オブジェクト指向開発論

2020年7月16日

海谷 治彦

目次

- デザイン(設計)とは，パターンとは
- パターンで使われるオブジェクト指向の表現法
  - 継承，インタフェース，抽象クラス，コンストラクタ，static メソッド/属性
- 設計の柔軟性を担保するオブジェクト指向の考え方.
- パターンの起源，メリット，表記法
- パターンの例
  - Singleton
  - Factory Method
  - Abstract Factory
  - Template Method
  - Iterator

ソフトウェアのデザインとは？

- デザイン Design 設計
- プログラムの構造，構成のことをさす.

- C言語等，手続き型の場合:
  - 関数とその呼び出し関係.
  - データフロー図等がよく使われた.
  - ソースから逆算するなら，コールグラフ.
- オブジェクト指向の場合:
  - クラス図 (クラスとクラス間の関連)

何故，デザイン(設計)するか？

- 機能する物を効率的に作るため．
  - 学校の演習で作るような小さなものは設計しなくても，プログラムはできてしまう．
  - しかし，そこそこの規模がある物は，設計をちゃんとしないと，そもそも完成しない．
  - 何の設計もせずに橋や家を作ることを考えてみてほしい．
- 開発後に機能や性能変更を容易に行なうため．
  - 家やビル等よりも頻繁に機能や性能の変更を迫られる．

- 良い設計 ⇒ 変更が容易な設計，柔軟な設計

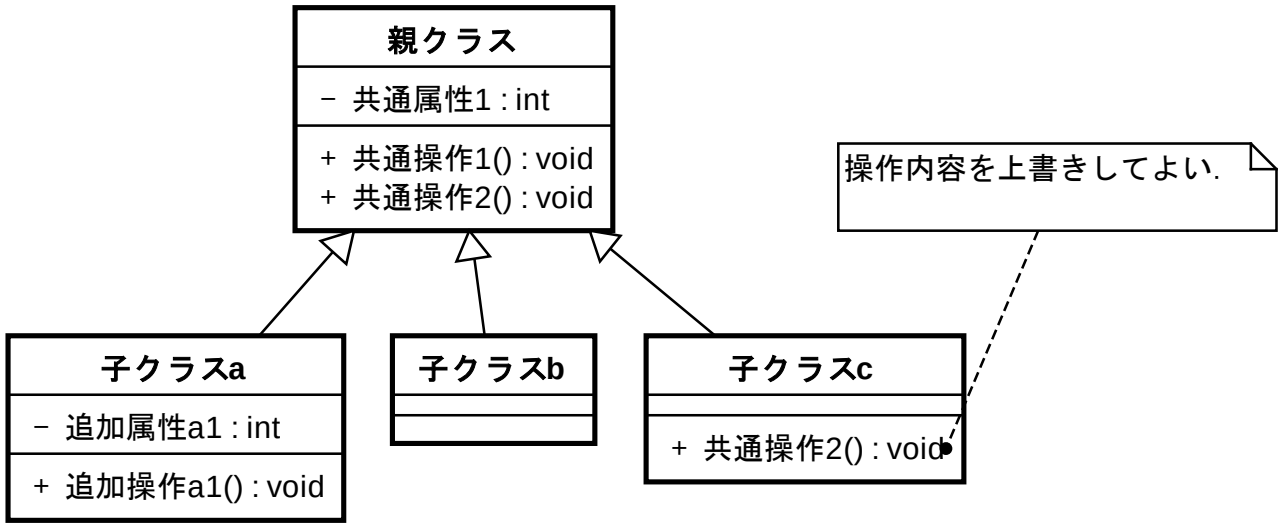
復習 オブジェクト指向の表記法

- 以下の表記法や概念が，デザインパターンをかたち作るキーとなっている.

- 継承
- インタフェース
- 抽象クラス
- コンストラクタ
- static メソッド, static 属性

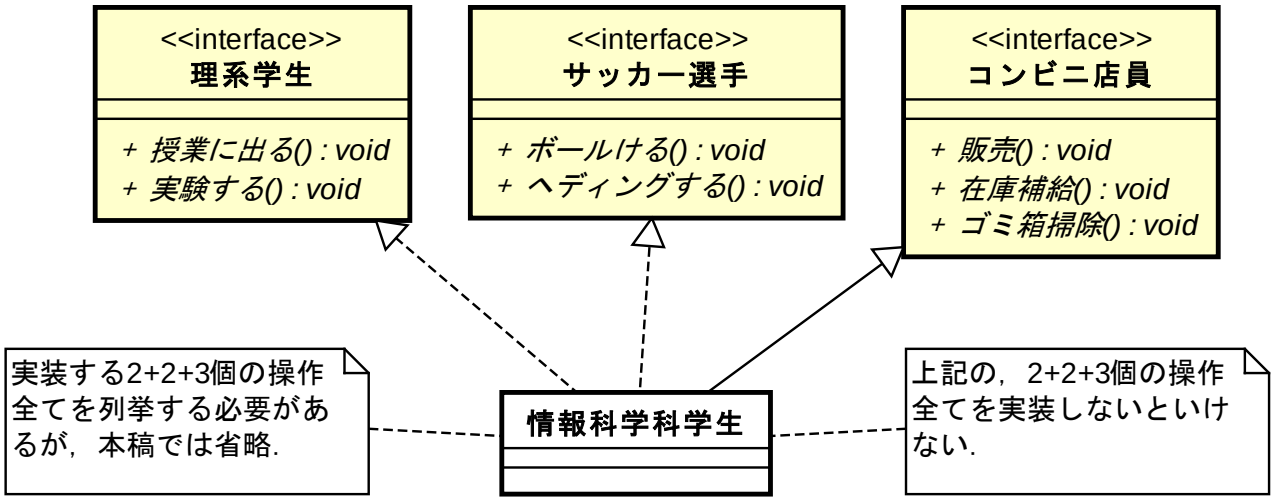
継承

- あるクラスを拡張して，属性や操作を追加したり，操作内容を変更したりすることができる.
- 元になるクラスを親クラス，拡張されたクラスを子クラスと呼ぶ場合もある.
- Javaでは，子から見た親は1個に限定される.



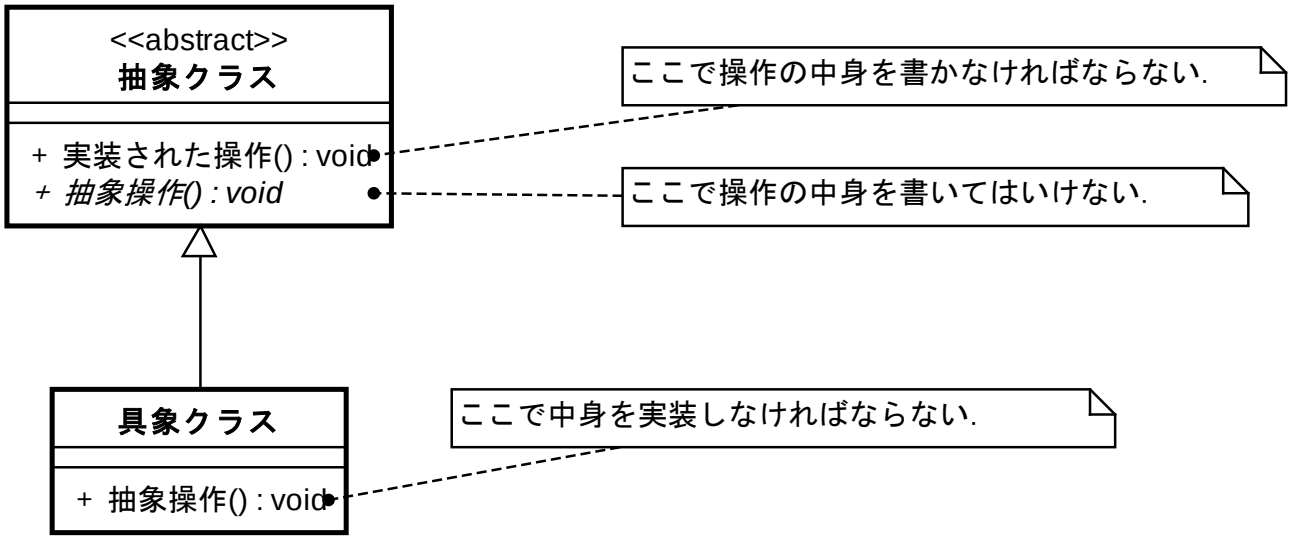
インタフェース

- クラスが持つべき操作の集合を規定できる.
- あるクラスはあるインタフェースを実装する，という表現する.
- 役割を規定しているとも意味的には考えられる.
- 一つのクラスは複数のインタフェースを実装してよい.



抽象クラス

- 一部メソッドのみインタフェースのように，実現法を子クラスに委ねるクラス.
- 継承の一種なので，Javaで開発する場合は，親クラスは1個しかとれない.



コンストラクタ

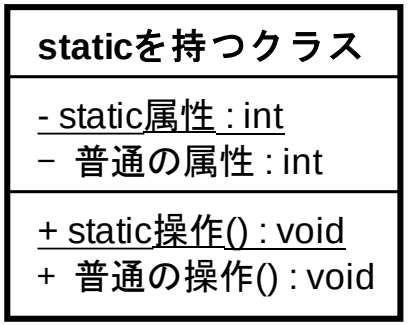
- クラスのインスタンスを生成する時に呼ぶメソッド.
- Javaの場合は,

new クラス名(引数)

- という表現となる.
- 通常は，publicアクセスだが，privateやprotectedにして，インスタンスの無制限な生成を制限できる.

static method, static 属性

- クラスに共通した操作や属性である.
- staticメソッドは，インスタンスを生成しなくても呼び出せる.
- static属性も，インスタンス生成をしなくても扱える.
- 加えて，static属性は，同じクラスのインスタンス間では，共通データとして扱うことができる.



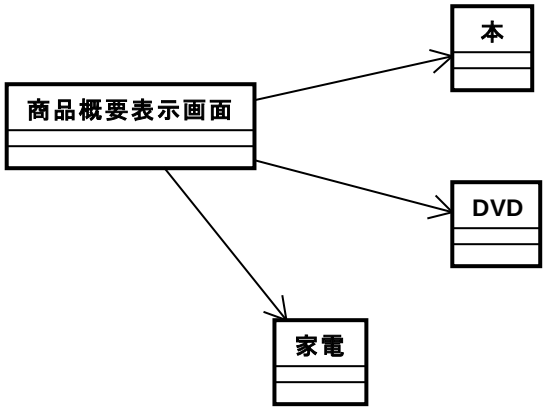

```
public class staticを持つクラス {

  private static int static属性;
  private int 普通の属性;
  public static void static操作() {/* なんか書いて */}
  public void 普通の操作() {/* なんか書いて  */}
}
```

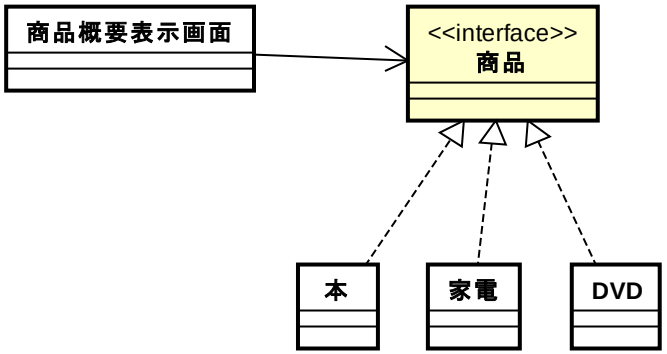
柔軟な設計の基本方針

- 具体的なクラス同士を結び付けない.
  - なるだけインタフェースを使う.
  - 委譲を使う.
- 使いまわし可能なクラス群(ライブラリやフレームワーク)を使う場合，使う側(クライアントと言う)で，使われる側のクラス等の名前を記述する箇所を最小化する.

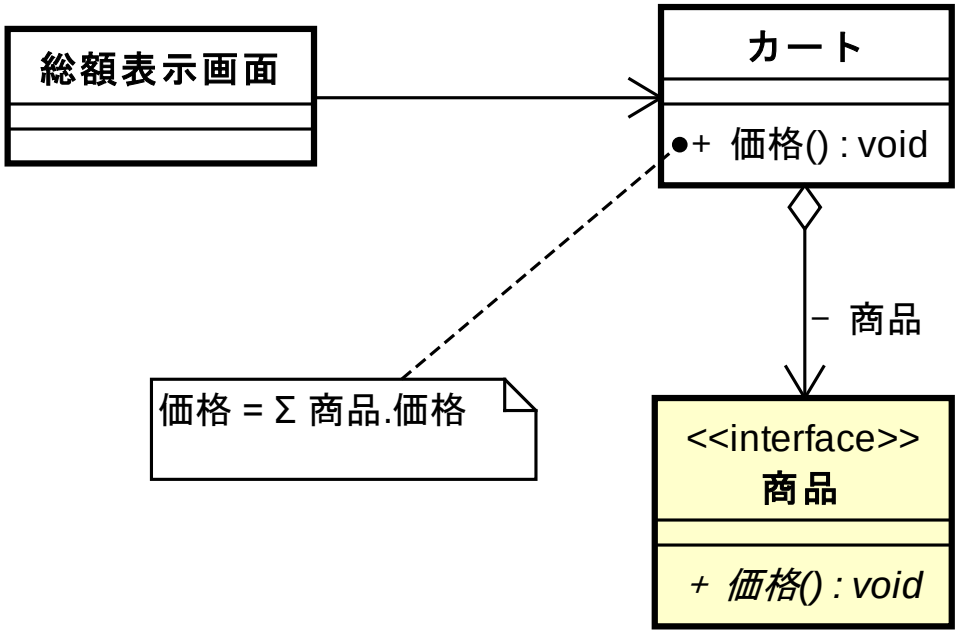
インタフェースの例



よりも



委譲の例



デザインパターンとは？

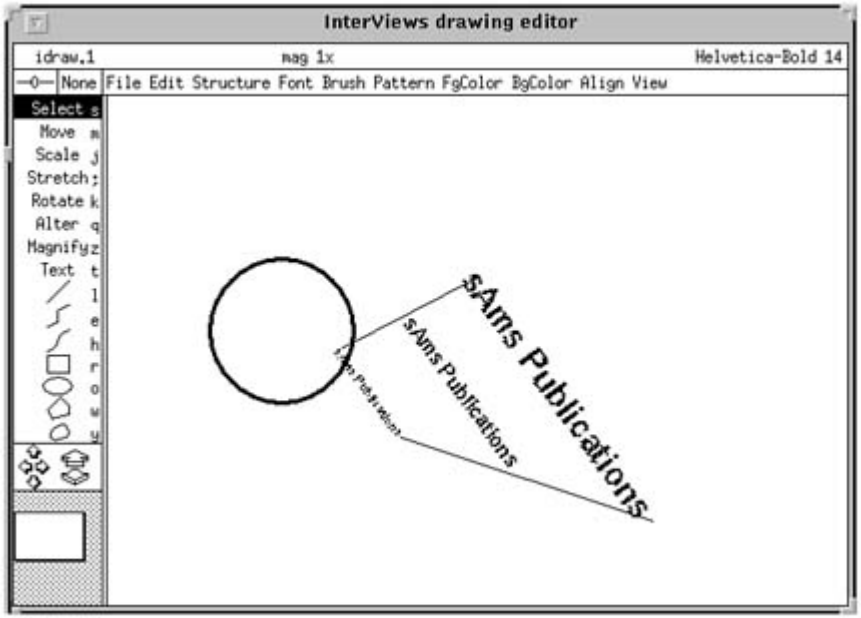
- クラスおよびそれらの関係の典型的な書き方.
- システム全体の設計では無く，断片的な部分の書き方である.
- その書き方に従うと，柔軟な設計となる場合が多い.
- パターンによって，どういう観点から柔軟になるか異なる.
- 一つの設計に複数のパターンを使ってもよい.

デザインパターンの生い立ち

- 経験的に収集されたものであり，理論的に構築されたわけではない．
- ほとんどのパターンには名前がつけられており，熟練したソフトウェア開発者は，主な物は，その名前だけでわかる．

デザインパターン起源の例

- Unidraw (InterViews) というお絵かきツールのためのフレームワークがあった(ある).
- 30年前からあるツールだが，文字や図形の変形や回転等の汎用性が抜群であった.
  - 特に，拡大縮小で，図形と文字を区別しない作りがすきだった.



- 少なくとも，10個のパターンが本フレームワークから生まれている.
- C++ で実装されている.

デザインパターンのメッリト

- 共通の語彙を提供する.
- メリット，デメリットが把握できる.
- 設計の考察を与えてくれる.
- 設計の目標を与えてくれる.
- 問題の解決策を与えてくれる.
- 適用できる範囲が広い場合が多い.
- 優れた設計を効率的に習得できる.

デザインパターンの表記法

- 伝統的にデザインパターンは，複数の項目を持つ表形式(フォーム形式)で書かれる場合が多い．

- 図やサンプルコードを含む場合が多いので，エクセルの表みたいに罫線がひいてあるわけじゃない．

代表的な表記項目

- パターン名と分類
- 目的
- 別名
- 動機: どんなときにこのパターンを使えばよいか？
- 適用可能性: どんなところに使えるか？
- 構造
- 構成要素
- 協調関係: 構成要素の責任分担を明示
- 結果: どんなメリットがあるか？
- 実装
- サンプルコード
- 使用例
- 関連するパターン

代表的なパターンの例

シングルトン

- Singleton

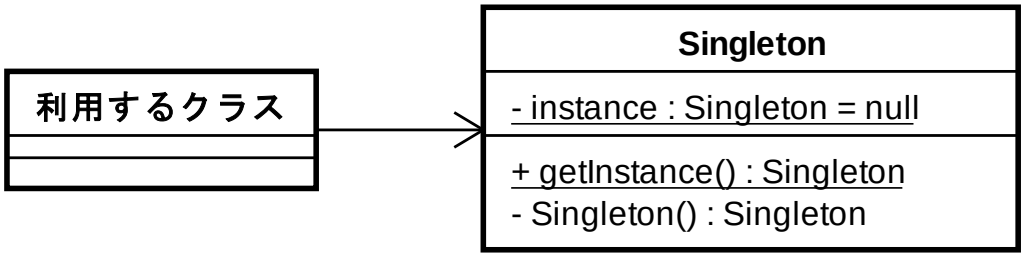
- クラスに対するインスタンスは文法的には，無制限に複数生成できる.
- しかし，実質的に一つのインスタンスしか体系(システム)中に存在しない場合も多い.
  - 会社には社長は一人しかいない.
  - コンピュータには一つ(程度)のCPUしかない.
  - それぞれのアプリからは，1つの標準入力(キーボード等)しか認識できない.

  等

構成，構造

- クラスのコンストラクタへのpublicアクセスの禁止.
  - 当該クラス中からのみのアクセスとする.
- 適当な static メソッド経由で，唯一のインスタンスへのアクセスを許す.
  - instance や getInstance 等の名前にする場合が多い.
- 実際のインスタンス生成は，最初のstaticメソッド呼び出しの際に行う.

クラス図とコードの例




```
public class Singleton {
	private static Singleton instance=null;

	private Singleton(){
		// do some initialize
	}

	public static Singleton getInstance(){
		if(instance==null) instance=new Singleton();
		return instance;
	}
}
```

結果: どんなメリットがある？

- インスタンスへのアクセスを制御できる.
- グローバル変数的なものを使わないで済む.
- インスタンスの数を計画的に設定，変更できる.

利用例

- 前述のUnidrawでも使われいる.
  - GUIの見た目切り替えに用いられている.
    - Win風のボタン等を，Mac風のボタン等に切り替えができる.
    - 実際は，Win, Macではなく，当時存在した，部品キット群だが.
- 結構，今でも頻繁に目にするデザイン.

ファクトリーメソッド

- Factory Method

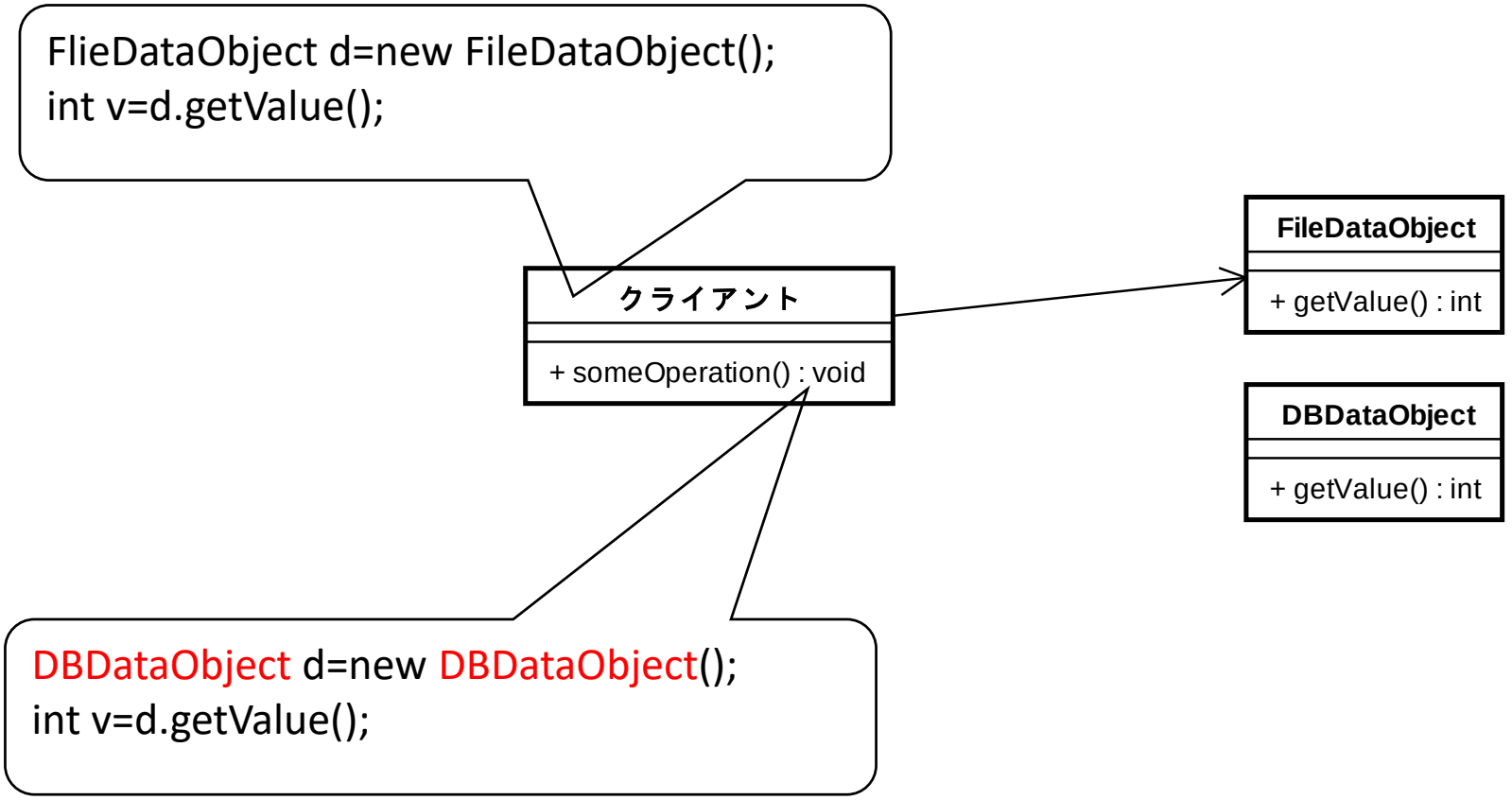
- オブジェクトを生成する時のインタフェースのみを規定して，実際にどのクラスをインスタンス化するかをサブクラスで決めるようにする.
- 結果として，生成されたオブジェクトと，生成を依頼した部分(クライアントと呼ぶ)との結合を緩めることができる.

最初の例題の説明

- 複数種類のデータを扱うクラスがあるとする.
- 例えば，ファイルから，DBから，Webからデータを読みだし，それを保持するクラス.
- クライアントは，将来，データを扱うクラスを，ちょいちょい変更する可能性があるとする.

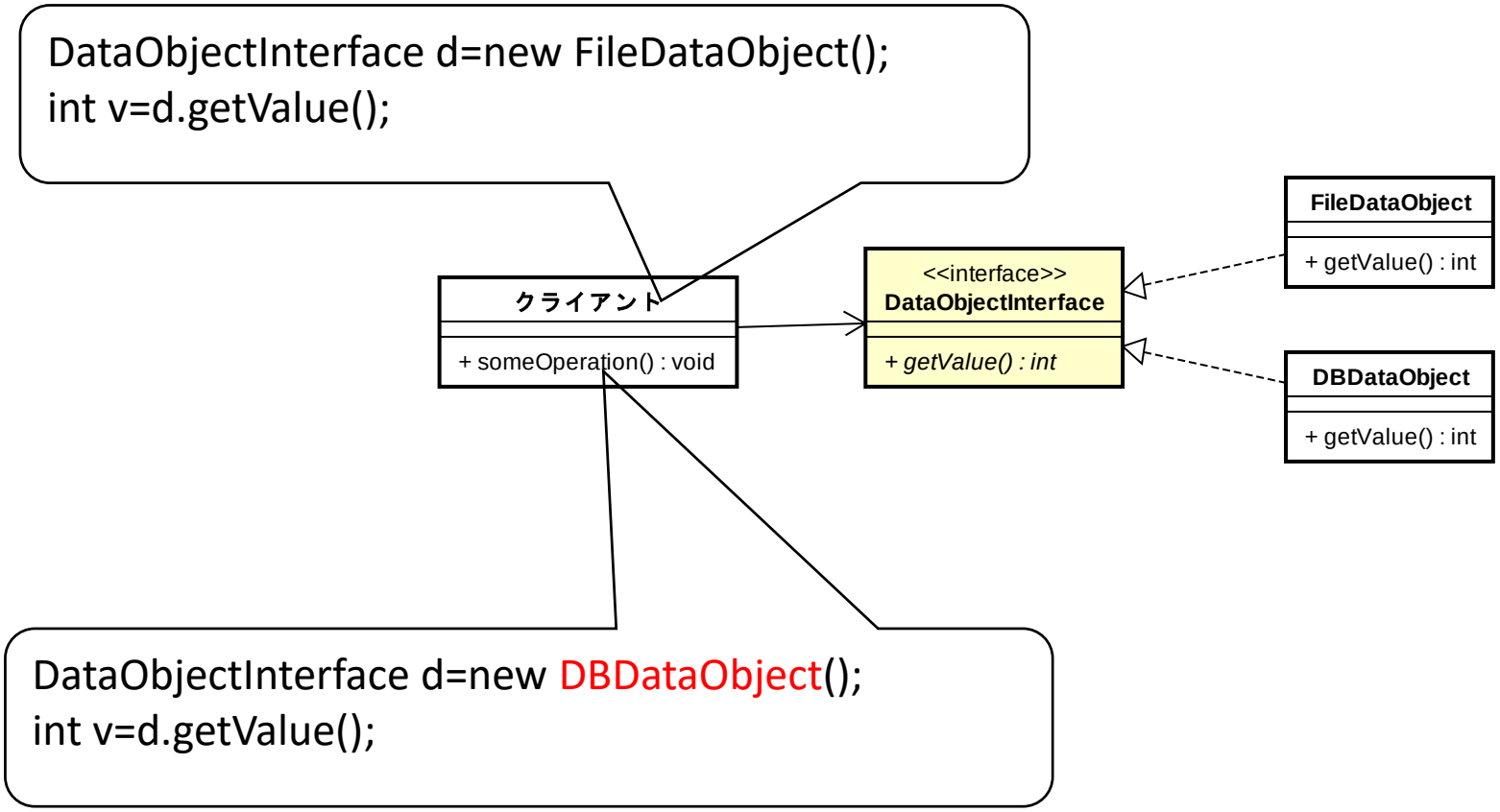
普通にクラスを使う

- データクラスを使う側において，赤い感じで直さないとだめ．



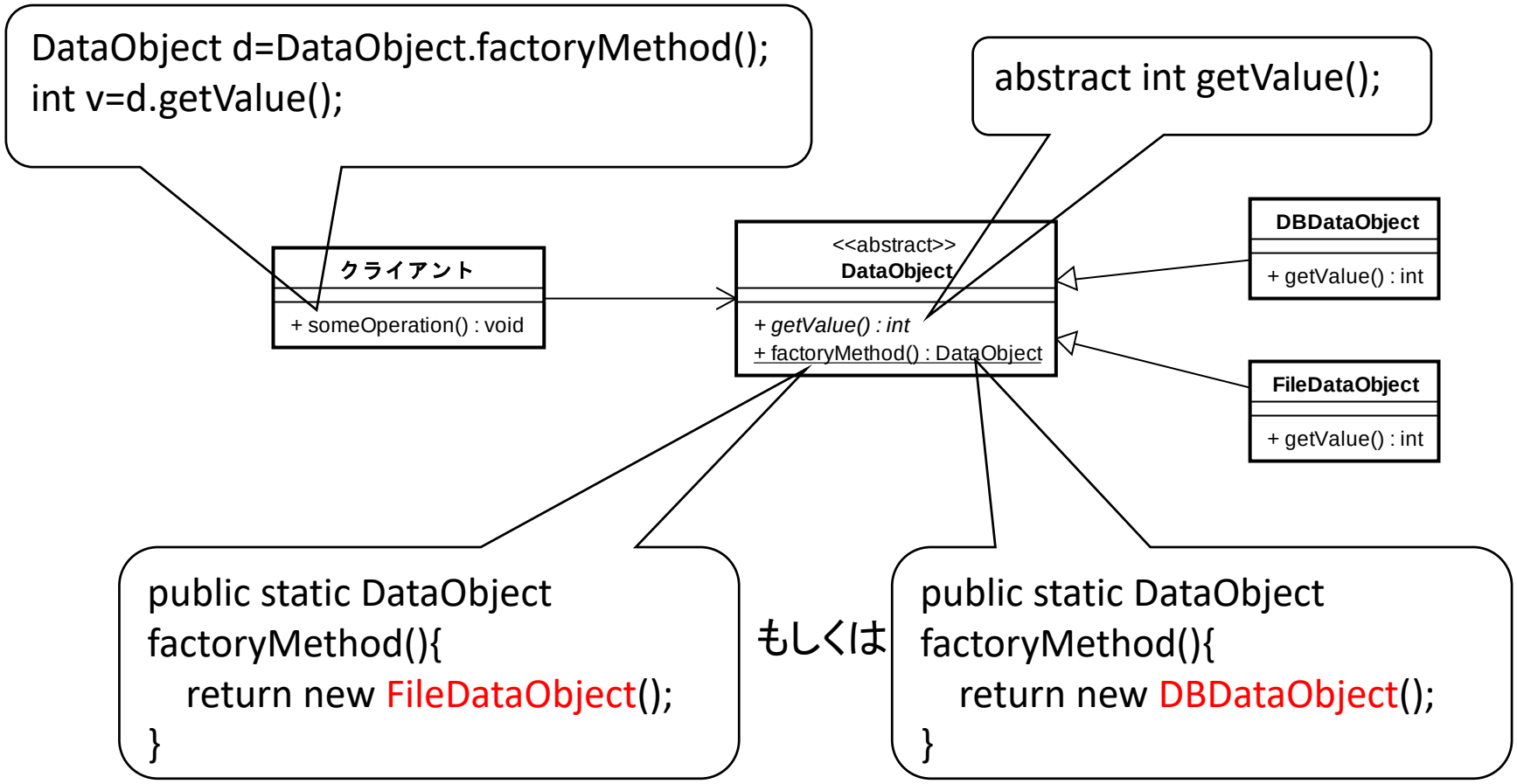
単純にInterfaceを使う

- データクラスを使う側において，赤い感じで直さないとだめ．



Factory Methodを利用

- 少なくとも，データクラスを利用する側(クライアント)には，全く具体的なクラスの名前は使われなくなる.



Factory methodの利点と特徴

- 複数の切り替え可能な選択肢があるようなクラス群を使うクライアント(自作プログラム)のほうを，あまり修正しなくて済む.
- 仕組みとしては，static method (クラス共通メソッド)を利用している.

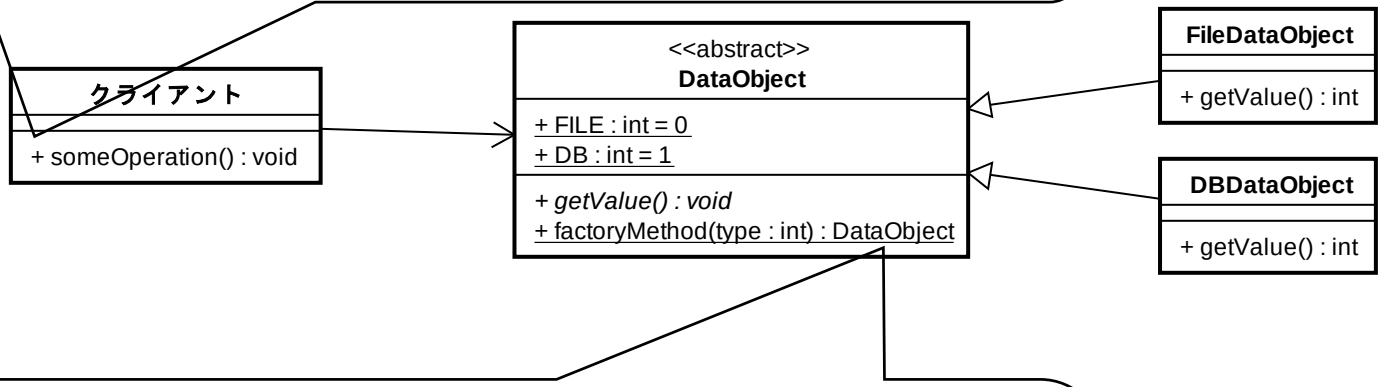
問題設定の更新1

- クライアント側でデータオブジェクトの種類を明示的に指定したい.

引数の利用

- これでは，new FileObject(), new DBObject() と書くのと変わらなくなってしまう.

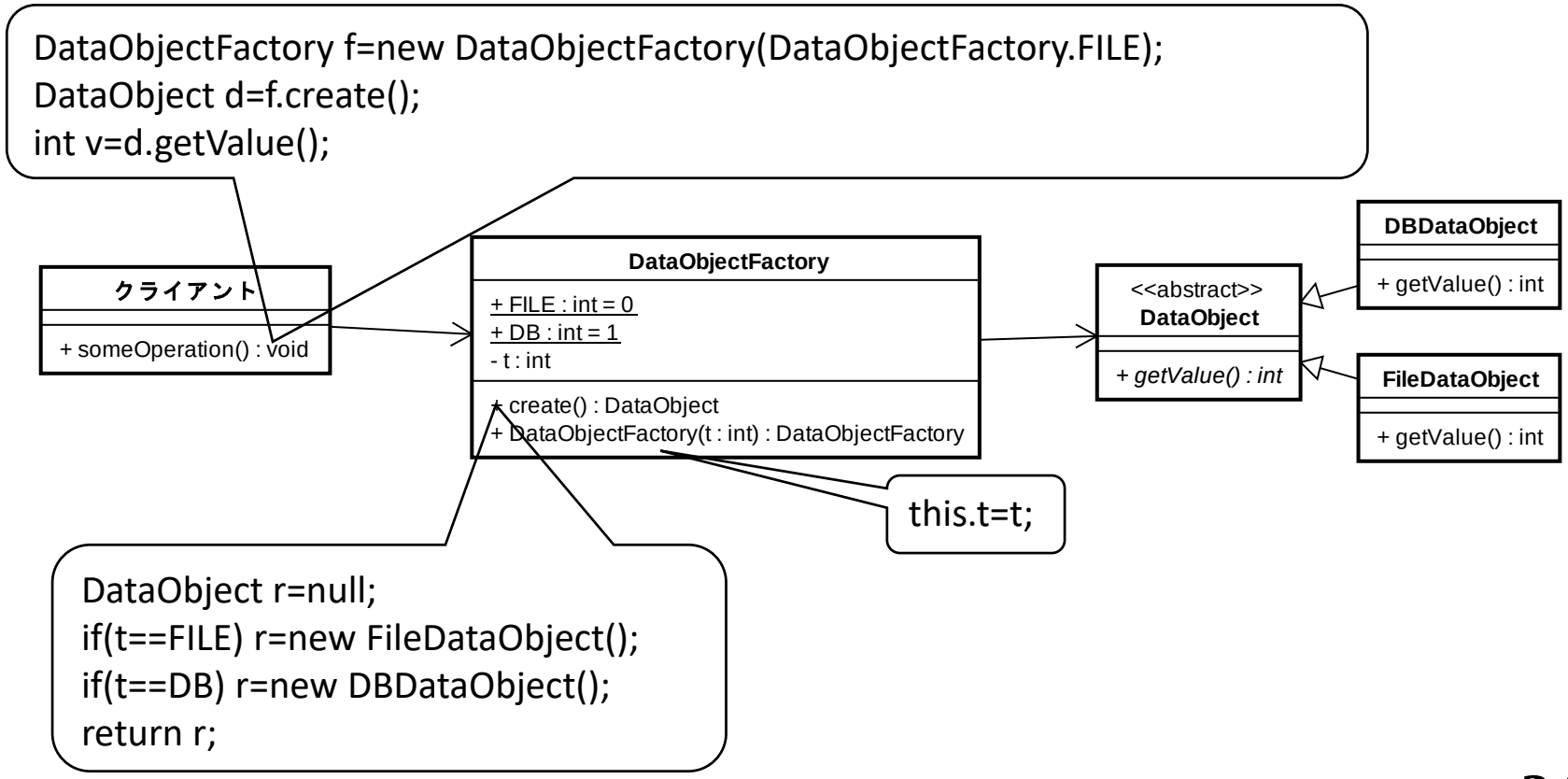
```
DataObject d=DataObject.factoryMethod(DataObject.FILE);
int v=d.getValue();
```




```
public static DataObject factoryMethod(int t){
DataObject r=null;
    if(t==FILE) r=new FileDataObject();
    else if (t==DB) r=new DBDataObject();
   return r;
}
```

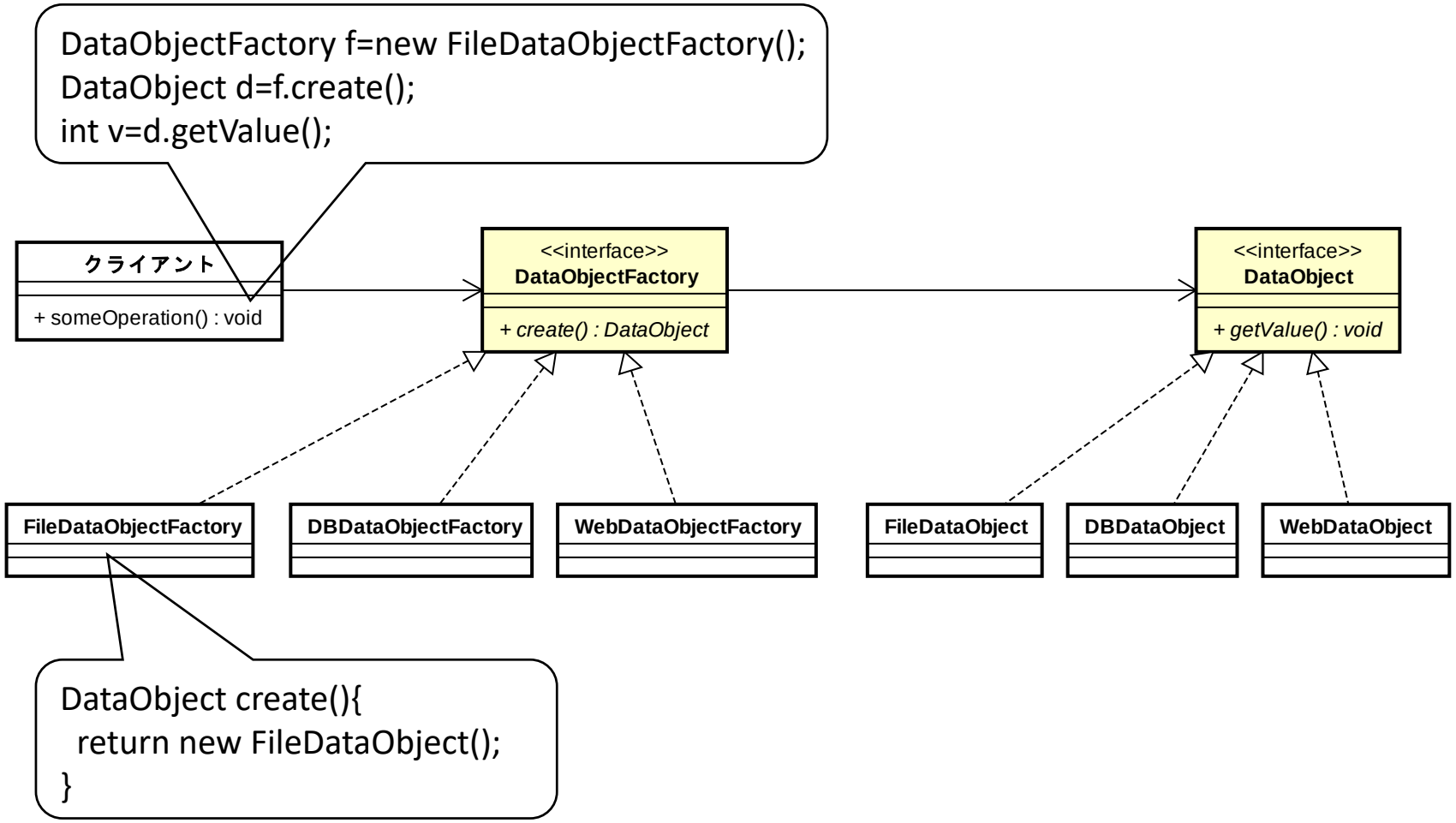
生成する責務をクラスとして独立

- 以下にすることによって，引数利用の問題は解決される.
- 繰り返し create する場合には有効.



インタフェースの利用

- 生成対象が増えた場合にも柔軟に対処可能.



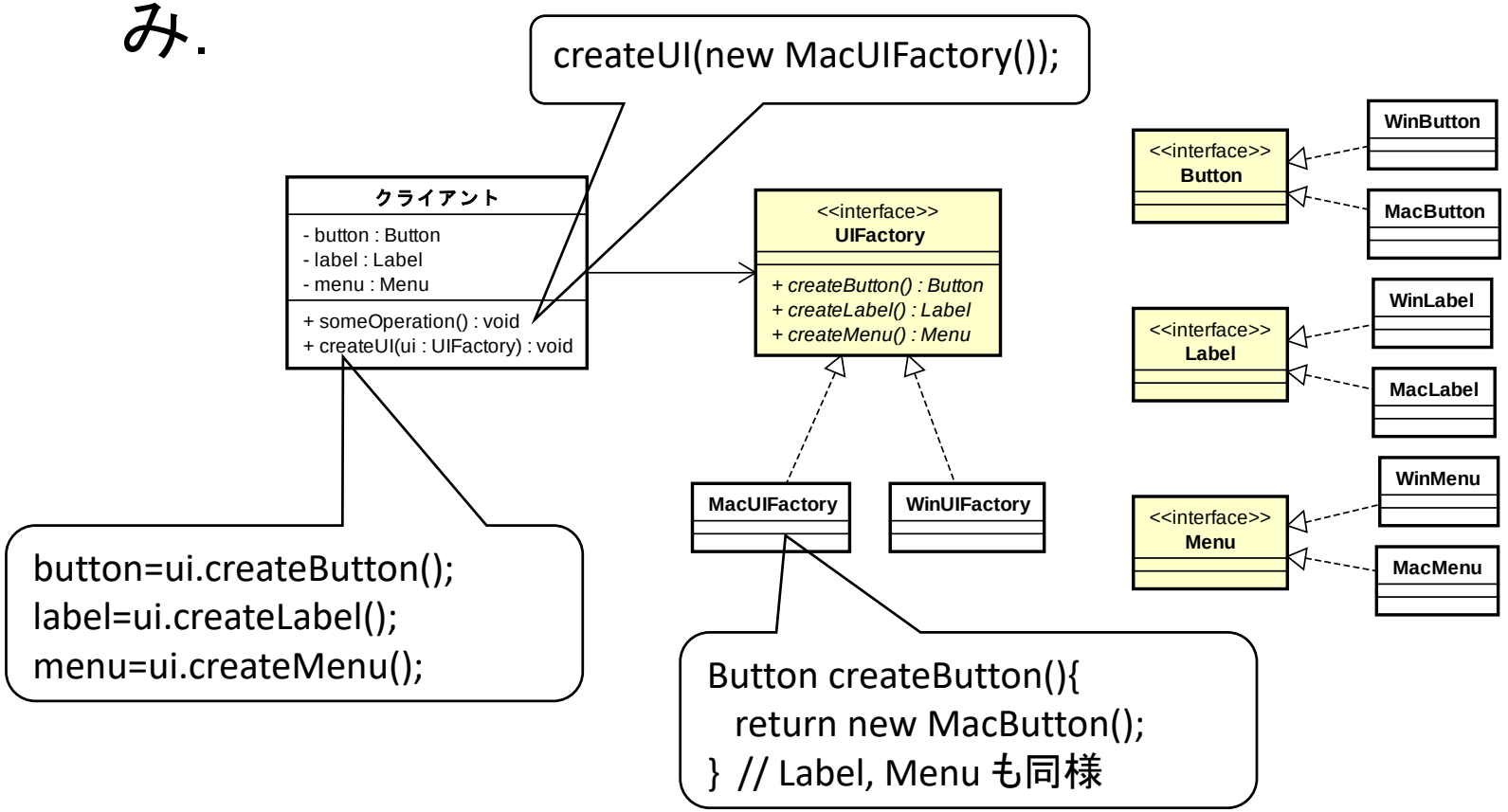
アブストラクトファクトリー

- Abstract Factory

- セットになっているオブジェクト群を一括して切り替えたりするのに便利にパターン.
- 次のページの例にもあるように，UI部品等でよく使われる.
- オブジェクトのセットを使う側(クライアント)のコードには，具体的なオブジェクトを示すクラスを指定しなくて済む．よって，クライアントと部品群との結合性が弱まってよい.

設計の例

- ユーザーインタフェース(UI)をMac風 or Win風で同時に切り替えるための構造.
- 単純化のため，UIは，ボタン，ラベル，メニューのみ.



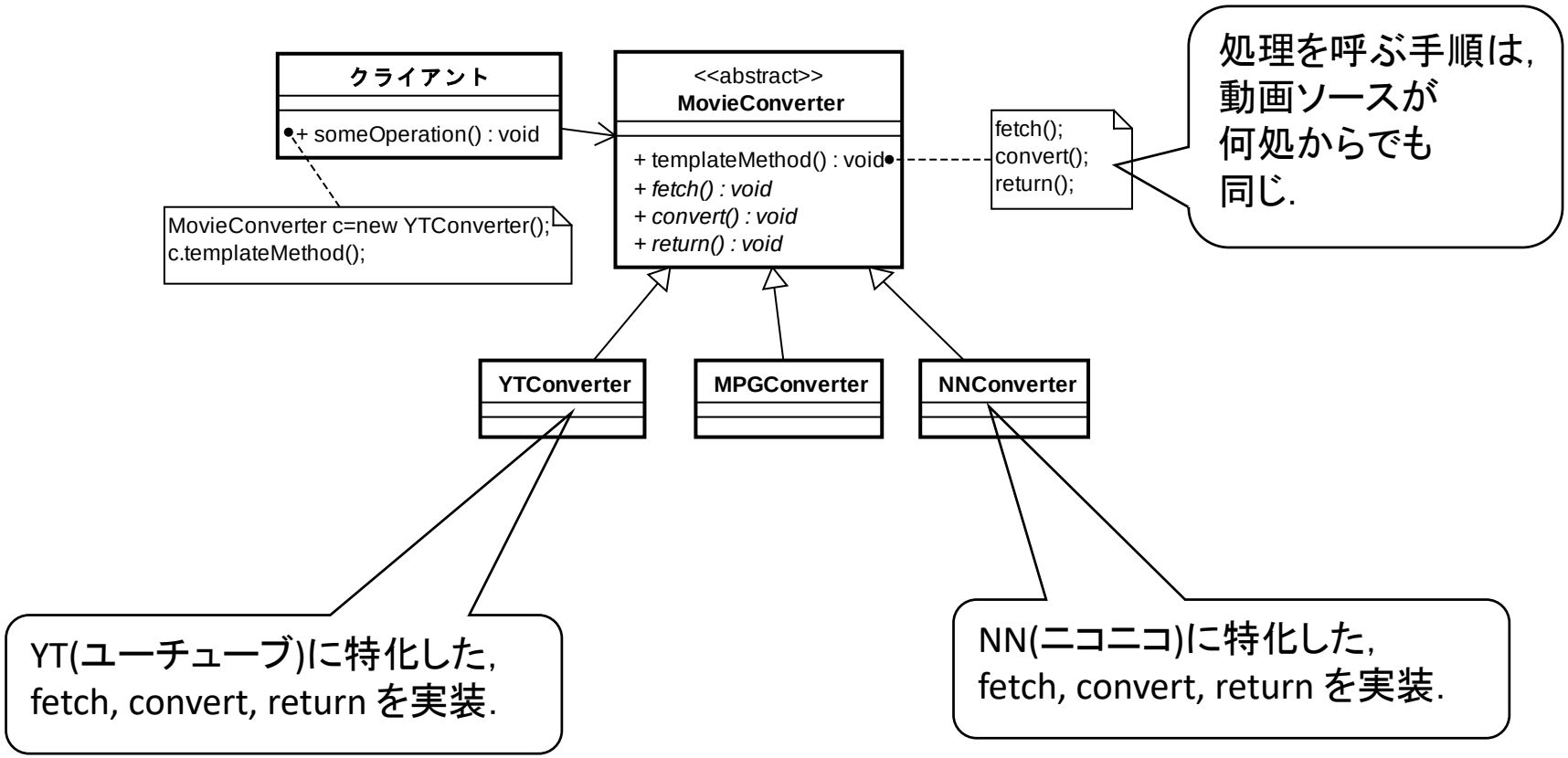
テンプレートメソッド

- Template Method

- 処理手順の大まかな流れは共通だが，個々のステップでの処理は異なるような処理群があるとする．
  - 一部のステップが同じ場合，一箇所に書けばよい．
- それら処理群を，一括して管理し，場合によっては，切り替えたりするのに役立つパターン．

設計の例

- 異なるソースの動画をmp4動画に変換し，ソースに戻すような処理を一般化.

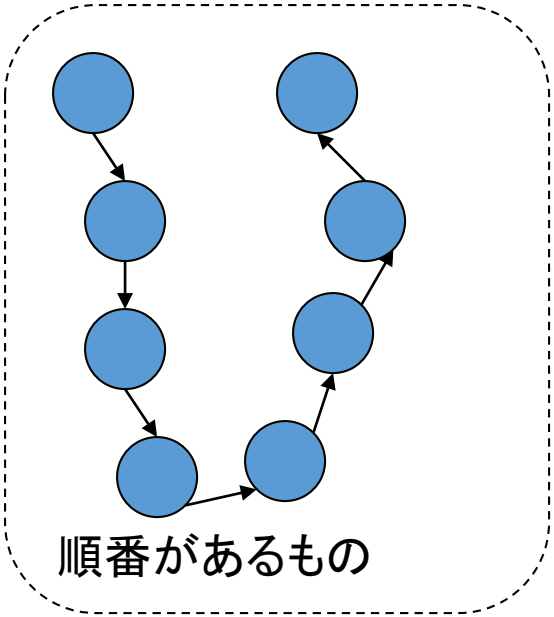


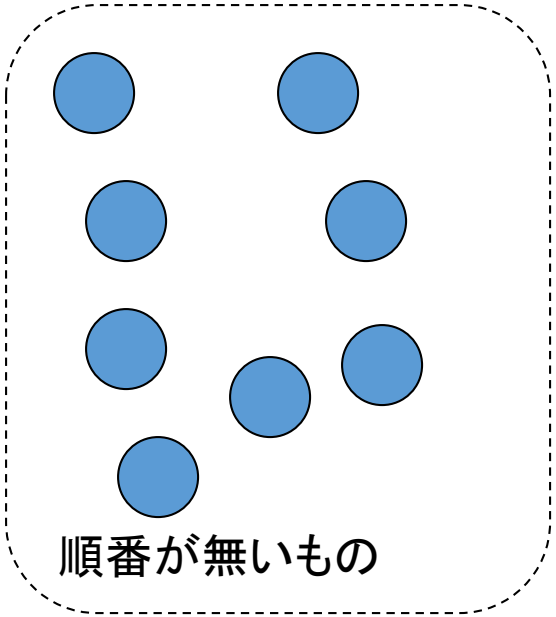
イテレーター

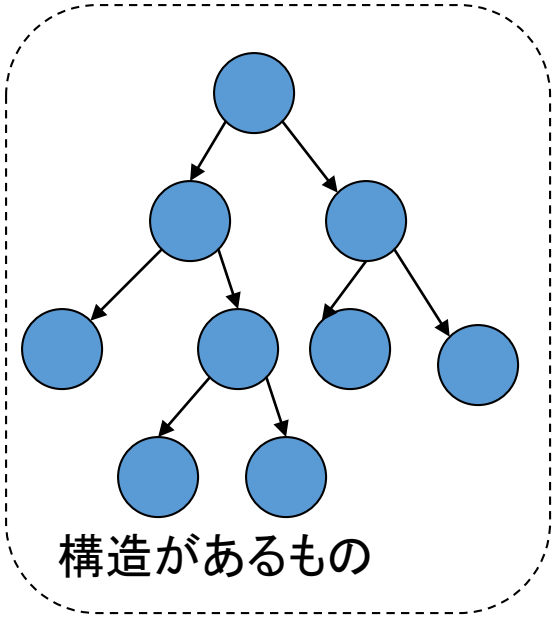
- Iterator

- 「物の集まり」と「列挙する」という概念を，それぞれ独立したクラスとして扱う考え方．
- 集合，リスト，木等，集まり方に関係なく，要素を列挙するということを一般化している．
- 一つの物の集まりに対して，複数の列挙ができるので便利．
  - 例えば，ある「人の集まり」に対して，「体重を計った人」の列挙順と，「身長を測った人」の列挙順は別インスタンスとして扱える．
- JavaのCollection系クラスでは，コレを使っている．

Iterationの考え方



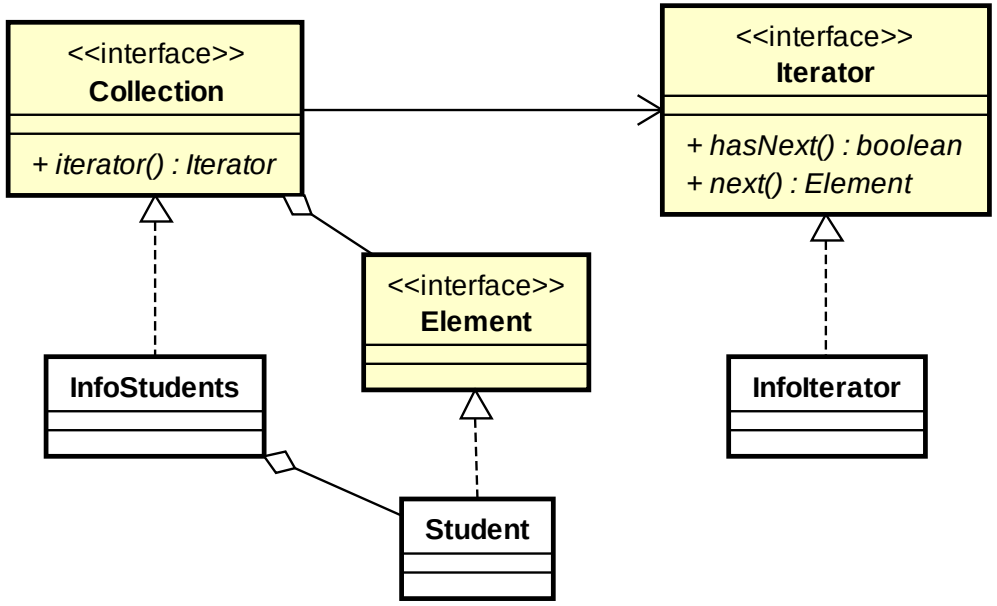




「ものの集まり」に関して，集まっている構造に関係なく，
中の要素を列挙する仕組み．
⇒「列挙する」という動作を抽象化

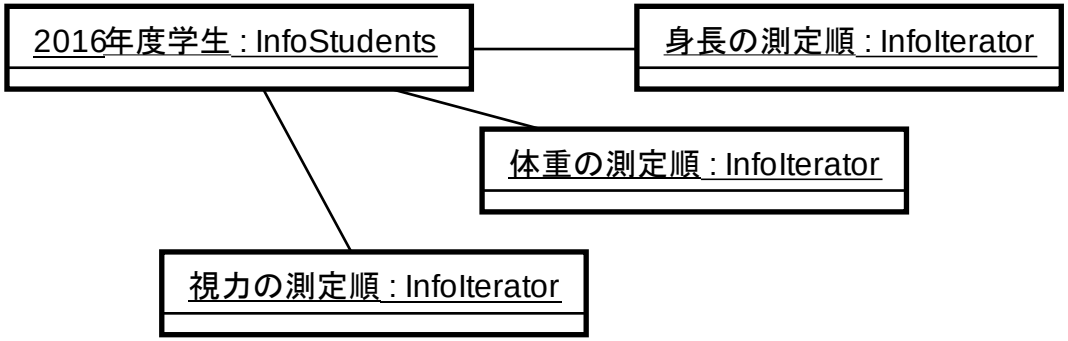
設計の例

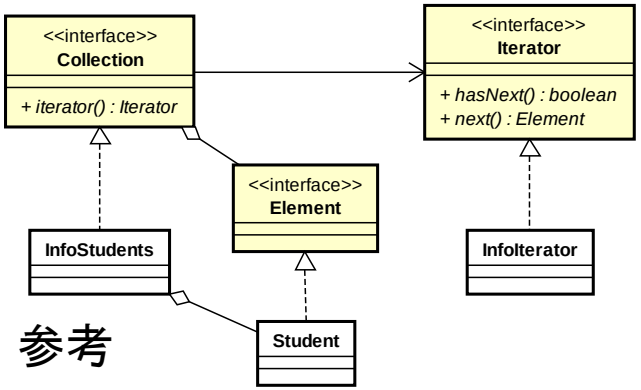
- 学生の集まりに対して，列挙を設定できる.
- hasNext: 列挙が全部終わってるかどうか判定.
- next: 列挙が終わって無い場合，次に列挙する要素.



インスタンスの例

- 者の集まり「2016年度学生」に対して，3つの異なる列挙順をオブジェクトとして管理できる.
- 実際，身長，体重，視力等は，学生によって，どんな順番に測定するか異なる.
- しかし，最終的には，全員，それぞれ測定しないといけない.
- そんなのを，わりと簡単にモデル化できる.





デザインパターンの分類

		目的		
		生成	構造	振る舞い
範囲	クラス	Factory Method	Adapter	Interpreter Template method
	インスタンス	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

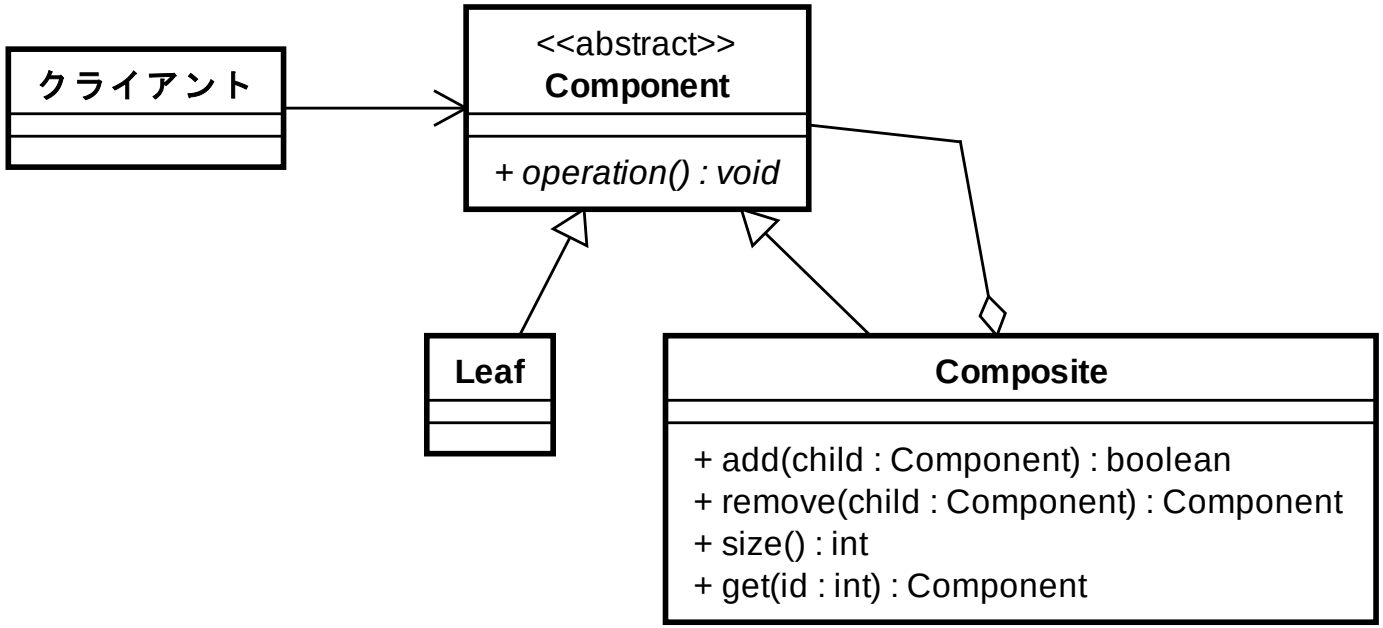
コンポジット

- Composite

- 階層構造，木構造を表現する強力なパターン.
- 例えば，OSのファイルシステムの構造，組織の階層等.
- 多分，階層構造を表現するこれ以上良い方法は無い.

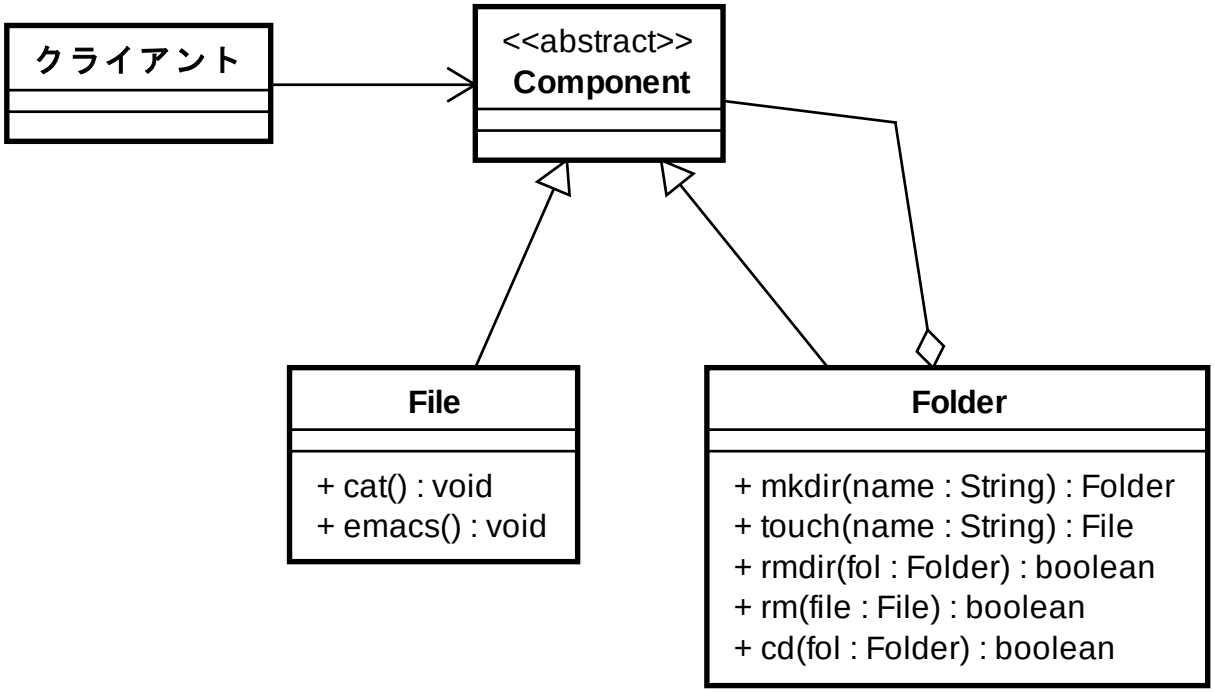
一般的な表現

- ファイル構造を例にとれば，Leaf がファイル，Composite がフォルダ(ディレクトリ)に相当.
- 無論，適宜，メソッド等を追加してよい.

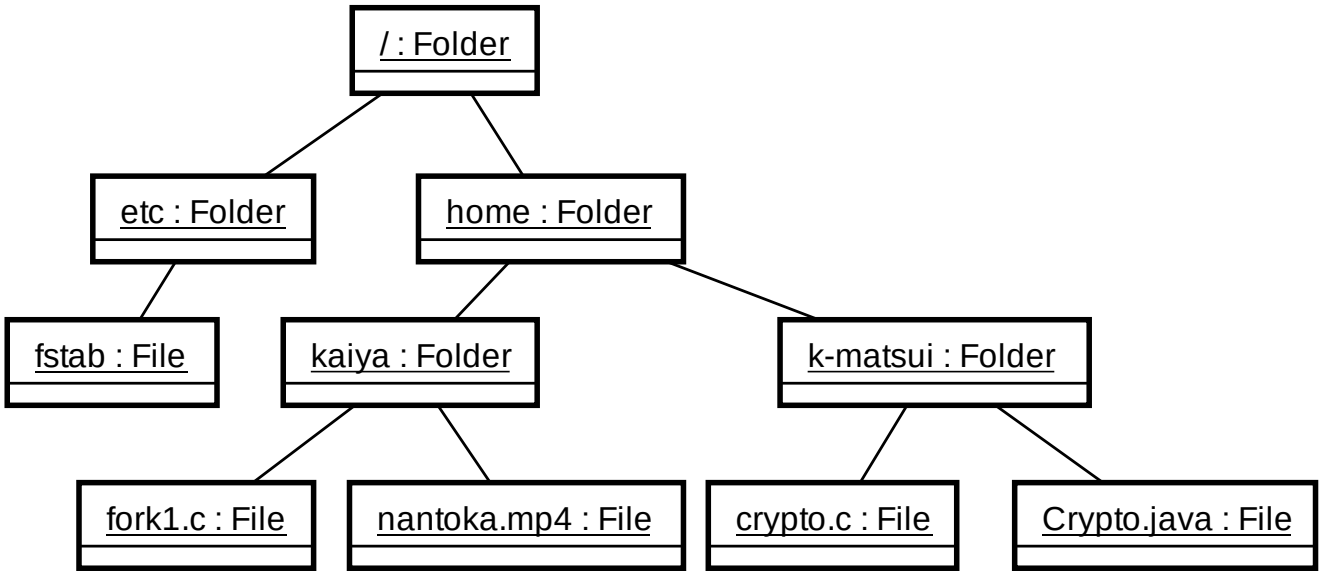


ファイルシステムの例

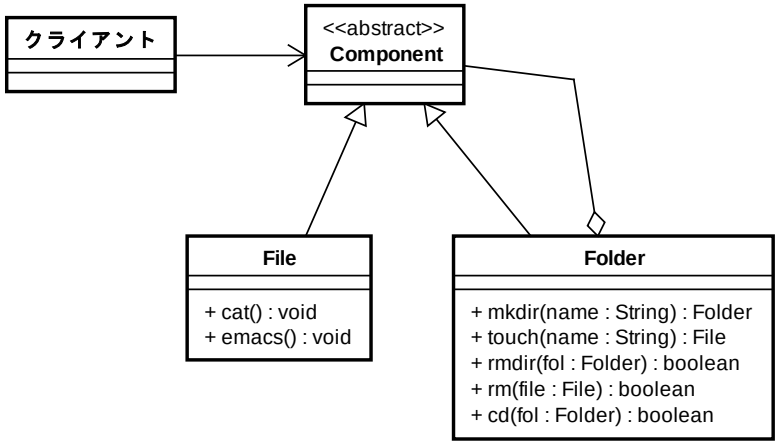
- UNIX風のコマンドをメソッドとして実装.
- 実際，UNIXでは，ファイルとフォルダの作成，削除のコマンドが異なるため，それも準拠.



インスタンスの例


/ : Folder
etc : Folder
home : Folder
fstab : File
kaiya : Folder
k-matsui : Folder
fork1.c : File
nantoka.mp4 : File
crypto.c : File
Crypto.java : File

参考


クライアント
<<abstract>>
Component
File
+ cat() : void
+ emacs() : void
Folder
+ mkdir(name : String) : Folder
+ touch(name : String) : File
+ rmdir(fol : Folder) : boolean
+ rm(file : File) : boolean
+ cd(fol : Folder) : boolean

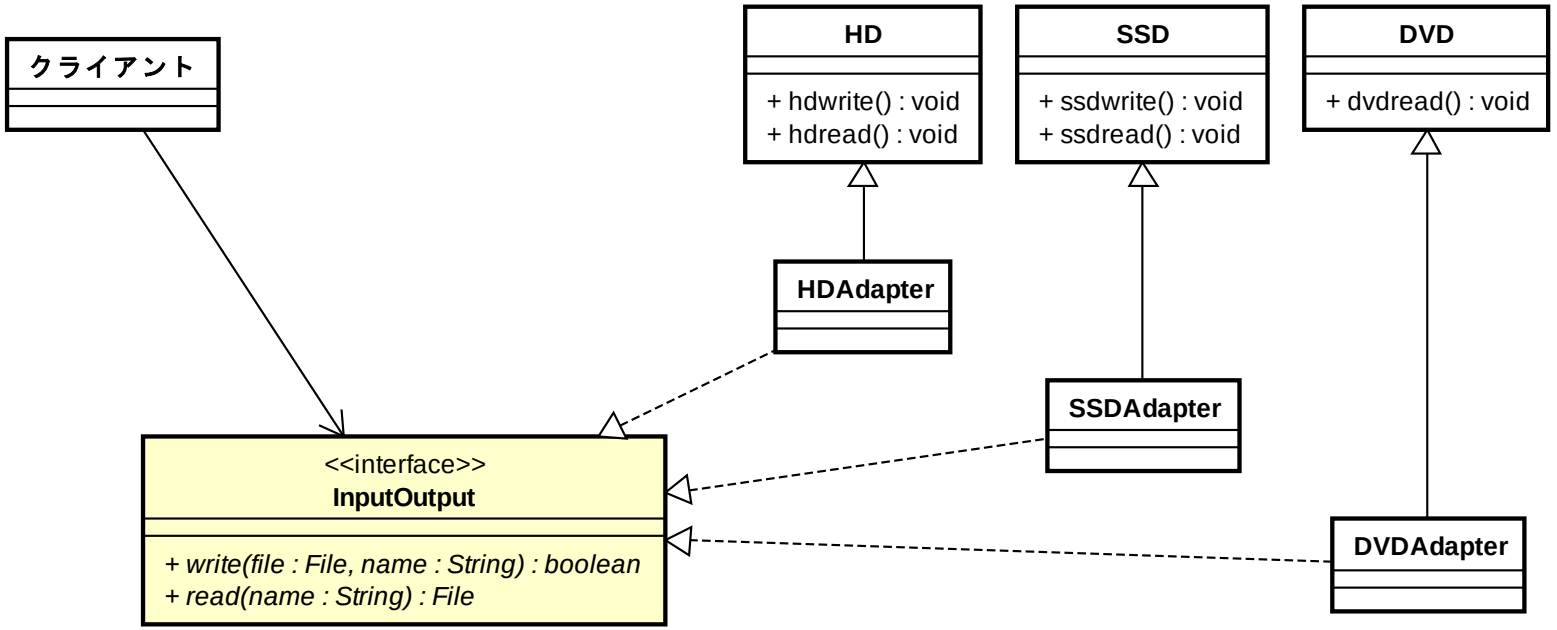
アダプター

- Adapter

- 既存のクラス群に対して，ある統一した処理を施したいが，それぞれメソッド名がばらばらであったり，そもそも，一つのメソッドで実現されてなかったりする場合がある.
- それを統一して扱うための「アダプター」を，それぞれのクラスにかぶせるためのパターン.

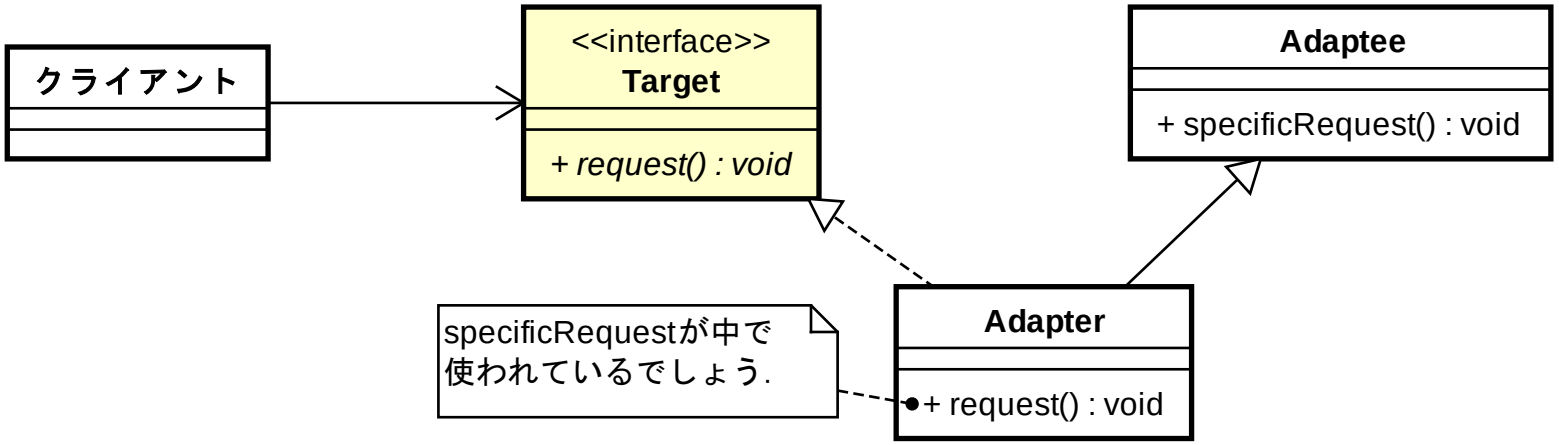
例

- 例えば，OSに接続されている，HD, SSD, CD, DVD, USBメモリ等は実際にはデータの読み書きの処理内容はばらばらである.
- これらを統一して扱うためのパターンが以下.



一般形

- Target: アタプターを規定するインタフェース
- Adaptee: アダプターをとりつける既存クラス
- Adapter: Targetに基づき作成されたアダプター



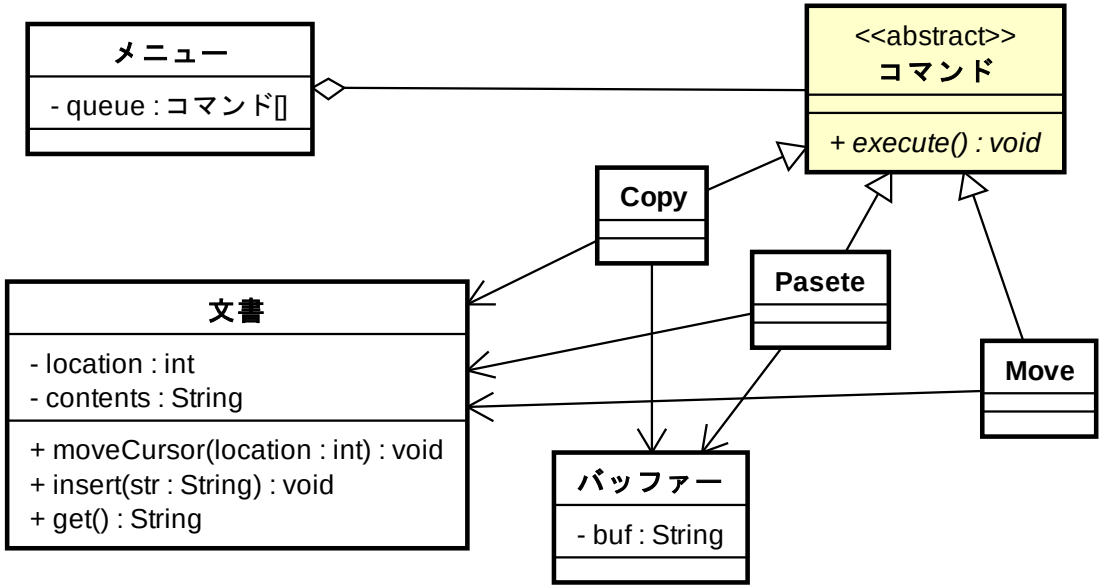
コマンド

- Command

- GUIアプリのメニューにあるようなコマンド選択肢自体を独立したオブジェクトとして扱う設計.
- 操作履歴を容易に残せるため，undo 等の実装も容易.
- また，操作対象(例えば文書ファイル等)の挙動が遅い場合でも，コマンドを待ち行列に保持することができる.
  - 結果の反映が遅くても，コマンド自体は入力できる.

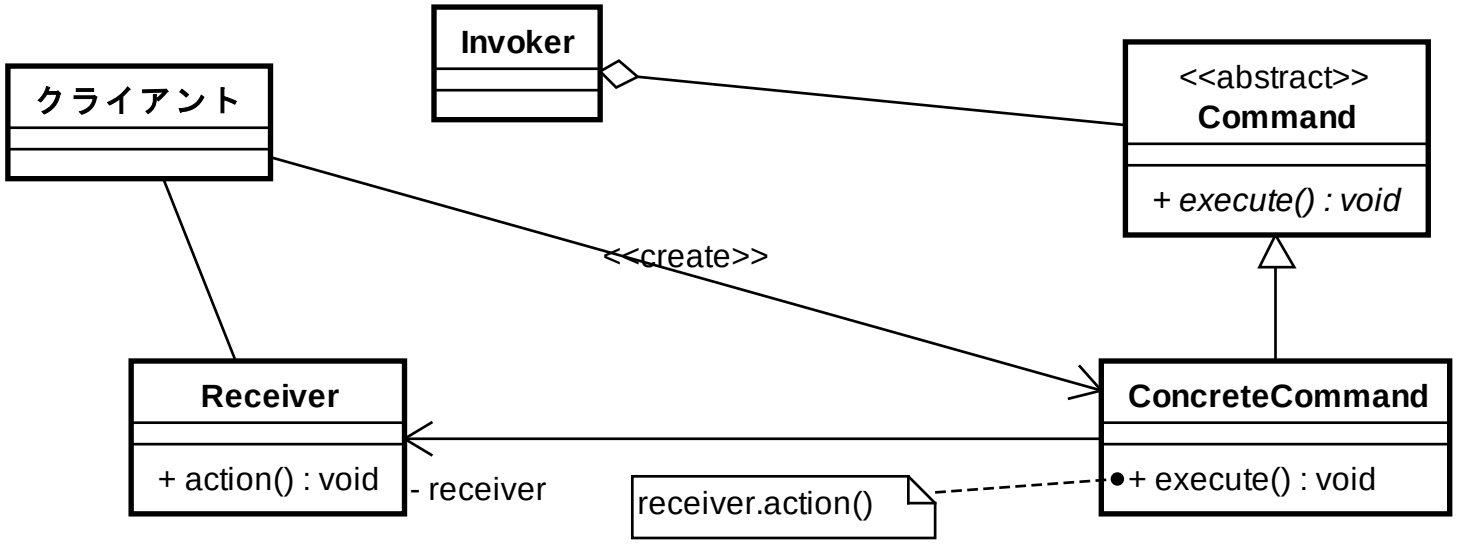
設計の例

- 簡単な文書エディタ
- Copy: 文書.get()を呼んで，現在の文字を取得.
  - バッファーに取得したものを保持.
- Paste: バッファーからstrを取得.
  - 文書.insert で文書に追加.
- Move: 文書.moveCursor でカーソル移動.



一般形

- Invoker: メニュー等に相当
- Command: コマンド一般を指す
- ConcreteCommand: 具体的な個々のコマンド
- Receiver: 実際にコマンドが適用されるデータ等



ファサード

- Façade

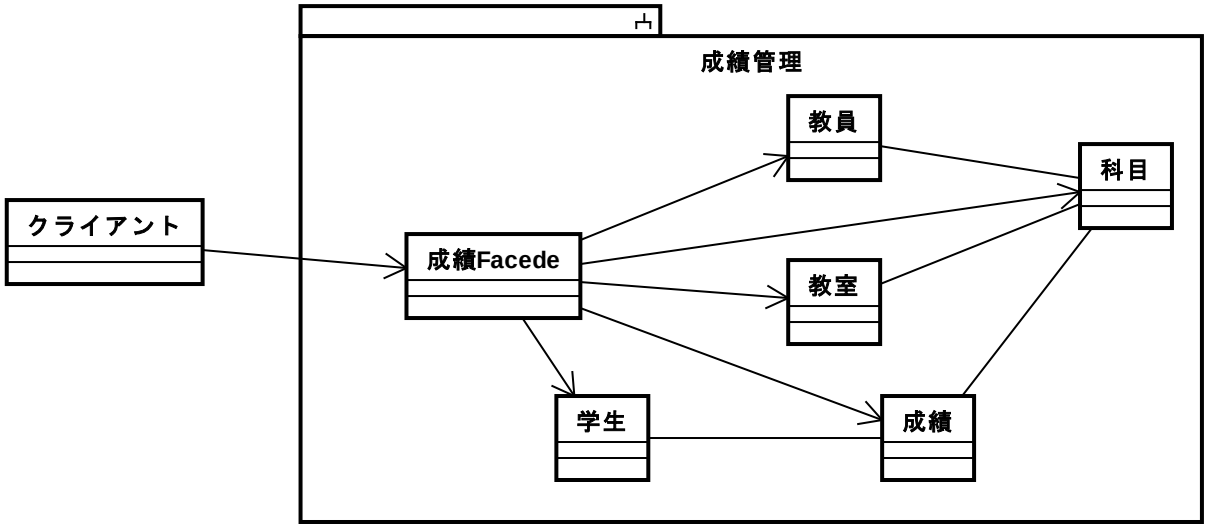
- サブシステムにまとめたクラス群への窓口を一手に引き受けるクラスを導入すること.
- これによって，サブシステム内の構造を変更しやすくなる.
- 名前は建物のファザードに

由来するらしい.

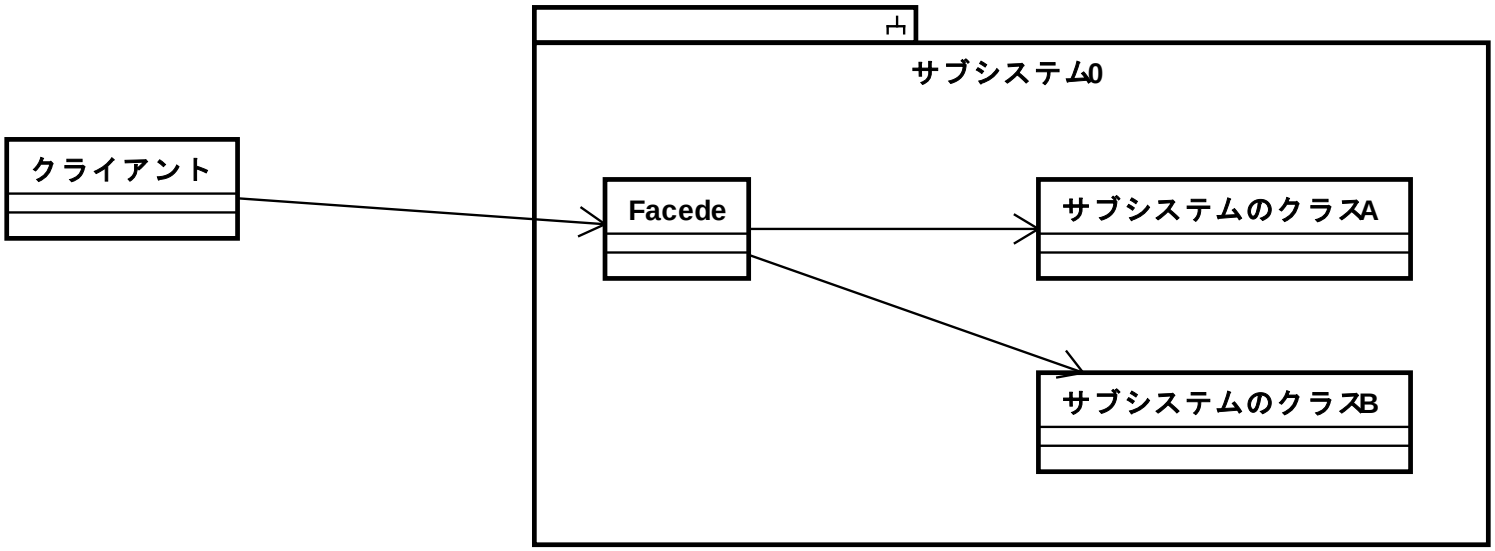
右記のような感じ.

設計の例

- 学生の成績を管理するクラス群にアクセスするクライアントアプリを考える.
- ここのクラスに直接アクセスするのではなく，サブシステムの代表をするクラスを導入する.
- 後に，学生成績のドメインモデルに変更があっても，クライアントへの影響は少ない.



一般形



- 尚，UMLにはサブシステムとパッケージがあり，前者はクラス等との関連が持てる点で後者と異なる.

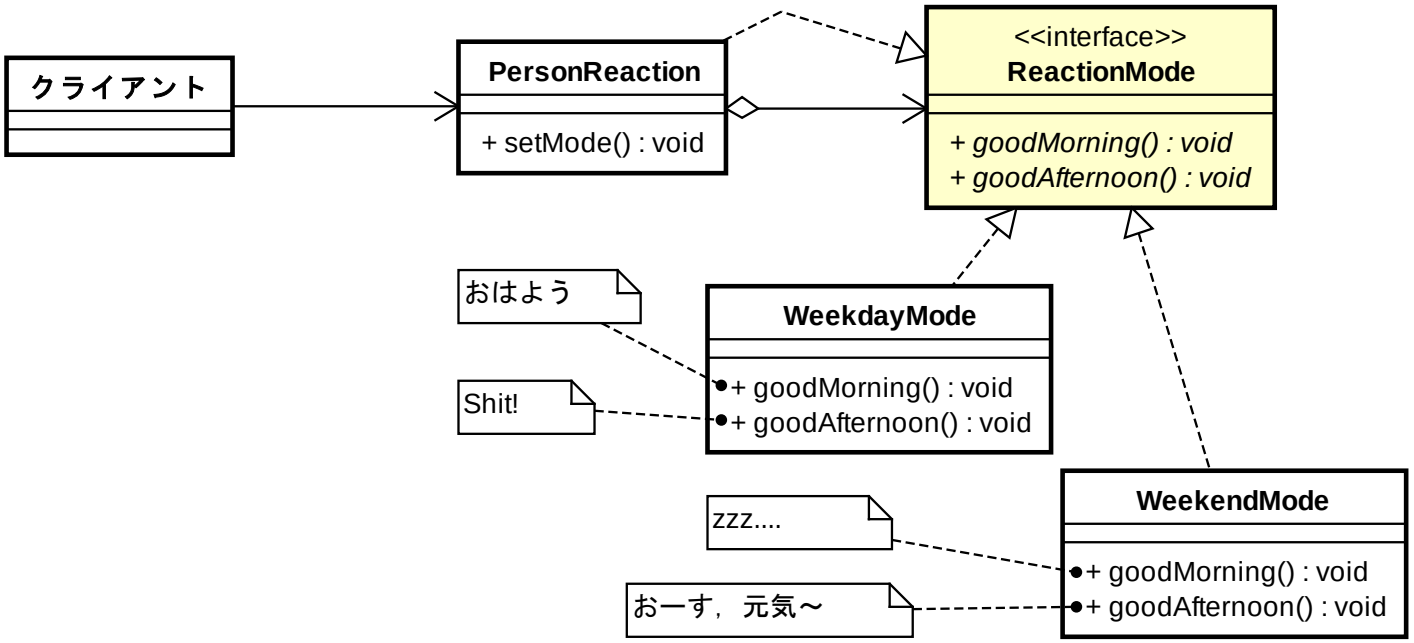
ステート

- State

- オブジェクトの状態が変化した際，その振る舞いが変化することをデザインしたい.
- 単純にオブジェクト内に多数の条件分岐を書くのではなく，状態毎にクラスを定義し，その振る舞いをクラス毎に記述してゆくデザイン手法.
- 多数の条件分岐があるより，多少見やすいかも.

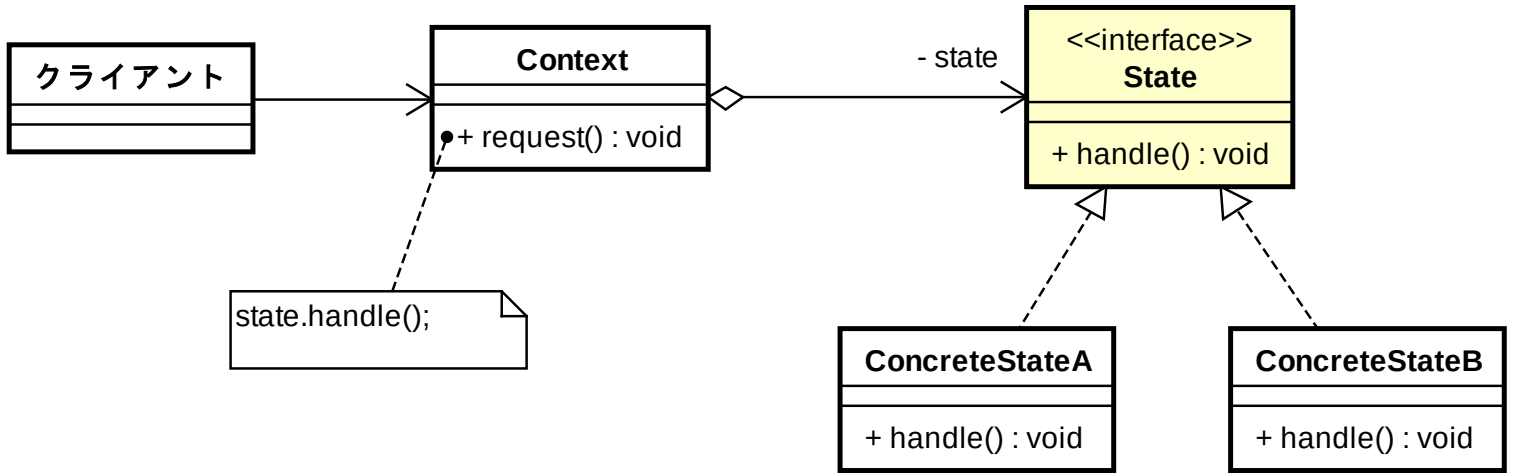
設計の例

- ニートじゃない人は，平日と週末では反応が違うだろう.
- 例えば，週末午前に「おはよう」といっても寝てるかもしれない.
- また，平日午後に「良い午後だね (Good afternoon)」といっても，「ちくしょう (Shit)」といってくるかもしれない.



一般形

- Stateが状態によって変化する振る舞いのリストに相当.
- ConcreteStateA, Bで，実際，個々の異なる振る舞いを規定.
- クライアントは Context にアクセスし，その時のStateに依存した振る舞いを返す.



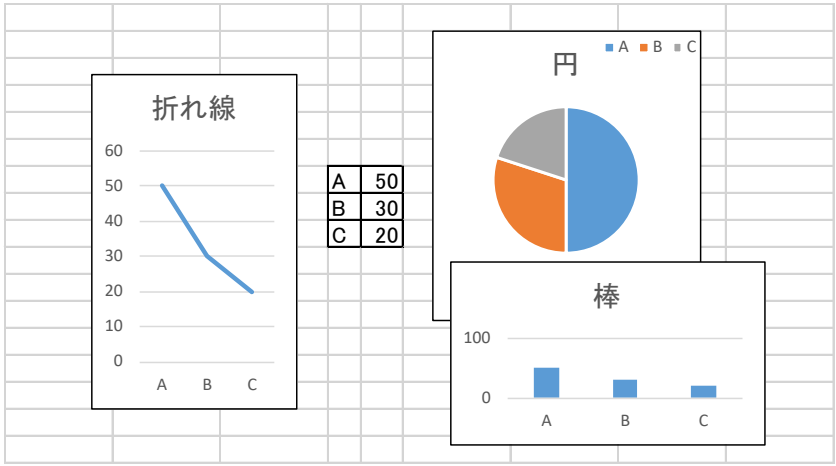
オブザーバー

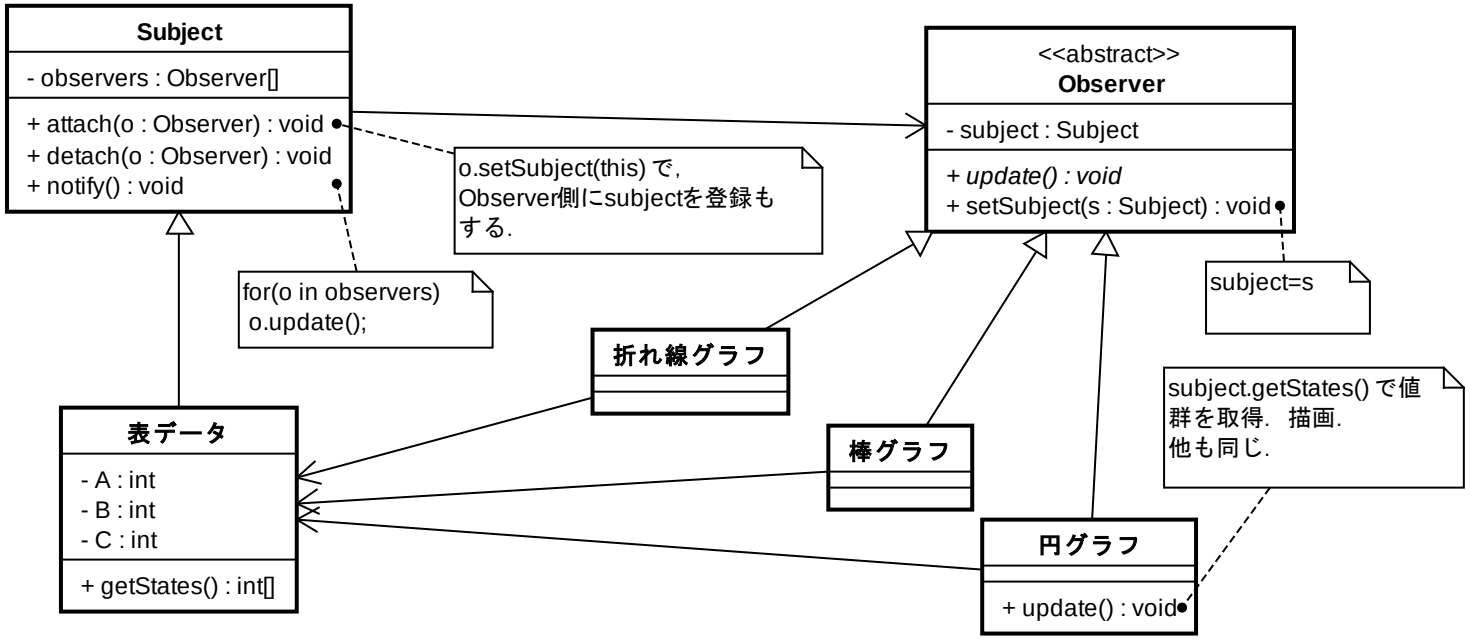
- Observer

- あるデータを持つオブジェクトAと，そのデータに依存する他のオブジェクトX,Y,Zがあるとする.
  - このAを Subject と呼ぶ.
  - X,Y,Zを Observer とそれぞれ呼ぶ.
- Aの変化にX,Y,Zが自動的に追従する仕組みをうまく実現する定石のデザイン.
- Observerを途中で追加，削除できるのでとっても便利.

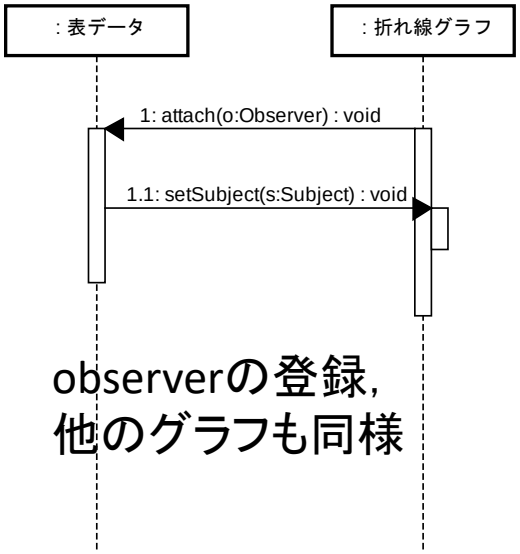
設計例

- 右のように，表データを多様なグラフで表現するとする.
- データが変化するとグラフもそれに追従する.



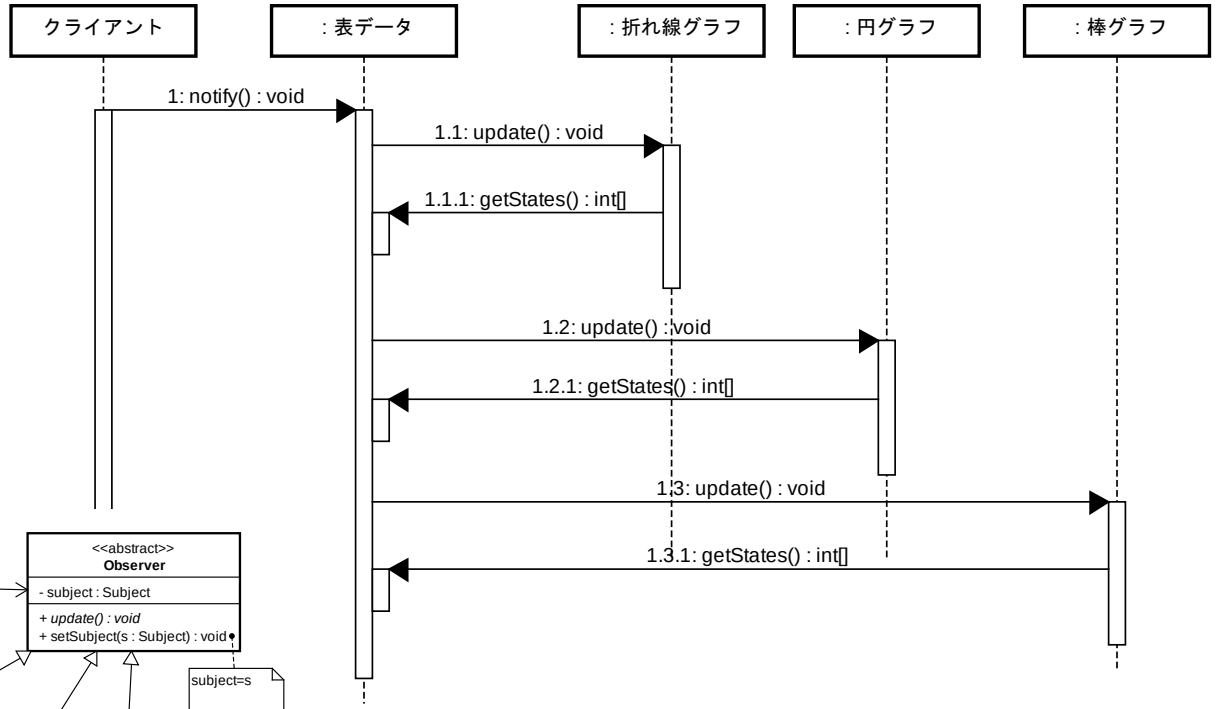


シーケンス図

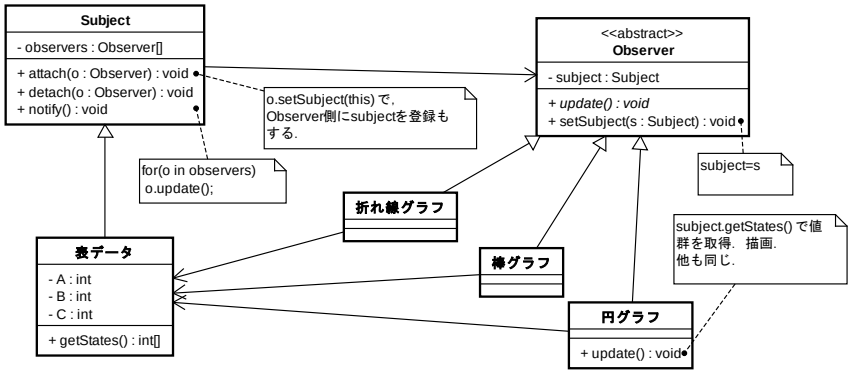


observerの登録,
他のグラフも同様

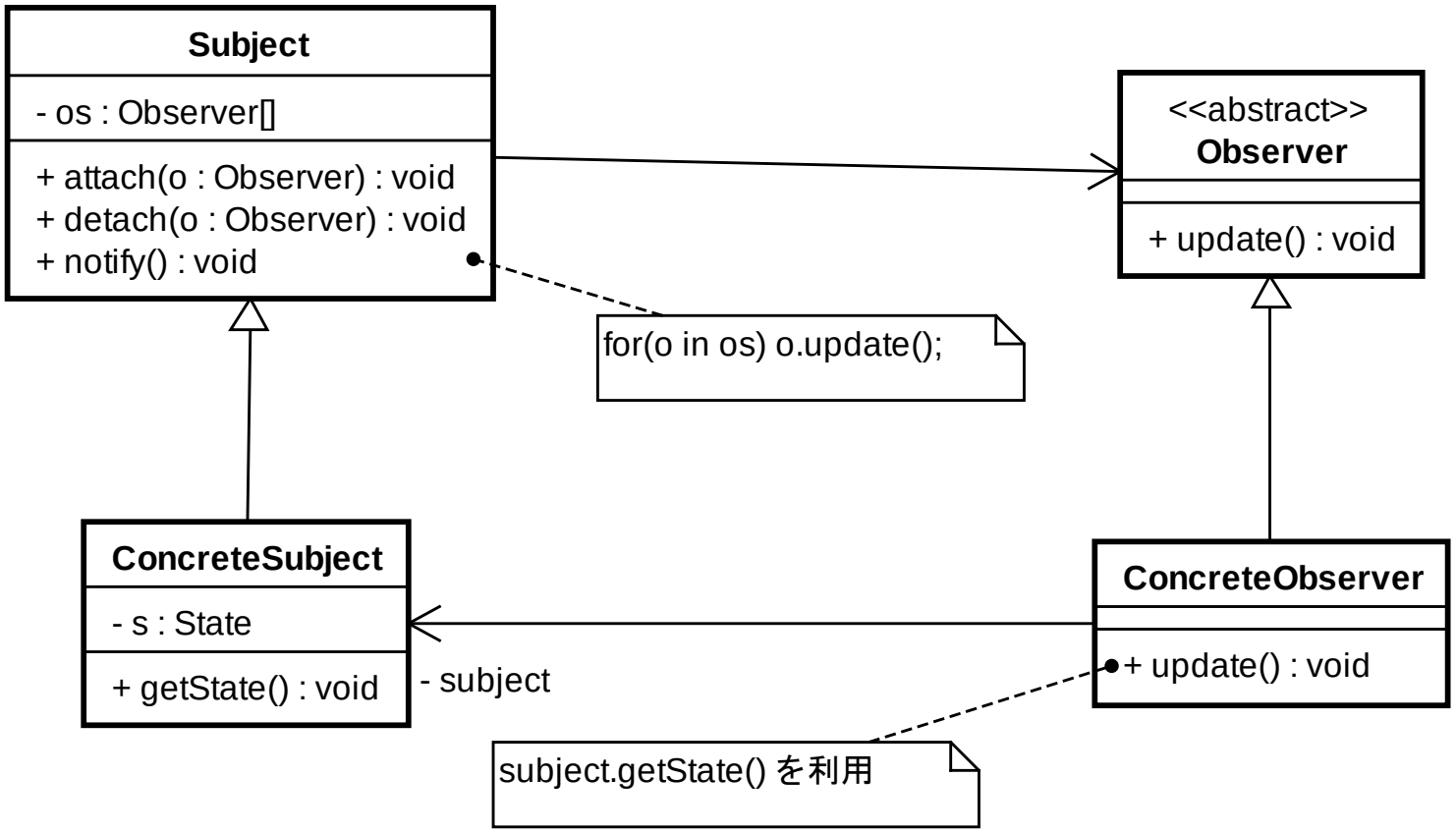
アップデートの様子



参考



一般形


Subject
- os : Observer[]
+ attach(o : Observer) : void
+ detach(o : Observer) : void
+ notify() : void
<<abstract>>
Observer
+ update() : void
for(o in os) o.update();
ConcreteSubject
- s : State
+ getState() : void
- subject
ConcreteObserver
+ update() : void
subject.getState() を利用

本日は以上