Design Pattern 学習メモ A.HO

Design Pattern

- オブジェクト指向でプログラムを作る際の定跡
- "Design Patterns: Elements of Reusable Object Oriented Software" (1995) で示された 23 種類が 有名
 - 通称 GoF本
 - 著者:GoF = Gang Of Four
 - E.Gamma、R.Helm、R.Johnson、J.Vissides

Ol [COMMAND]

■ 処理データと処理ロジックが格納されている Commandオブジェクトを汎用的なInvokerに実行させる



Client

```
main(){
  Command cmd = new XXCommand(args);
  Res res = Invoker.action(cmd);
```

Invoker

Res action(Command cmd){ return cmd.execute();

[COMMAND]の応用例

■タスク制御



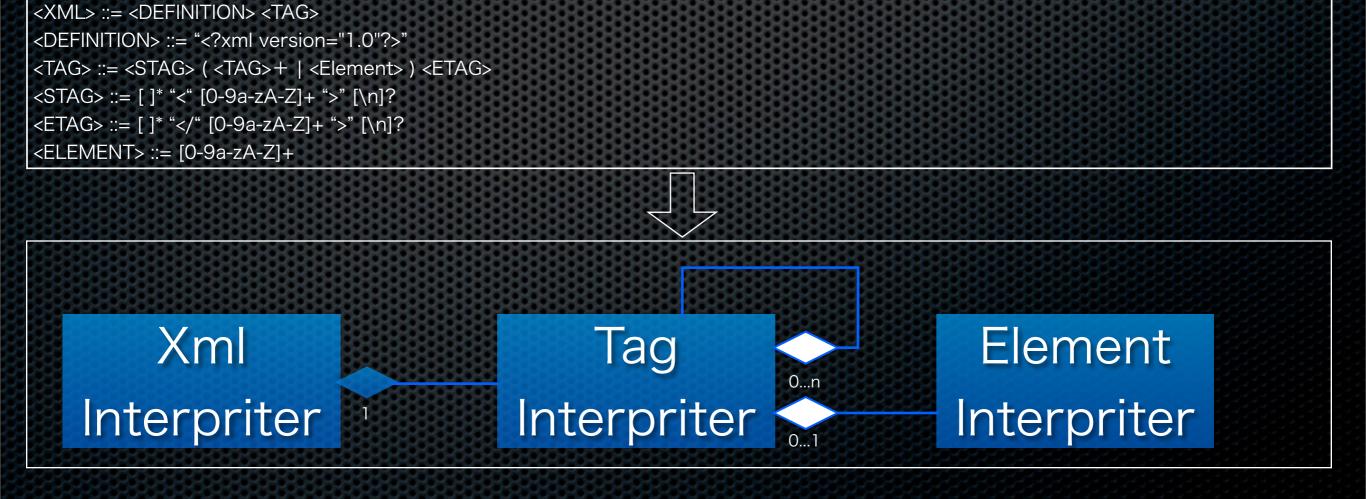
■ Undo (→ 16 [MEMENTO])



Command Stack

02 [INTERPRITER]

- 文章解析プログラムを作るときに使う
- ▼ 文法構造を解析プログラムのオブジェクト構造に写し取る
- 手続き型言語の再起処理設計のオブジェクト指向版



XmlInterpriter#interprit() に解析対象のXMLを渡すと構文木の形でオブジェクト構造が生成される。

XmlInterpriter

String xml宣言; TagInterpriter chileTag;

```
void interprit( String ){
  int p; // 処理済み文字数
  p = XML宣言の読み取り(String)
  childTag.interprit(p,String);
}

String dump(){
  print( xml宣言);
  childTag.dump();
}
```

TagInterpriter

String tagName;

List<TagInterpriter> childTagList; ElementInterpriter childElem;

```
int interprit( offset, String ){
 int p; //処理済み文字数
 p = 開始タグ読み取り処理(offset,String);
 if(次はタグ){
  do{
   p = childTag.interprit(p,String);
   childTagList.add(childTag);
  }while(次は終了タグ);
 }else{
  p = childElement.interprit(p,String);
  childElementList.add(childElement);
 p = 終了タグ読み取り処理(p,String)
 return p;
String dump(){
 print( tagName );
 for( childTag : childTagList ){
  childTag.dump();
 childElem.dump();
```

ElementInterpriter

String element;

0...n

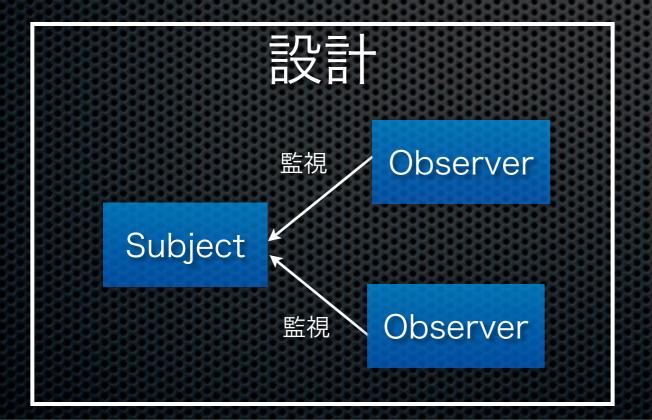
0...n

int interprit(offset, String){
 offset から"<" が出てくるまで
 読み込んで elementに格納する
 return 処理済み文字数;

String dump(){
 print (element);

03 [OBSERVER]

- Subject を多数の Observer が監視していて、Subject の状態変化をトリガーに Observer が何かをするアプリケーションを作るときに使う。
- 実装するときには、発想の転換をする。
 - Subject が、Observerを保持する
 - Subject が、状態変化時(たとえば setter が呼ばれたとき)に、Observerを呼び出す
- Listener と呼ばれることもある





■ Java には、標準で Observer パターンが実装されている。

```
package example;
public class MySubjectOperator {
  * @param args
 public static void main(String[] args) {
  MySubject subject = new MySubject();
subject.addObserver(new MyObserver());
   subject.setState("Running");
subject.setState("Stopped");
package example;
import java.util.Observable;
public class MySubject extends Observable {
 private String pState;
 public void setState(String state){
   pState = state;
   this.setChanged();
   this.notifyObservers();
 public String getState(){
   return pState;
package example;
```

```
package example;
import java.util.Observable;
import java.util.Observer;

public class MyObserver implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println("The state of MySubject is " + ((MySubject)o).getState());
    }
}
```

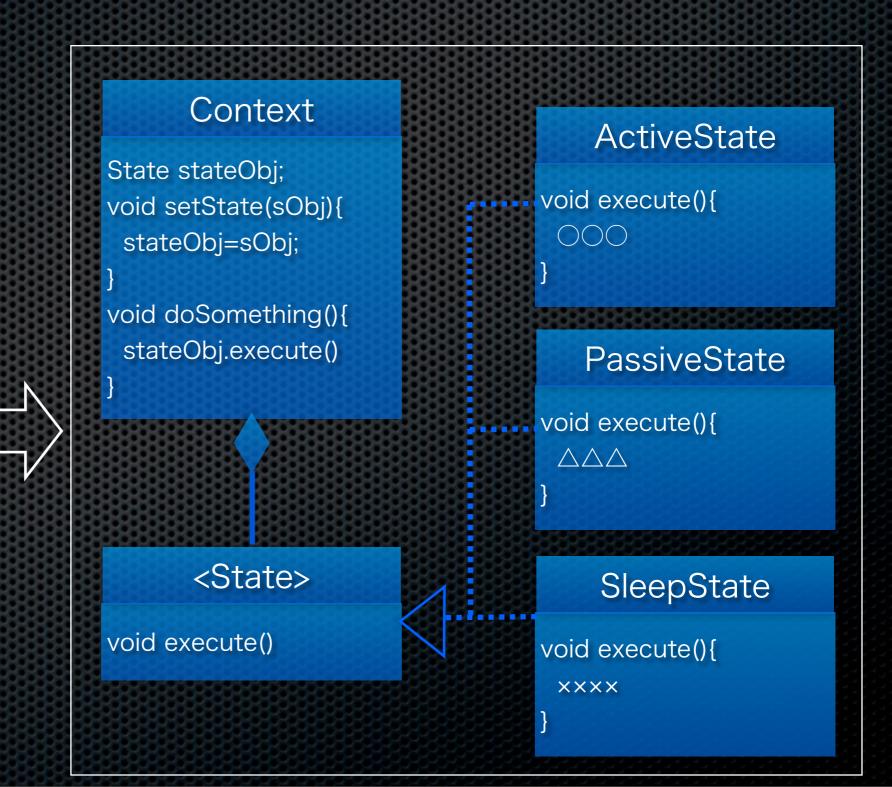
実行結果:

The state of MySubject is Running The state of MySubject is Stopped

04 [STATE]

■ 状態に意味と操作を持たせる

```
Traditional Context
int state:
void setState(s){
 state=s;
void doSomething(){
 switch(state){
 case ACTIVE:
   \bigcirc
  break;
 case PASSIVE:
   \triangle \triangle \triangle
   break:
 case SLEEP:
   XXXX
   break;
```



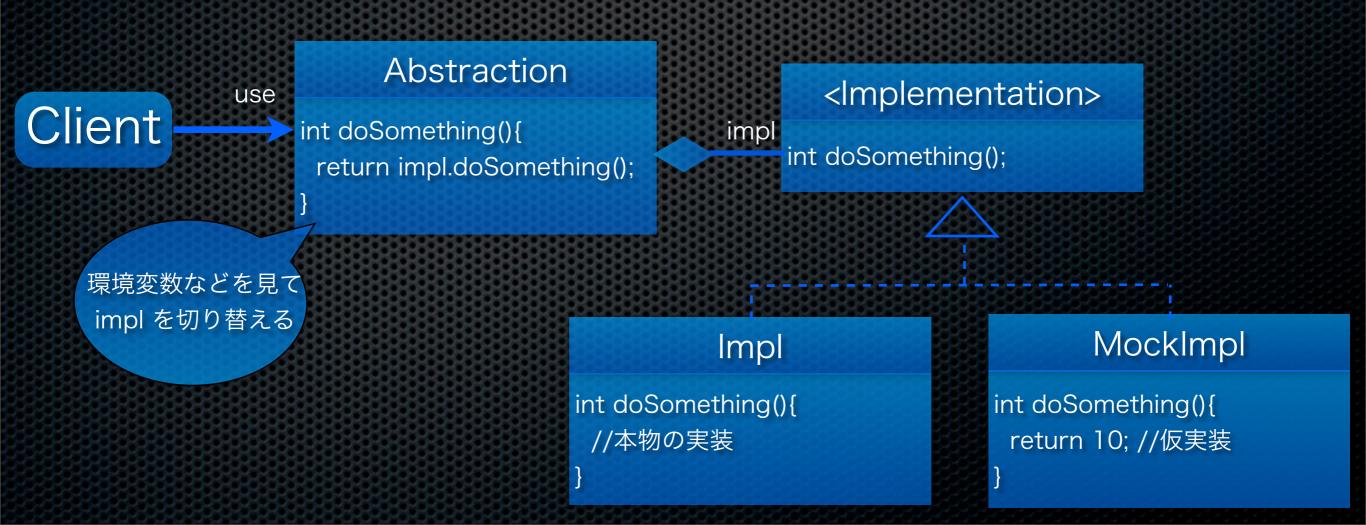
05 [ADAPTER]

■ 既存の部品(Adaptee)を新しいアプリケーション (Target)から使うためのつなぎ役



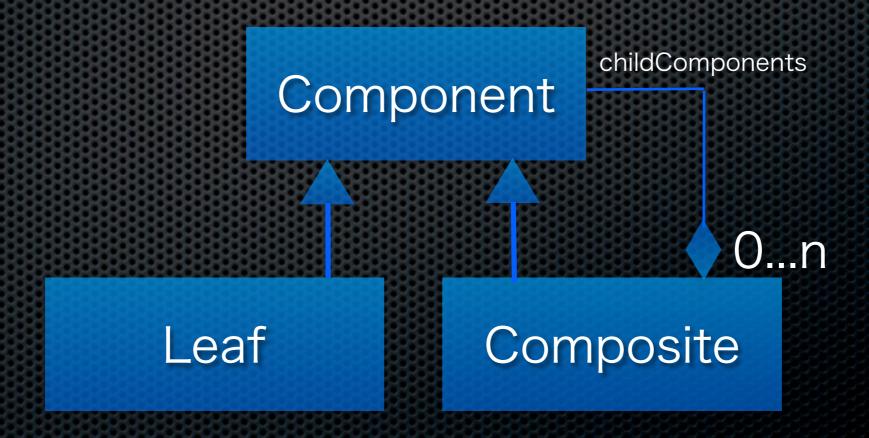
06 [BRIDGE]

- 機能(Abstraction)と実装(Implementation)を分ける
- ▼ →再利用とメンテナンス性の向上。
 - (後で実装を差し替えたくなったときに改修箇所が限定される)
- →実行環境で実装を分ける
 - (進捗の遅い別チームが作っているモジュールをMock Object にする)



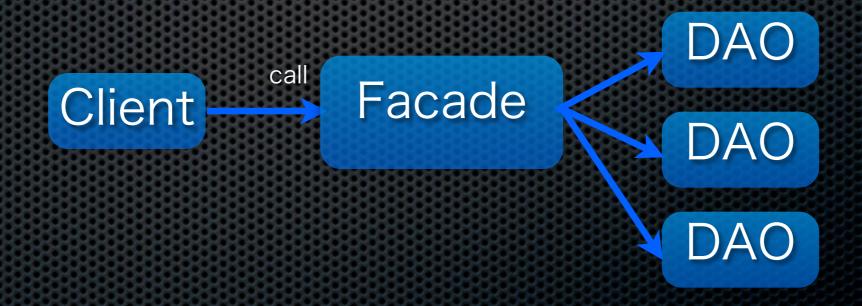
07 [COMPOSITE]

- 自分を再帰的に保持できるようにしたオブジェクト構造
- GUIフレームワークでよく使われる
 - Window、Panel、Text Field、Button は Component
 - AWT/Swing などの他に、JSF/Wicket/Tapestry 5 などのWebフレームワークでもこの設計思想が使われている。



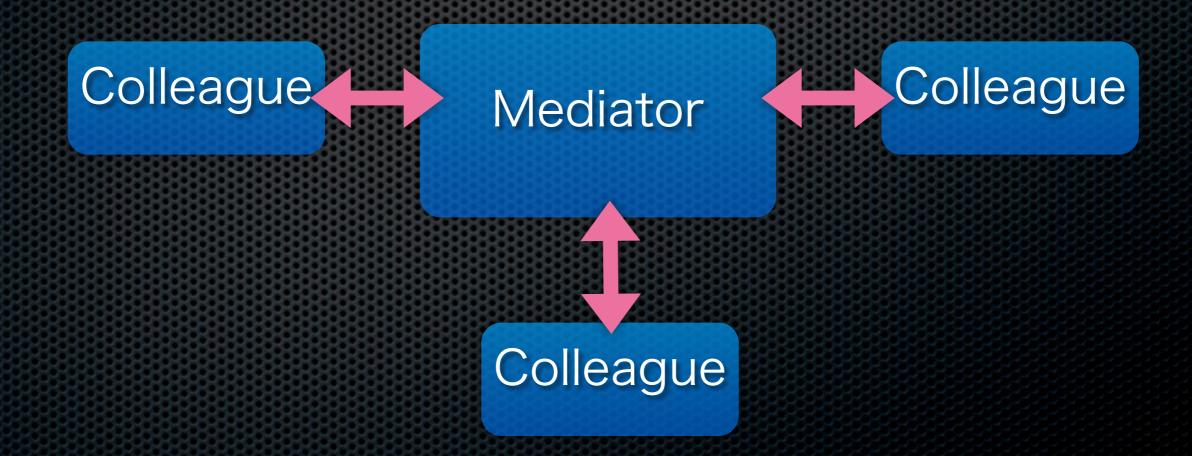
08 [FACADE]

- FACADE:建物の正面を表すフランス語
- クライアントからの処理依頼を集約する
 - →階層を明確化する
 - ■→通信量の低減



09 [MEDIATOR]

■ COLLEAGUE 同士で直接通信せずに、必ず MEDIATOR(仲介者)を通して情報のやりとりをする。



10 [TEMPLATE METHOD]

■ Java の Abstract Method

Abstract Class int doSomething(){ // 共通処理 individual(); // 共通処理 } abstract int individual();

Derivation A int individual() { 派生 A 固有の処理; } Derivation B int individual() { 派生 B 固有の処理; }

11 [VISITOR]

■ Element(データ構造) と、Visitor(処理ロジック)を 分離して再利用できるようにする。



Element A

HashMap dataA;
String accept(Visitor v){
 return v.visit(this);

ElementB

List dataB;
String accept(Visitor v){
 return v.visit(this);
}

<<Visitor>>

String visit(ElementA e); String visit(ElementB e);

Visitor 1

String visit(ElementA e){
 // e.dataA の処理1
}
String visit(ElementB e){
 // e.dataB の処理1

Visitor 2

String visit(ElementA e){
 // e.dataA の処理2
}
String visit(ElementB e){
 // e.dataB の処理2
}

Client (How to use these?)

```
Element e = new ElementA();
Visitor v = new Visitor2();
System.out.println( e.accept(v) );
```

12 [Cor(Chain of Responsibility)]

- 処理要求を自分で処理できなかったら上司に任せる
- 上に行くほど細やかな対処はできなくなるが、広い見地から対処できるように なる

HandlerA

Handler myBoss = null; void handle(request){ requestを処理する myBoss

myBoss

HandlerB

```
Handler myBoss = a;
void handle( request ){
    if(requestを処理できない){
        // 上司に任せる
        myBoss.handle(request);
    }
    // 自分で処理する
}
```

HandlerC

```
Handler myBoss = a;
void handle( request ){
  if(requestを処理できない){
    // 上司に任せる
    myBoss.handle(request);
  }
  // 自分で処理する
}
```

HandlerD

```
Handler myBoss = b;
void handle( request ){
  if(requestを処理できない){
    // 上司に任せる
    myBoss.handle(request);
  }
  // 自分で処理する
```

myBoss

myBoss

HandlerE

```
Handler myBoss = b;
void handle( request ){
    if(requestを処理できない){
        // 上司に任せる
        myBoss.handle(request);
    }
    // 自分で処理する
```

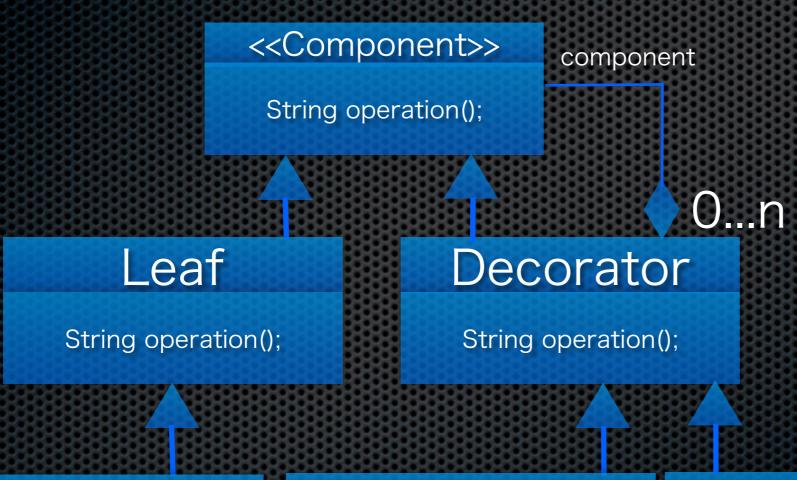
d.handle(req)

13 [DECORATOR]

■ Java の IOStream のように、処理を付け加えられる処理構造

Decorator d = new FruitDecorator(new CreamDecorator(new SpongeCake(小麦粉,卵))); System.out.println(d.operation());

→スポンジケーキを作った。そこにクリームを塗った。そこにフルーツをのせた。



SpongeCake

String operation(){
return "スポンジケーキを作った。";

CreameDecorator

String operation(){
return compnent.operation()
+ "そこにクリームを塗った。";

FruitDecorator

String operation(){
 return component.operation()
 + "そこにフルーツをのせた。";

14 [FLYWEIGHT]

- あるクラスの全インスタンスで共通な部分を Flyweight Object に切り出す。
 - →メモリの節約、インスタンス生成時間の節約

use

- Flyweight Object は、状態を持ってはいけない(immutable でなければならない)
 - どこから呼ばれるか分からないので

Client

HeavyObj obj=
 FlyweightFactory.createHeavyObj();
obj.doSomething();

FlyweightFactory

Flyweight flyweight = Flyweight.getInstance(); public static HeavyObj createHeavyObj(){ return new HeavyObj(flyweight);

instantiate

HeavyObj

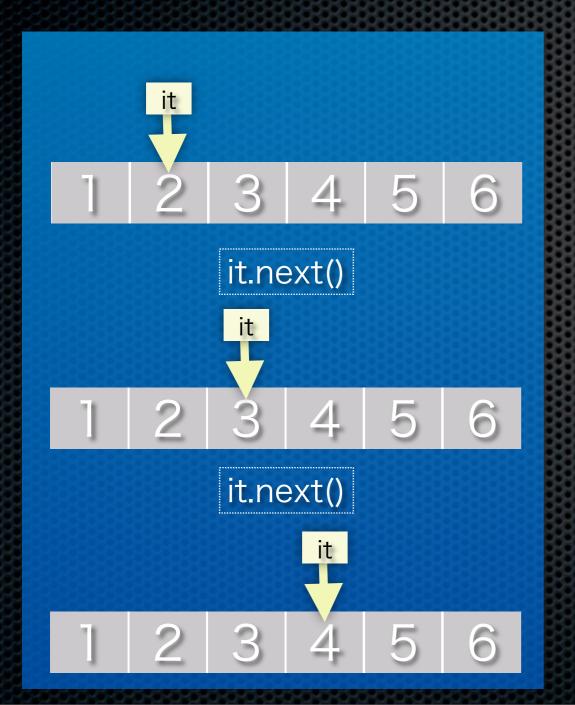
Flyweight flyweight;
public HeavyObj(flyweight){
 this.flyweight = flyweight;
}

Flyweight

HeavyObjの共通部分 (大きなデータ)

15 [ITERATOR]

- リストから順番通りに要素を取り出す
- Javaでは標準APIの実装がある
 - java.util.Enumeration、java.util.lterator、java.util.ListIterator



JavaのIterator

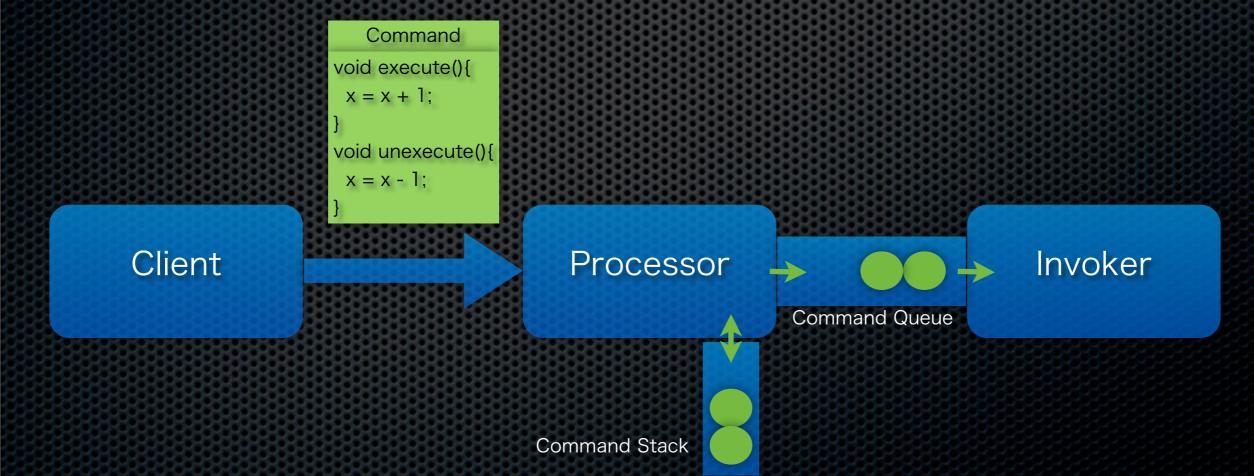
```
Iterator it = list.iterator();
while(it.hasNext()){
   ElementObj elem = (ElementObj)it.next();
   elem.doSomething();
}
```

JDK5から以下のようにも書ける

```
for(ElementObj elem : list){
   elem.doSomething();
}
```

16 [MEMENTO]

- Memento:形見、記念品、思い出の品
- Undo 機能のついた COMMAND
 - Processorは、CommandをStackにとっておく
 - Undo したいときは、Stack からコマンドを取り出して、Invoker に unexecute() を実行させる。



17 [PROXY]

- 本体(RealSubject)と同じインタフェースを持つ代理人 (Proxy)に処理を依頼する
- 通信経由で処理を実行する場合に、クライアント側に本体と同じインタフェースの Proxy を作ることがよく行われる。

<<Subject>>

RES doSomething(req)

Client

Proxy.doSomething(req);

Proxy

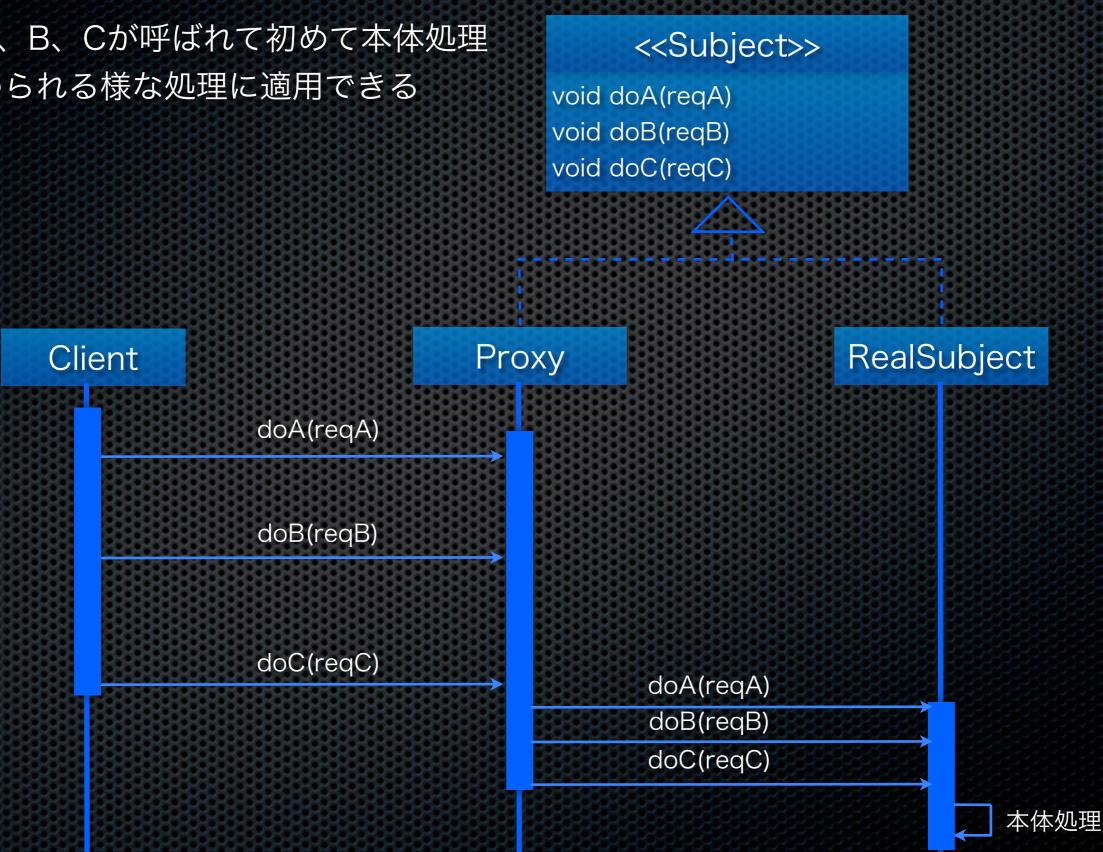
RES doSomething(req){
// RealSubject の lookup
return real.doSomething(req);

RealSubject

RES doSomething(req){
// 本体処理
return res;

RMI/IIOP etc.

- 非同期遅延実行などもProxyで実現できる
- 処理A、B、Cが呼ばれて初めて本体処理 が始められる様な処理に適用できる



18 [STRATEGY]

■ interface が同じで、処理内容の異なる algorithm を動的に使い分ける時に使う

```
Client (How to use these?)

context.setStrategy(new PlusStrategy());

context.doSomething(1,2);
```

Context

```
Strategy strategy;
void setStrategy(Strategy s){
   strategy = s;
}
int doSomething(int x,int y){
   return strategy.algorithm(x,y);
}
```



PlusStrategy

```
int algorithm(int x,int y){
  return x + y;
```

MinusStrategy

int algorithm(int x,int y){
 return x - y;

19 [ABSTRACT FACTORY]

- 背後にたくさんの関連オブジェクト(Family)を持っていて、複雑な初期化処理が必要なオブジェクト(Product)をインスタンス化したい。
- インスタンス化を行う工場(Factory)を作って、Client から Product のインスタンス化処理を分離する
- ▼ Factory も Client から直接クラスをたたくのではなく、インタフェース (AbstractFactory)を介する用にした方が変更に強い

Client

AbstractFactory factory = new ConcreateFactory(); AbstractProduct product = facotry.createProduct2() product.doSomething();

<<AbstractFactory>>

AbstractProduct createProduct1()
AbstractProduct createProduct2()

ConcreteFactory

AbstractProduct createProduct1()
AbstractProduct createProduct2()

<<AbstractProduct>>

Product 1

Family

Product 2

Family

20 [BUILDER]

- Director (このパターンを使う Client プログラム)が、逐次部品を投入したり設定情報を渡して Product を作るときに、その手順を Builder で一般化する
- Java の XMLライブラリの初期化手順は、どのライブラリ (Crimson、Xerces、GNU JAXP...) を使っても一緒 → BUILDERパターン

Director (Client)

Concrete b = new ConcreteBuilder(); b.createProduct(); b.addPart1(部品1); b.addPart2(部品2); AbstratProdut p = b.getResult();

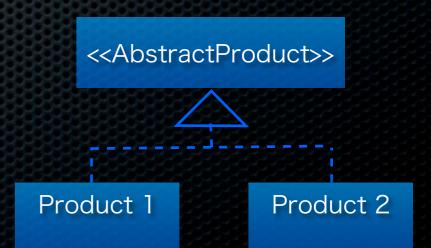
<<Builder>>

void createProduct()
void addPart1()
void addPart2()
AbstractProduct getResult()



ConcreteBuilder

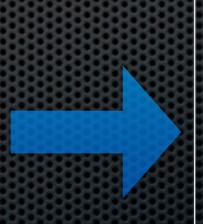
void createProduct()
void addPart1()
void addPart2()
AbstractProduct getResult()



21 [FACTORY METHOD]

- あるオブジェクトを作るときに、複雑な初期化手順が必要だったり、実行時に 作成するクラスを差し替えたいときに、クラスをインスタンス化するメソッドを 作成する。
- 19 [ABSTRACT FACTORY] の createProduct() メソッドは、まさに FACTORY METHOD。事実上 19 [ABSTRACT FACTORY] と 21 [FACTORY METHOD] は二つで一つのパターン

ComplexProduct p = new ComplexProduct();
// 初期化処理
p.addSomething();
p.setSomething();
...
// やっと使えるようになった
p.doSomething();



ComplexProduct p = Factory.createProduct(); p.doSomething();

Factory

public static ComplexProduct createProduct(){
 ComplexProduct p = new ComplexProduct();
 p.addSomething();
 p.setSomething();
}

22 [PROTOTYPE]

複雑な初期化処理が必要なオブジェクトは、毎回作 らずに、最初一つ作って、Client には、その Clone (複製) を使わせる。

Client

```
Prototype p = new Prototype();
```

```
Prototypep[] massProduct = new Prototype[100];
for( int cnt = 0 ; cnt < 100 ; cnt++){
  massProduct[cnt] = p.clone();
1</pre>
```

Prototype

Prototype clone()

複雑な初期化手順が必要 な内部データ構造

23 [SINGLETON]

- プロセス内で一意になるオブジェクト
- 厳密にプロセス内で一意になることは、期待しない方がいいかも・・・
 - たとえば、Java EE コンテナで、オブジェクトが Passivate/Active されると インスタンスが複数できる
 - コンストラクタを private にしていても reflectionでオブジェクトを作成可能

Client

Singleton.getInstance().doSomething();

Singleton

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

が多数