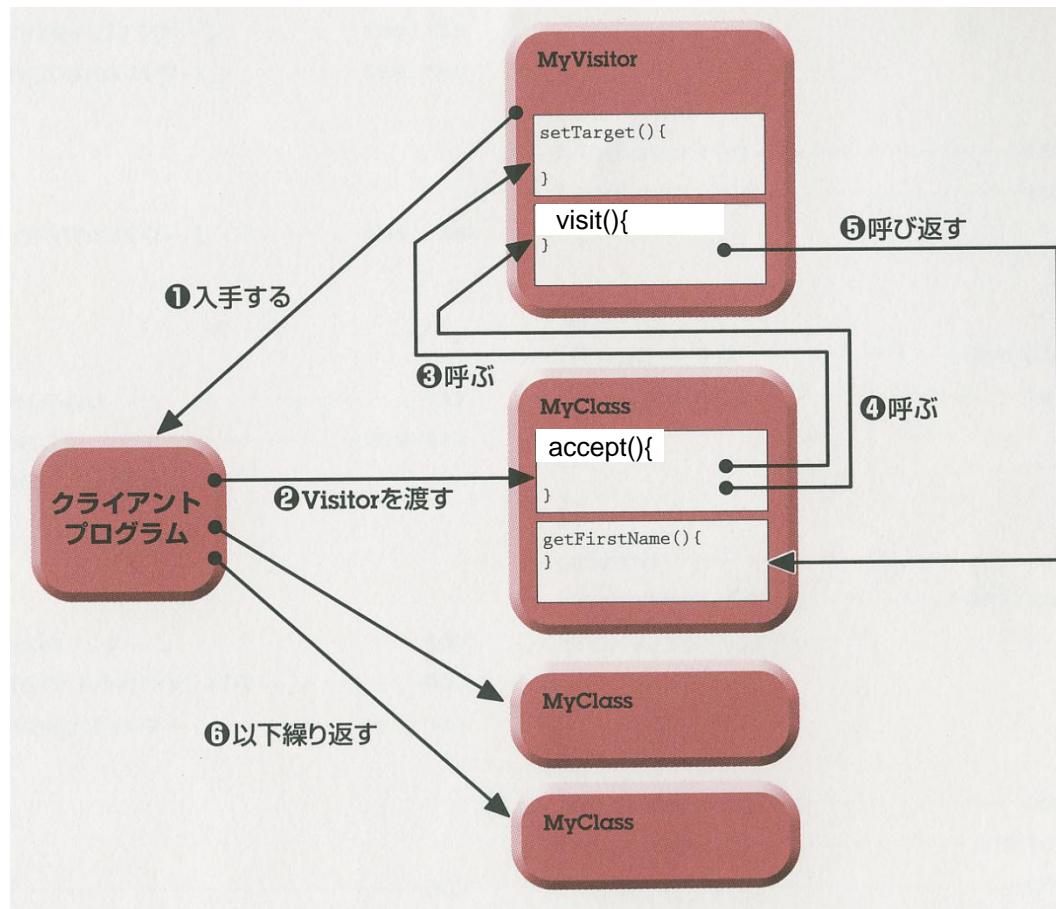
〈処理をデータ構造から分離〉

- 関連する操作を各クラスから取り出して別オブジェクトにまとめることにより、新しい操作の定義を容易にする。

- データ構造を渡り歩く「訪問者」クラスを用意して、「訪問者」クラスに処理をまかせる

- 新しい処理を追加する時は、新しい「訪問者」を追加すればよい。

- データ構造は配列、リストなど何でもよい。



Visitor パターン(2)

```

class MyClass { //訪問を受けるクラス
    private String firstName;
    private String lastName;
    MyClass(String f, String l){
        firstName=f; lastName=l; }
    String getFName() {return FirstName;}
    String getLName() {return lastName;}
    void accept(Visitor vistor){
        visitor.visit(this); }
} // 訪問を受け入れ、visitをthis(そのオブ
  // ジェクト自身)を引数として呼び出す。
interface Visitor {
    void visit(MyClass target);
}
class MyVisitor1 implements Visitor{
    void visit(MyClass t){
        System.out.println( t.getLName()+
    "さん、こんにちは。"); } }
```

```

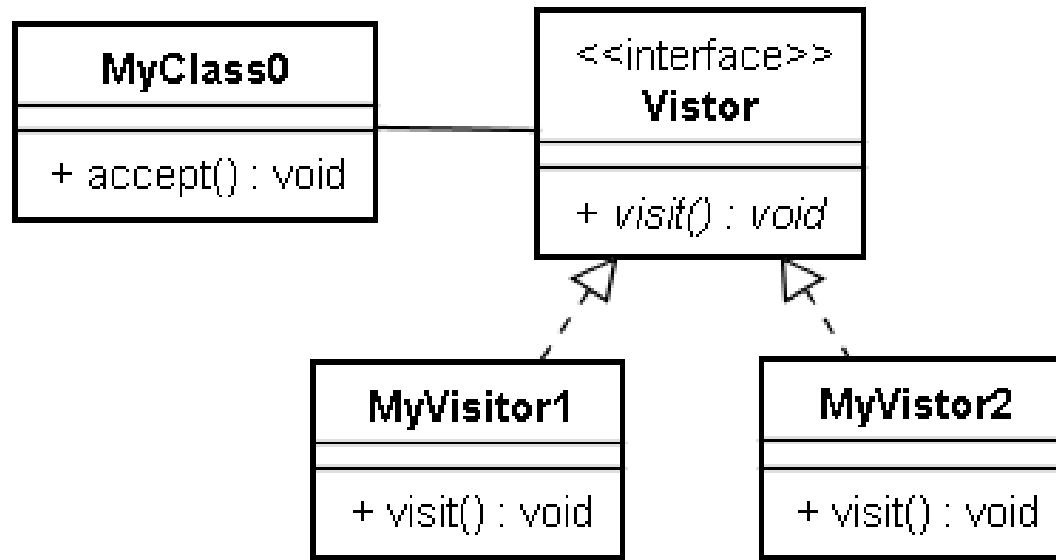
class MyVisitor2
implements Visitor{
    void visit(MyClass t){
        System.out.println("Hey, "+t.getFName()+" !");}
// main
MyClass m=new MyClass("Ichiro",
"Suzuki");
m.accept(new MyVisitor1());
m.accept(new MyVisitor2());
```

Suzuki さん、こんにちは。 (日本式)
 Hi, Ichiro ! (アメリカ式)

訪問を受けるクラスのaccept()を呼ぶと、visitorクラスのvisitメソッドが内部で呼ばれる。Visitor クラスの変更によって動作が変更可能。

visitorパターンのクラス図

- Acceptメソッドが呼ばれると、訪問者 visitorオブジェクトのvisitメソッドをそのオブジェクト自身(this)を引数として呼び出す



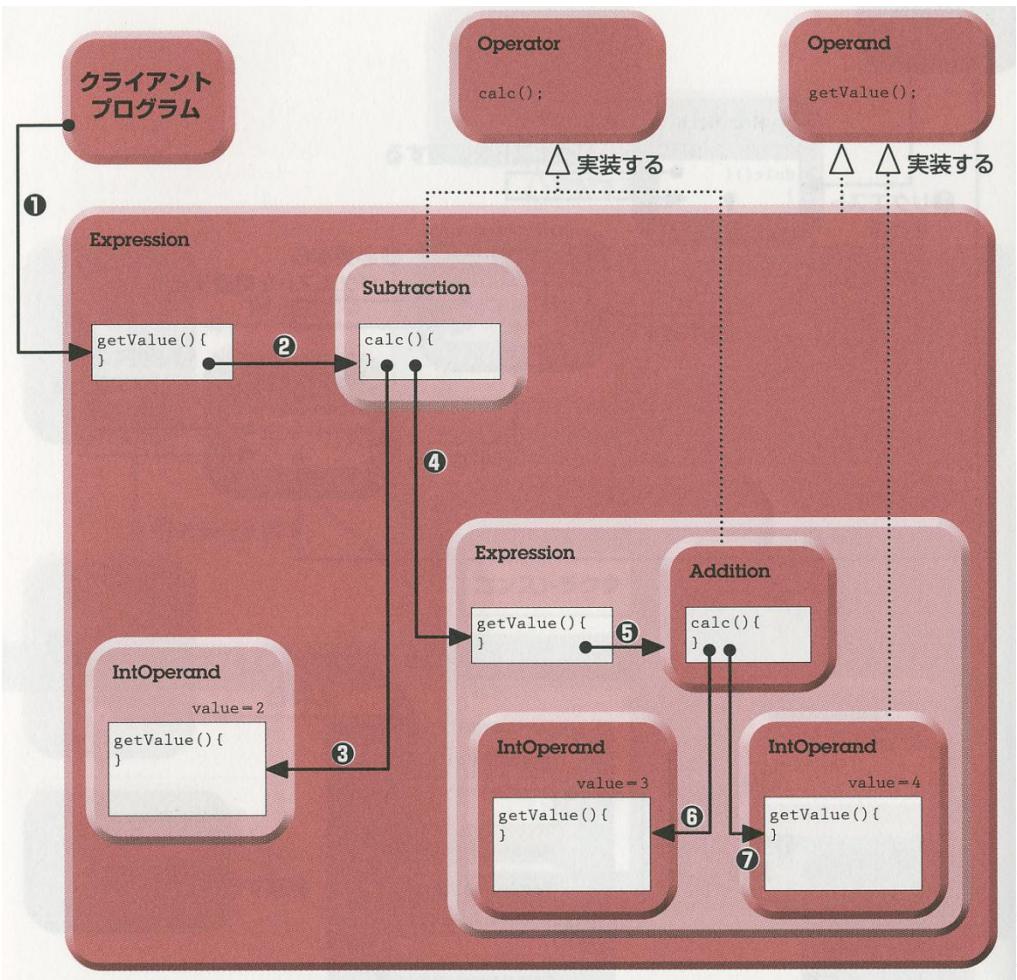
Interpreter パターン

<文法規則をクラスで表現>

- 言語の文法表現と、それを利用して文を解釈するインタプリタを定義する

構文解析結果を保持する
オブジェクトに実行機能を
与えて、処理を行う

構文解析用の
特殊なパターン



Interpreterパターン(2): 数式処理

```
// op1 + op2 のように2つの値を
// セットし、演算するインターフェース
interface Operator{ // 演算子
    void setOperand1(Operand op);
    void setOperand2(Operand op);
    int calc();
}

interface Operand{ // 被演算子
    int getValue();
}

// 整数クラス
class IntOperand
    implements Operand{
    private int val;
    IntOperand(int v){ val=v; }
    public int getValue(){ return val; }
}

class Exp implement Operator{
```

```
    Operand ope1,ope2;
    Operator opera;
    Exp(Operand o1,Operator o,Operand o2){
        ope1=o1; ope2=o2; operator=o;
        o.setOperand1(o1); o.setOperand(o2);}
    public int getValue(){ return opera.calc();}
}

// + のクラス
class Add implements Operator {
    private Operand op1,op2;
    public void setOperand1(Operand o)
        {op1=o;}
    public void setOperand2(Operand o)
        {op2=o;}
    public int calc(){
        return op1.getValue()+op2.getValue();}
}
```

Interpreterパターン(3): 数式処理

// - のクラス

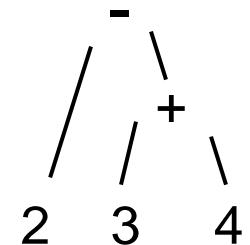
```
class Sub implements Operator {
    private Operand op1,op2;
    public void setOperand1(Operand o)
    {op1=o;}
    public void setOperand2(Operand o)
    {op2=o;}
    public int calc()
    return op1.getValue()-op2.getValue();
}
```

// main

```
IntOperand op1=new IntOperand(2);
Sub sub=new Sub();
IntOperand op2=new IntOperand(3);
Add add=new Add();
IntOperand op3=new IntOperand(4);
Exp subexp=new Exp(op2,add,op3);
Exp exp=new Exp(op1,sub,subexp);

System.out.println(exp.getValue());
// 2-(3+4) の演算
```

getValue()の再帰的呼び出しをオブジェクトで実現する.
再帰による数式処理のオブジェクト指向版.



数式処理だけでなく、正規表現やスクリプト言語の処理も実現可能。

デザインパターンのまとめ

- 品質のよいソフトウェアが効率的に設計できるようになる
 - 経験者の「経験」を利用する
 - 保守性が向上.
 - 変更容易性: 機能追加が容易になる.
 - 理解容易性: 「○○パターンを利用」ということがわかれれば、プログラムが理解しやすくなる.
- ただし、多少コードが長くなったり、実行時の効率が悪くなったりする.
- 1回しか実行しないような簡単なプログラムの開発時には不要.
- 「パターン」にこだわりすぎるのは、逆効果.
- “自然に”用いることが重要 ⇒ やはり経験が必要.
- 一通り知っておけば、役立つ(はず)！
- あとはリンク集 <http://mm.cs.uec.ac.jp/soft/link.html> のリンク先で勉強しましょう.

