アーキテクチャパターン

ソフトアーキテクチャ本

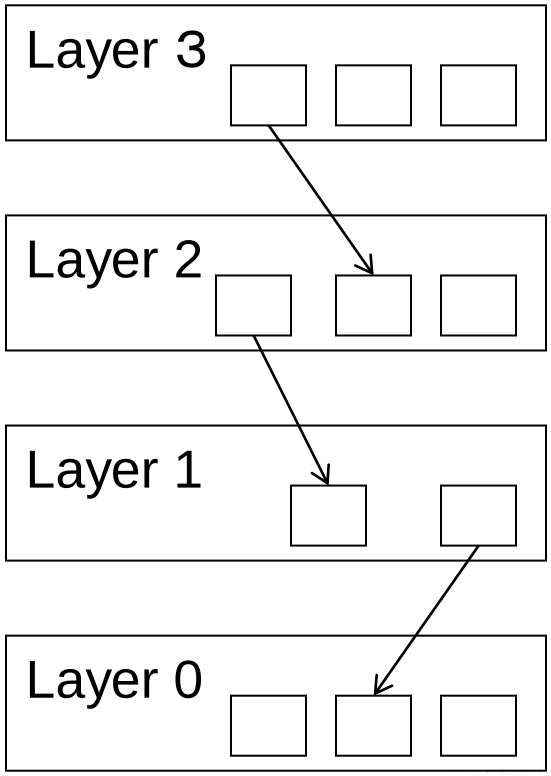
- F.Buschmann, H.Rohnert, P.Sommerlad, R. Meunier and M.Stal: Pattern-Oriented Software Architecture, Wiley, 1996 (邦訳：近代科学社, 2000)
  - アークテクチャパターンに加えて デザインパターン，さらに さらに粒度の小さいイディオム の概念を提案
  - アーキテクチャパターンは8種，デザインパターンも8種類を 提案
  - POSA本　と呼ばれることもある.

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

Layers

- 大規模システムを分割して開発する一手段.
- Layer(層)の積み重ねでシステムを実装.
- 異なるLayerの独立性は高い
- 各層は複数のクラス(モジュール)から構成される
- Layer n の機能は,下位のLayer n-1を利用して実現.
  上位が抽象度が高い.
  層の飛び越えはしない.
- OSIモデルが有名.



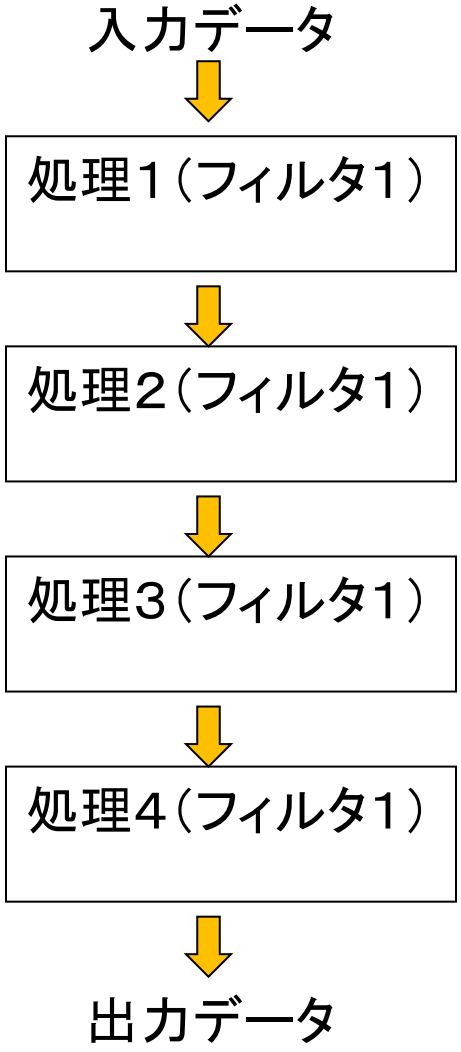
Layers

- 利点
  - レイヤーの再利用.
  - 依存性の局所化. 交換容易性.
- 欠点
  - 上位レイヤーの仕様を大幅に変更すると,
    すべての下位のレイヤーも変更しないといけない
    場合がある.
  - 効率性が高くない.
    レイヤー間呼び出しのオーバーヘッド.
  - レイヤーの粒度の決定が難しい.

Pipes and Filters

- データストリーム処理
- ほぼ完全に独立したシステムを直列に組み合わせて，入力データを出力データに変換
- UNIXのパイプ

  例） cut –d”,“ –f2 seiseki.txt |sort –n | uniq -c
- コンパイラの例
  - 字句解析→構文解析→意味解析→中間コード→最適化→CPU別コード生成



Pipes&filtersの例：画像認識システム

画像入力

前処理（リサイズ，平滑化，明るさ調整）

セグメンテーション（直線抽出，領域分割）

特徴抽出，特徴ベクトルの生成

識別器の訓練（学習ステップ）
or
未知画像の認識・分類（認識ステップ）

認識結果

画像認識特有の処理

一般的な機械学習手法

320x240に統一

領域に分割

5つの領域から色，模様，形状特徴を抽出し，特徴ベクトル v を生成

特徴ベクトルに対して最近傍法，SVMなどの学習法で　学習・分類

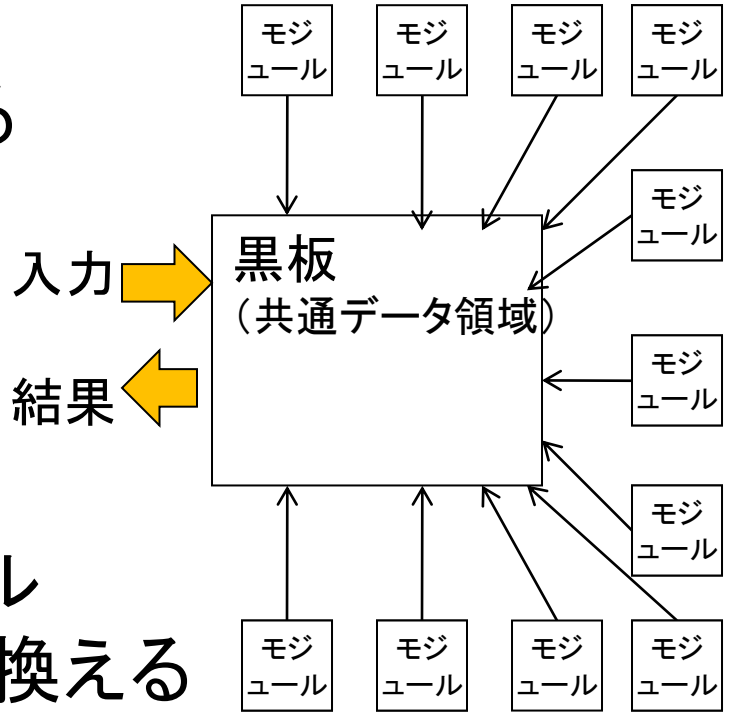
未知画像に対するトラ確率[0,1]出力

Pipes and Filters

- 利点
  - フィルタ交換が容易．組合わせの変更が容易
  - 既存フィルタの組合わせにパイプラインのラピッドプロトタイピング．
  - 独立した処理は，並列化可能．
  - 中間ファイルが不要．
- 欠点
  - 標準的データ形式への変換のオーバーヘッド
  - 個々のフィルターが独立しているので，統一したエラー処理が難しい

Blackboard

- 処理手順が明確でない，
  もしくはデータによって変化する
  場合に利用．
- まず，黒板と呼ばれる共通
  データ領域（共有メモリ）に
  データを書き込む．
  次に，データに応じたモジュール
  が次々に起動し，データを書き換える
- 元々，音声認識システム（人工知能システム）で，
  用いられた．モジュールは「知識源」と呼ばれる

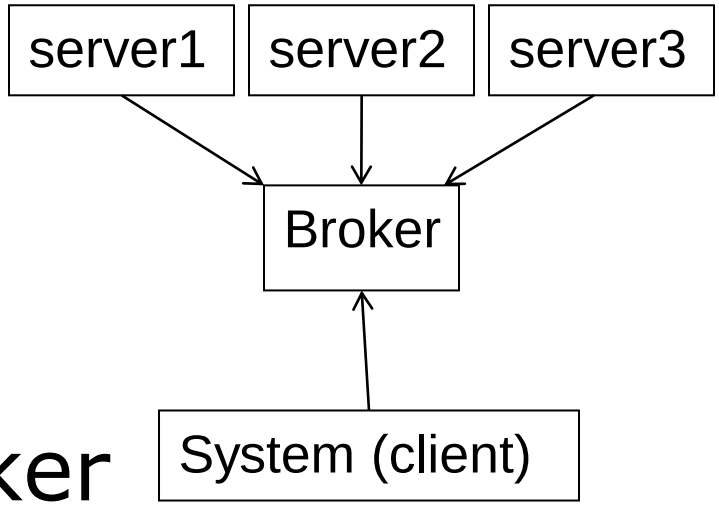


Blackboard

- 利点
  - 多数の処理モジュールを用意し，データに応じて動的に処理の順番（パイプライン）が構成される
  - モジュール間は完全独立．黒板を通じてのみ相互作用をする．
- 欠点
  - 動作の予測が不可能．デバッグしにくい．
  - 解の妥当性が保証されない．
  - 効率性が低い

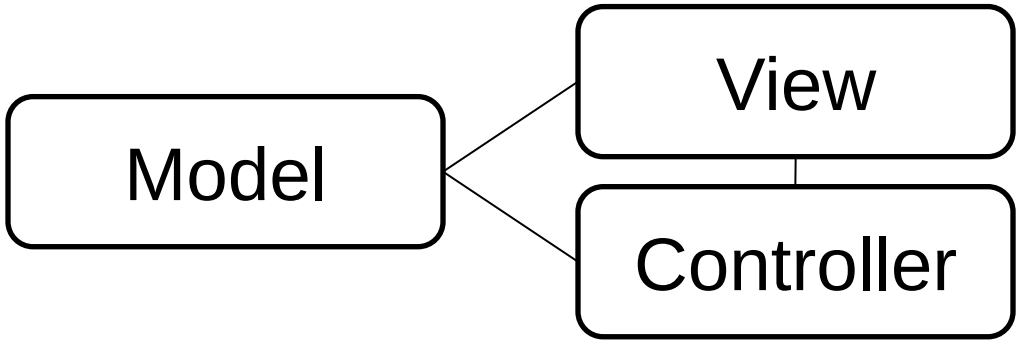
Broker

- 分散システムのためのパターン.
- Brokerがクライアントからの呼び出しを仲介する.
- サーバはbrokerに登録.
- クライアントは必要な機能をブローカーに与えると，broker経由で適切なサーバが呼び出される.
- 利点：柔軟性．拡張性.
- 欠点：間接レイヤ(broker)によるオーバヘッド.



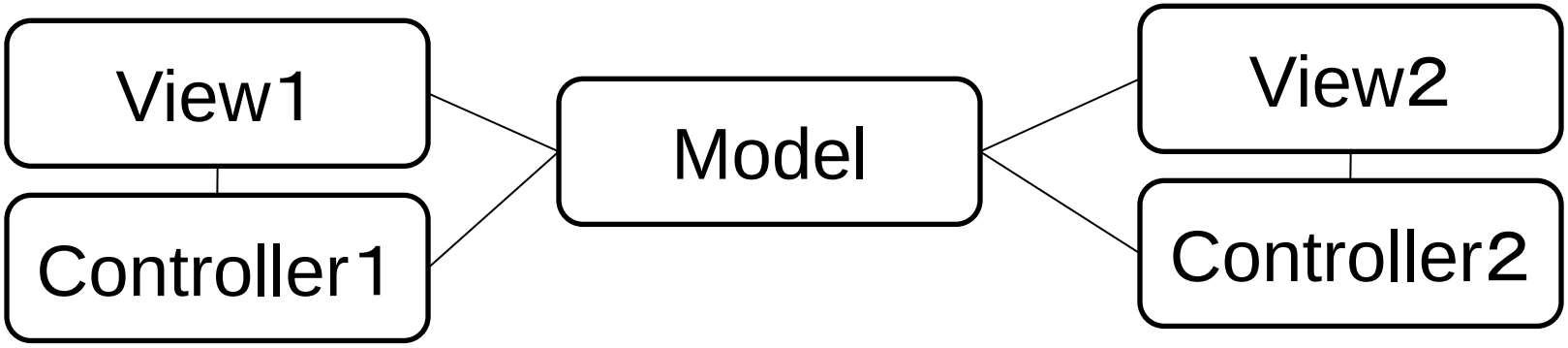
Model-View-Controller (MVC)

- 対話型システムの設計方法.
- Model, View, Controllerの3つの要素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
    - 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

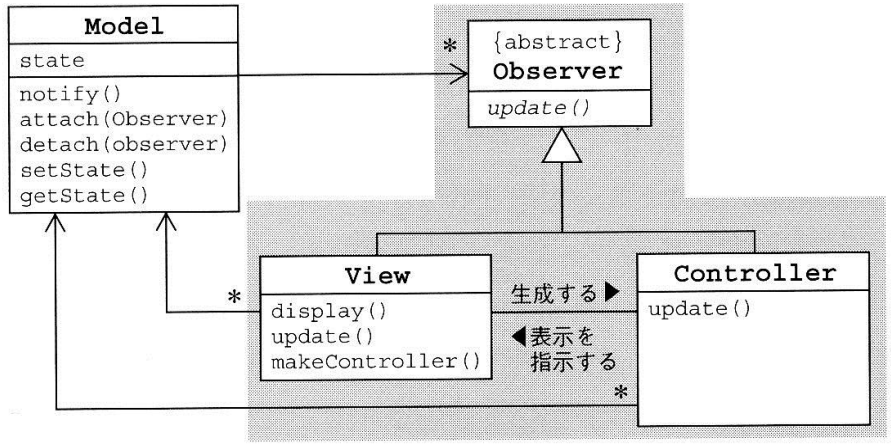
MVCモデル と observerパターン

- **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で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.

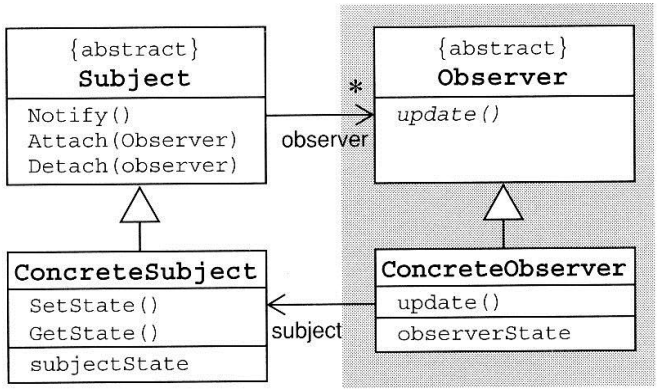
図 C-1 MVC モデルと Observer パターン

(a)MVCモデル



- **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.

(b)Observerパターン



MVCモデル と observerパターン

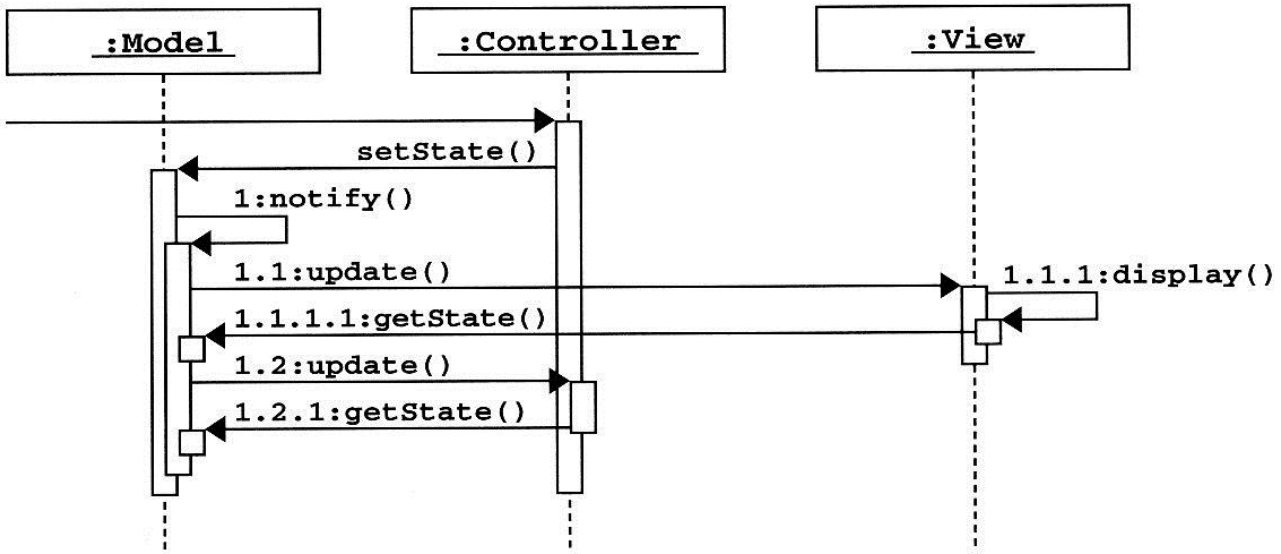
Model: データ構造の中心 （基本データ構造 と その操作）

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



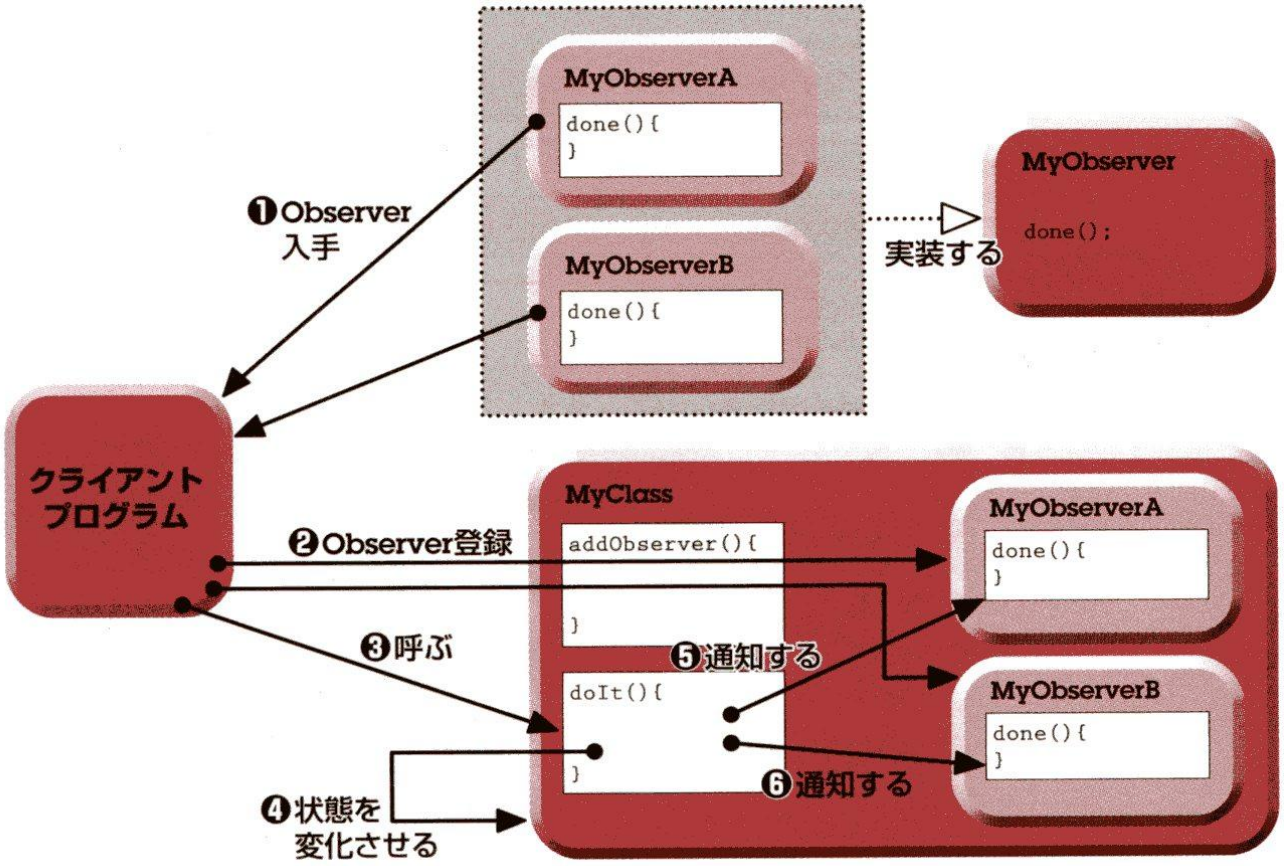
MVC

- 利点
  - インタフェースとモデル部分の分離
  - インタフェースの交換が容易

- 欠点
  - 複雑度の増大
  - 更新が過度に起こる可能性
  - ビューとコントローラの依存性が大きくなりやすい
  - VCとモデルの直接の結合
    - 次のPACでは，それを解決する方法を提案

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)

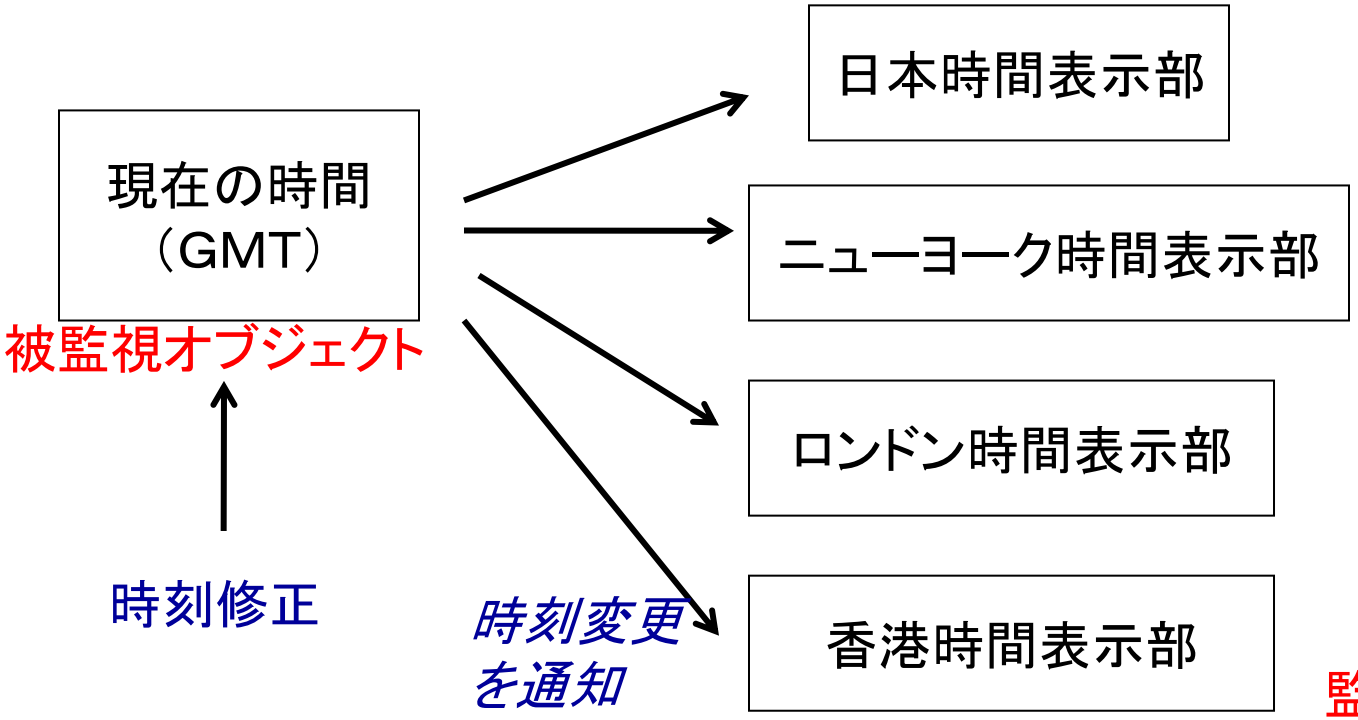
```
// 監視する(状態変化を通知
// される)オブジェクトのクラス
interface Observer {
   void update(int u);
} // doneは更新時に呼ばれ
//   るメソッド
class ObserverA
      implements Observer{
  public void update(int u){
     System.out.println(
       "A: updated : "+u); }
}
class ObserverB
      implements Observer{
  public void update(int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

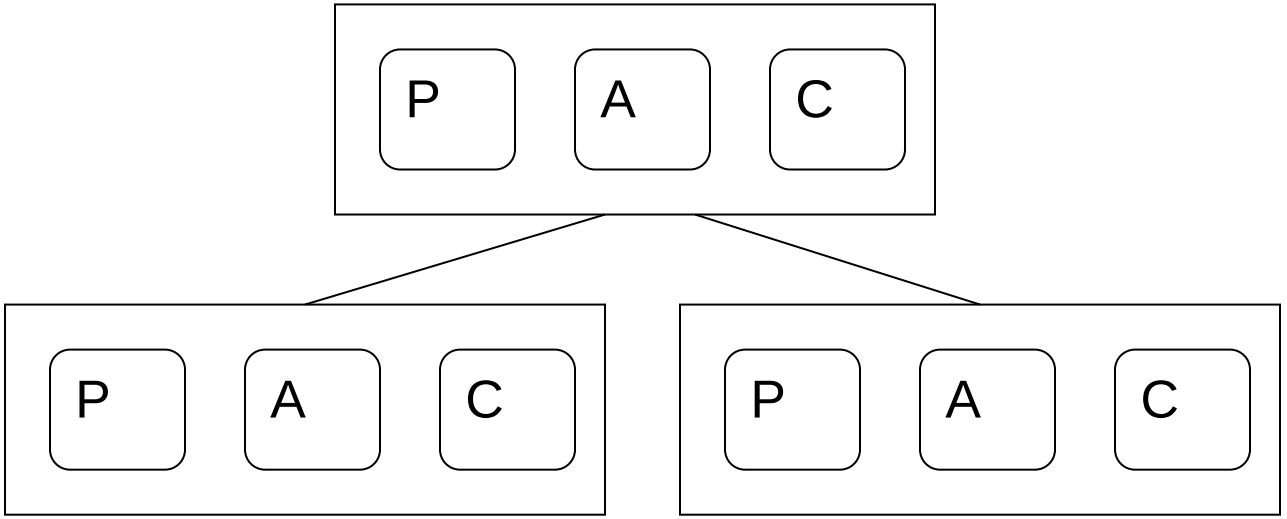
- Javaの標準ライブラリ　Java.utilに Java.util.Observerインターフェース　と Java.util.Observableクラスがある.
  - 考え方は同じ. 更新を通知する方が, Obserableを継承する.
  - ただし, Observableがクラスであるので, 更新を通知するクラスは他のクラスを継承できない. (欠点)(継承は1つのクラスのみなので)
- Observerパターンは, アーキテクチャパターンのMVCモデルと関係が深い

Presentation-Abstraction-Control (PAC)

- 協調するエージェントの階層として，
  対話型システムを構築．各エージェントは，
  Presentation, Abstraction, contorol
  の3種類のコンポーネントから構成される．
- Presentation:　ユーザとの対話部
- Abstraction: データを保持
- Control: プログラムの実行を制御
- P,A,Cからなるエージェントの階層として，
  システムを実現．

Presentation-Abstraction-Control (PAC)

- Pがboundary, Aがentity, CがControlクラスに相当.
- MVCと似ているが，AとCが分離している.



その他

- Microkernel　（OSの実装法の一つ）
  - 必要最小限な機能をmicrokernelに実装し，それ以外の機能は外部サーバに実装する.
- Reflection　（自己反映システム）
  - システム自身の構造と振る舞いを動的に変更を可能とする機構をシステムに持たせる.
  - メタレベルという特別なレベル（レイヤ）を用意し，そこにそのシステム自身の構造と振る舞いの知識を記述.（メタオブジェクト）
  - メタレベルの変更で，システムが動的に変わる.

アーキテクチャパターンのまとめ

- 大(中)規模システム設計において有用な基本的なアーキテクチャのパターンを体系化
- プログラミング言語に依存しない
- どれも基本的なシステム構成方法
- いろいろな分野のシステムのアークテクチャから取ってきている.
  - OS, AIシステム, GUIシステムなど.
- ただし, これらのパターンはシステムアーキテクチャのあくまでも一例に過ぎない.
  - 他にもアーキテクチャはいろいろある.