MVC

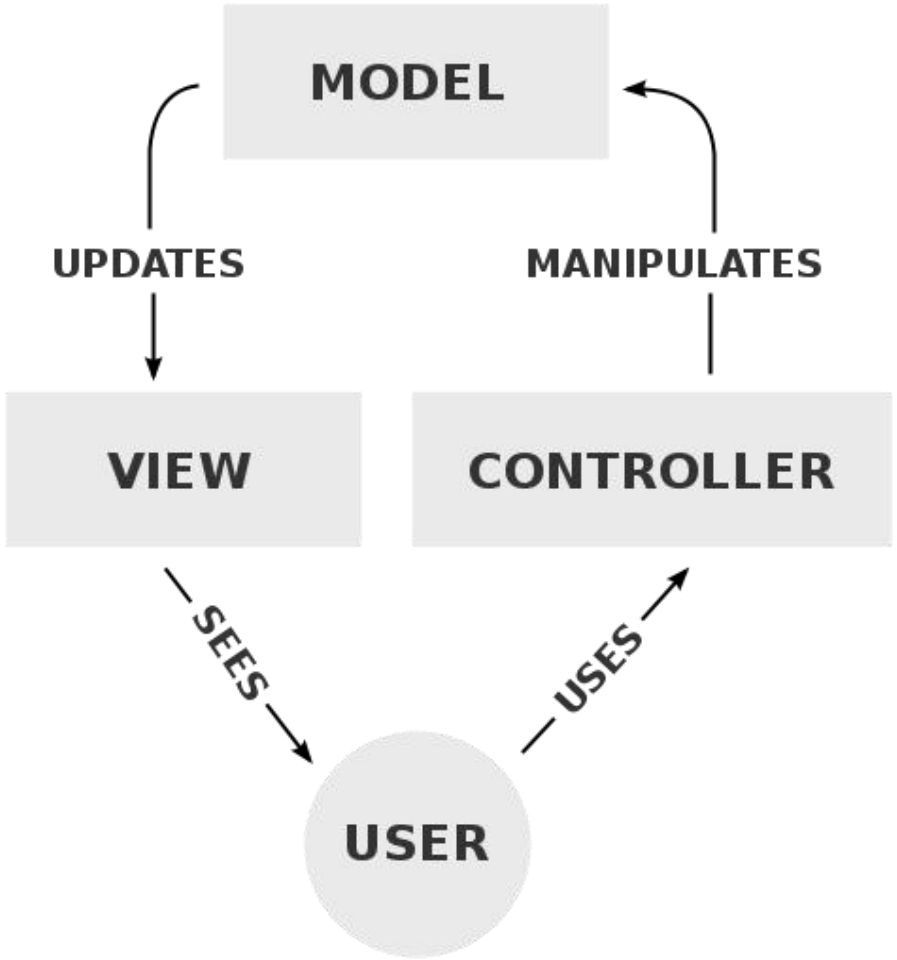
Model–View–Controller

簡單來說

View
就是使用者可以看到的東西

Controller
是當使用者透過View操作之後負責命令Model去做事情

Model
就是負責處理事情，處理完後再更新View到最新狀態



Q:當使用者透過View操作的時候Controller要怎麼知道呢?

A:透過ActionListener在View上註冊Controller的Listener監聽View的動作

↓Controller內已經實作好的Listenter

```
class AnswerListener implements ActionListener{

    @Override
    public void actionPerformed(ActionEvent e){
        String ans=ansView.getAns();
        user.checkAns(ans);

    }
}
```

↓透過View的方法註冊進去

```
public ResetPWController(ResetPW QView,DBMgr model,Authen a
    this.QView=QView;
    this.model=model;
    this.ansView=ansView;


    this.QView.addQbuttonListener(new QuestionListener());
    this.ansView.setbuttonListener(new AnswerListener());
}
```

↓View裡面的方法，實際上是註冊到Button

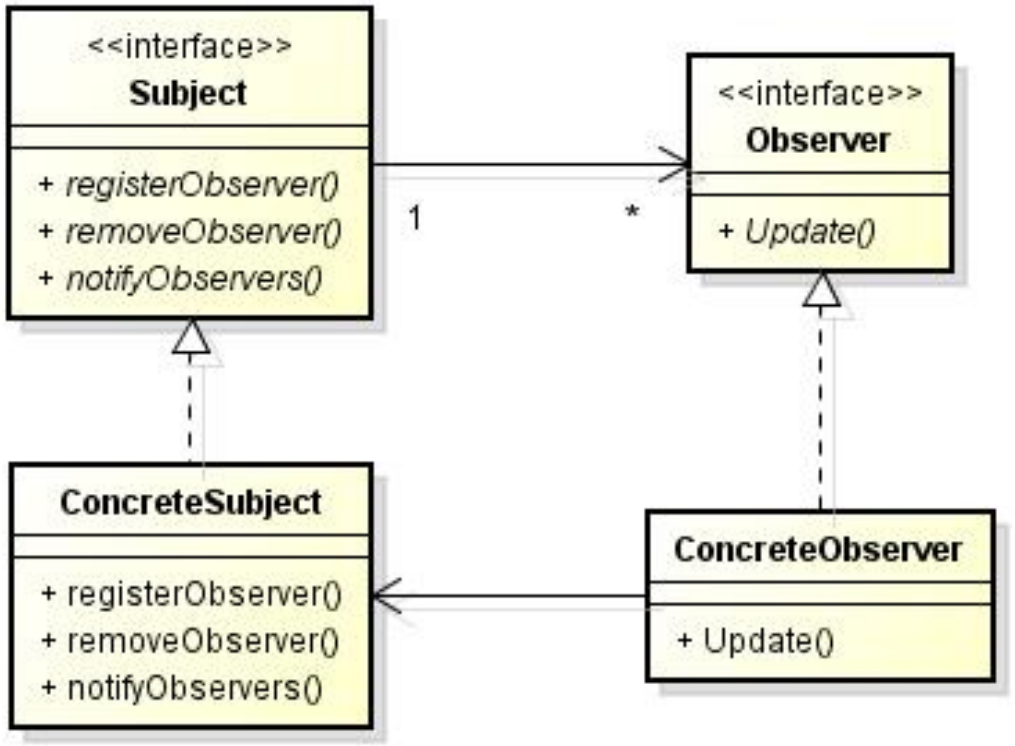
```
public void setbuttonListener(ActionListener listener){
    checkbutton.addActionListener(listener);
}
```

Q:Model要怎麼通知View要更新?

A:用Observer Pattern，model是Subject，View是observer

Subject(model)裡註冊observer(view)，更新時就會通知observer(view)需要更新

*****我給的MVC Code裡面有可以參考*****



↑ Observer Pattern參考圖

Liskov Substitution Principle

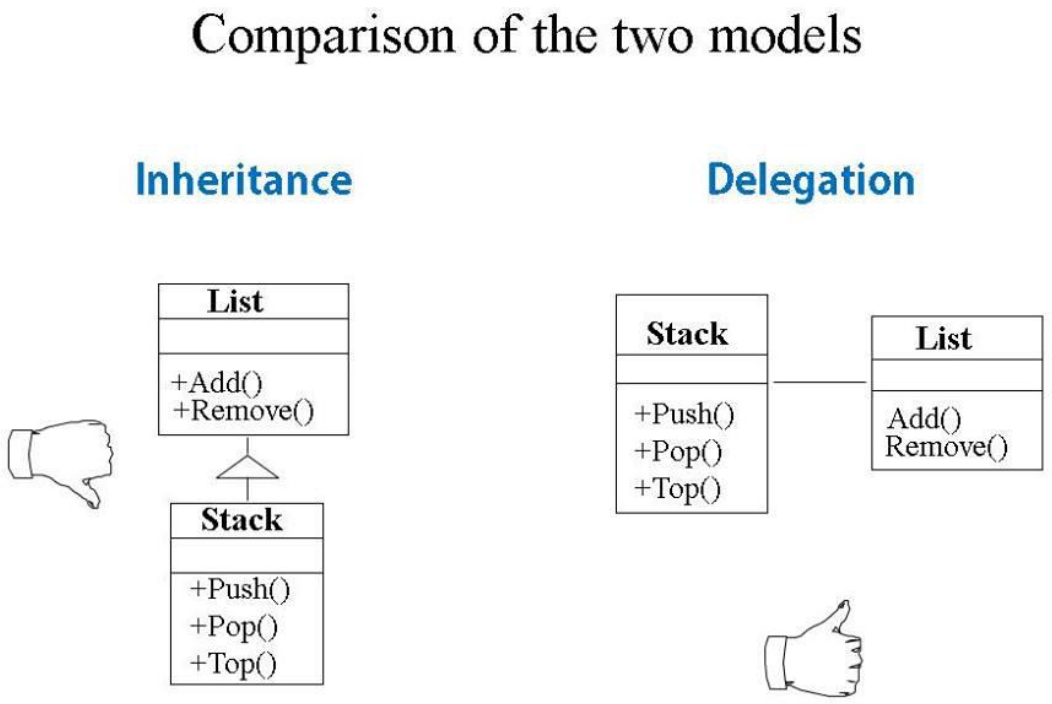
里氏替換原則

簡單說呢
就是子類別要能夠取代父類別
然後情況要合理

子類別不要繼承一個父類別是跟他毫無相關的

那為什麼右邊那張圖左邊的設計比右邊差呢?
因為左邊違反了LSP
List的方法跟Stack的方法毫不相關
因此Stack不能遵循父類別List的方法
所以這樣的設計不適合
如果硬要用，就用右邊的Aggregation的方式完成

*我給的Code裡面有右邊的範例，無聊可以看看

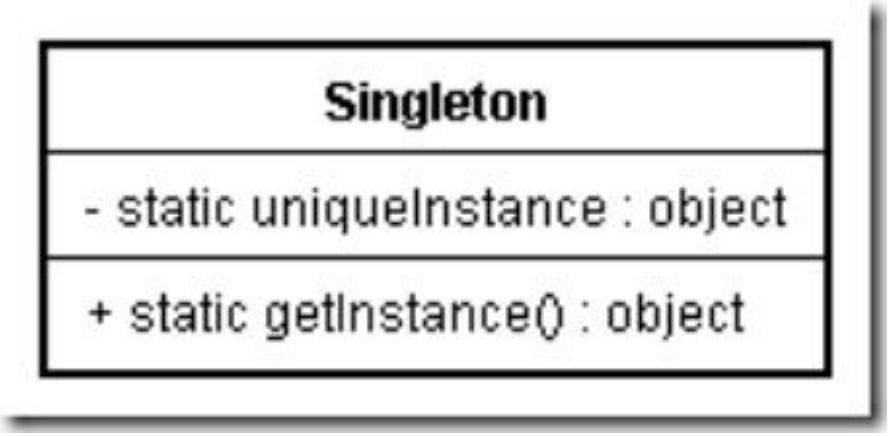


Singleton

單例模式

Singleton就是要確保物件只有一個實例
可以被重複使用

所以常常是被使用率極高(重複存取)的原件才會這樣做



Force：

- 你設計的類別代表了在真實世界中的特定情境之下，只會存在一份實例且具有狀態的「實體」、「服務」或「概念」，例如硬體的輸入/輸出埠、GPS、時鐘、印表機多工緩衝處理程式、檔案系統；而非在某種情境下，某個類別恰巧只需要產生一個實例。

- 這個單一實例要很容易地被其他物件存取。

--笑談軟工

實作方法

- Lazy
  - Synchronized
  - Double Checked Locking

  被需要的時候才建造實體

- Eager

  一開始就存在著實體

最基本的Lazy Singleton

```
class singleton{
    private static singleton obj=null;
    private singleton(){}
    private int data;
    public static singleton getInstance(){
        if (obj==null){
            obj=new singleton();
        }
        return obj;
    }
```

在取得實體時判斷是否已經建造
若為否則在此時建造實體

Problem：在大量(多人)存取時可能會造成問題產生多個實體

因此有以下三種Singleton方法

1. Synchronized

```
class singleton{
    private static singleton obj=null;
    private singleton(){}
    public static synchronized singleton getInstance(){
        if (obj==null){
            obj=new singleton();
        }
        return obj;
    }
}
```

使用Java的synchronized鎖住
當多重存取時會鎖住其他
當第一位完成才會進行下一位的動作

Problem：會有效率上的問題，太多存取時會塞車。

因此有以下三種Singleton方法

2. Double Checked Locking

```
class singleton{
    private static singleton obj=null;
    public static singleton getInstance(){
        if (obj==null){
            synchronized(singleton.class){
                if (obj==null){
                    obj=new singleton();
                }
            }
        }
        return obj;
    }
}
```

先判斷是否已經建造過實體

真的沒有實體時再進行第一次的建造實體

因此有以下三種Singleton方法

3. Eager

```
class singleton{
    private static singleton obj=new singleton();
    public static singleton getInstance(){
        return obj;
    }
```

在一開始就先建造好實例，反正都要反覆存取一定會用到

需要的時候直接回傳同一個

結論：Synchronized、Double Checked Locking、Eager都可以解決多重存取的問題。

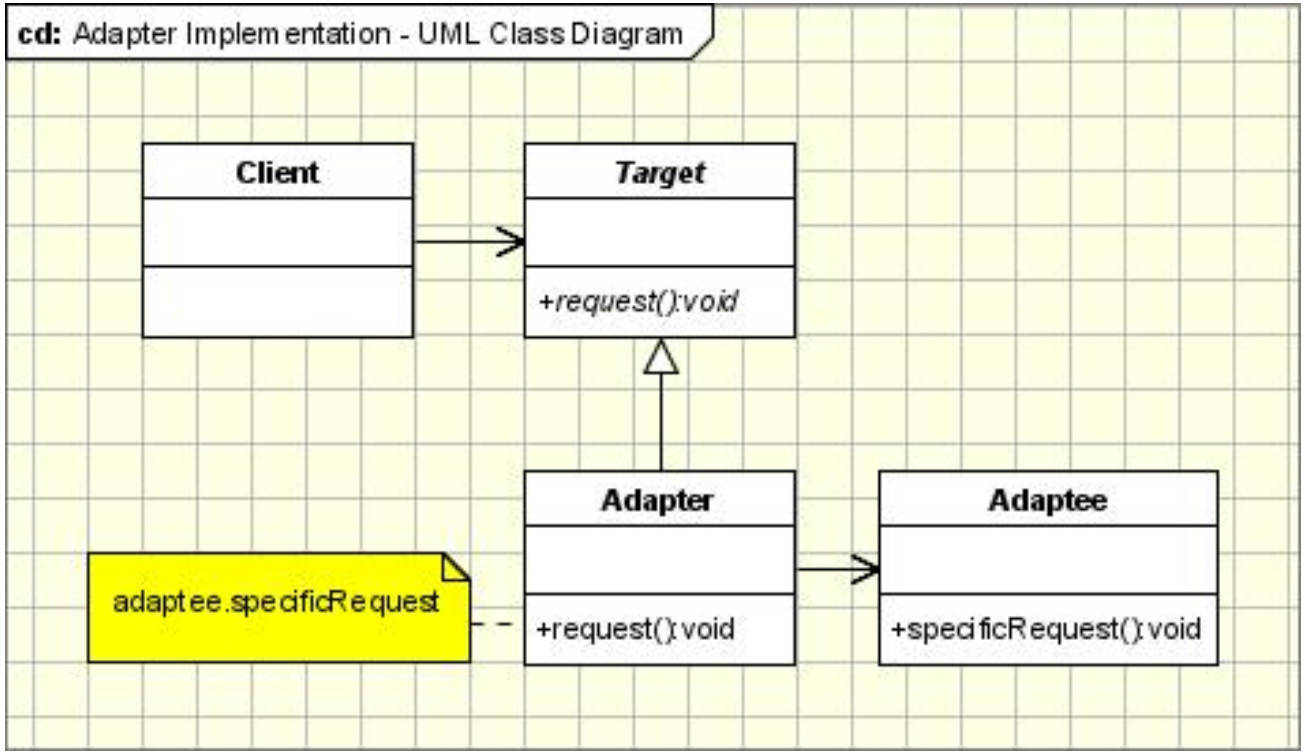
Adapter

轉接器模式

這個Pattern就是把舊有的物件(Adaptee)轉換成需要的物件(Target)
讓使用者可以使用舊有物件的功能
來達成新物件的功能

右圖是物件方式的Adapter
簡單來說就是透過aggregation
把Adaptee放在Adapter中
Adapter繼承Target

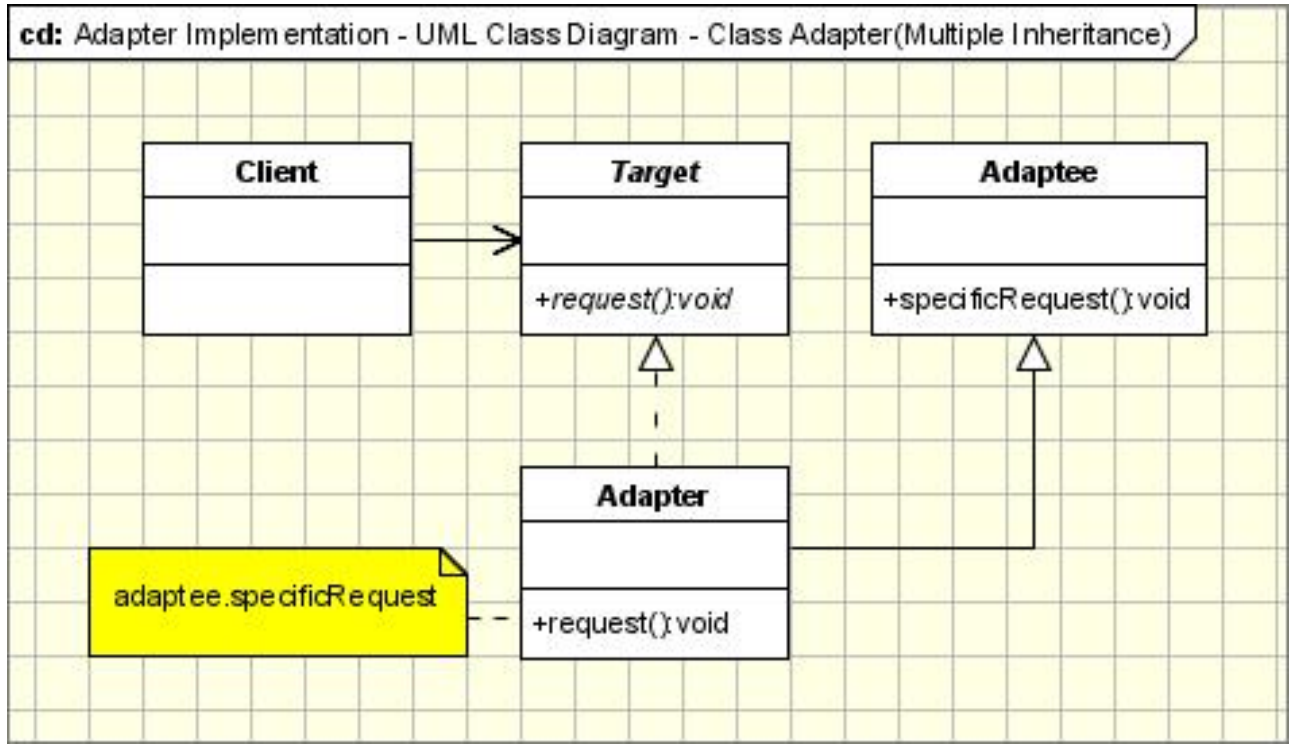
使用者就會認為Adater是Target
Adapter再用Adaptee的方法來做成
Target的方法



Class的方式其實跟Object差不多
只是換成了用多重繼承的方式

讓Adapter多繼承Adaptee
因此也可以使用Adaptee的方法
來做成Target的方法

Java不支援多重繼承故不討論



```java
class Target{
    public void request(){
        System.out.println("我是Target.");
    }
}

class Adaptee{
    public String SpecificRequest(){
        return "Adaptee";
    }
}

class Adapter extends Target{
    private Adaptee adaptee=new Adaptee();
    @Override
    public void request(){
        System.out.println("我是"+adaptee.SpecificRequest()+"裝成的Target.");

    }
}
```

Adapter透過Adaptee的方法實現Traget的Request方法

```java
public class AdapterDemo{
    public static void main(String[] arg){
        new Target().request();
        new Adapter().request();
    }
}
```

```
我是Target.
我是Adaptee裝成的Target.
```

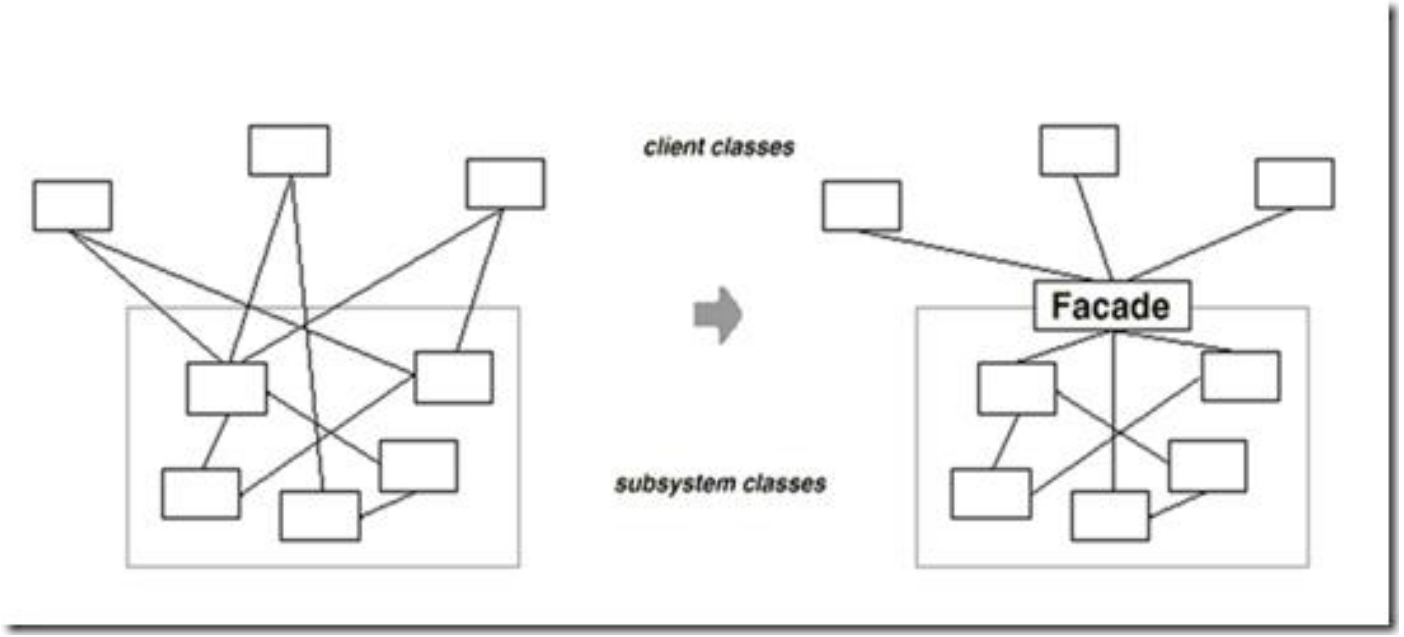
Facade

外觀模式

把原本客戶端跟子系統
間錯綜複雜的關係簡化

讓客戶端透過一個介面
就能夠使用所有的功能

客戶端也不會知道有多
少子系統在運作只會知道介面



```java
class Computer {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;
    private long BOOT_ADDRESS,BOOT_SECTOR;
    private int SECTOR_SIZE;
    public Computer(long address,long sector,int size){
        cpu=new CPU();
        hardDrive=new HardDrive();
        memory=new Memory();
        BOOT_ADDRESS=address;
        BOOT_SECTOR=sector;
        SECTOR_SIZE=size;
    }
    public void startComputer() {
        cpu.freeze();
        memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
        cpu.jump(BOOT_ADDRESS);
        cpu.execute();
    }
}
```

Façade介面裡面包含所有子系統

```java
class CPU {
    public void freeze() { System.out.println("CUP freeze."); }
    public void jump(long position) { System.out.println("CUP jump:"+positio
    public void execute() { System.out.println("CUP execute..."); }
}

class Memory {
    public void load(long position, byte[] data) {
        System.out.println("Meory load:"+position+" length:"+data.length);
    }
}

class HardDrive {
    public byte[] read(long lba, int size) {
        System.out.println("HardDrive read ...");
        return new byte[size];
    }
}
```

總共三個子系統

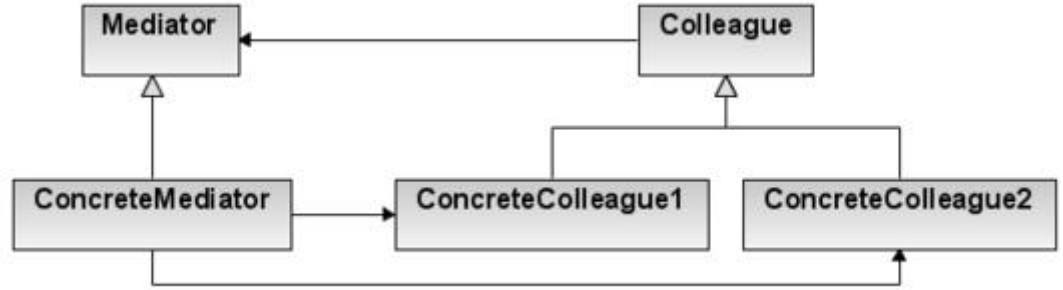
```java
public class You {
    public static void main(String[] args) {
        Computer facade = new Computer(0,10,10000);
        facade.startComputer();
    }
}
```

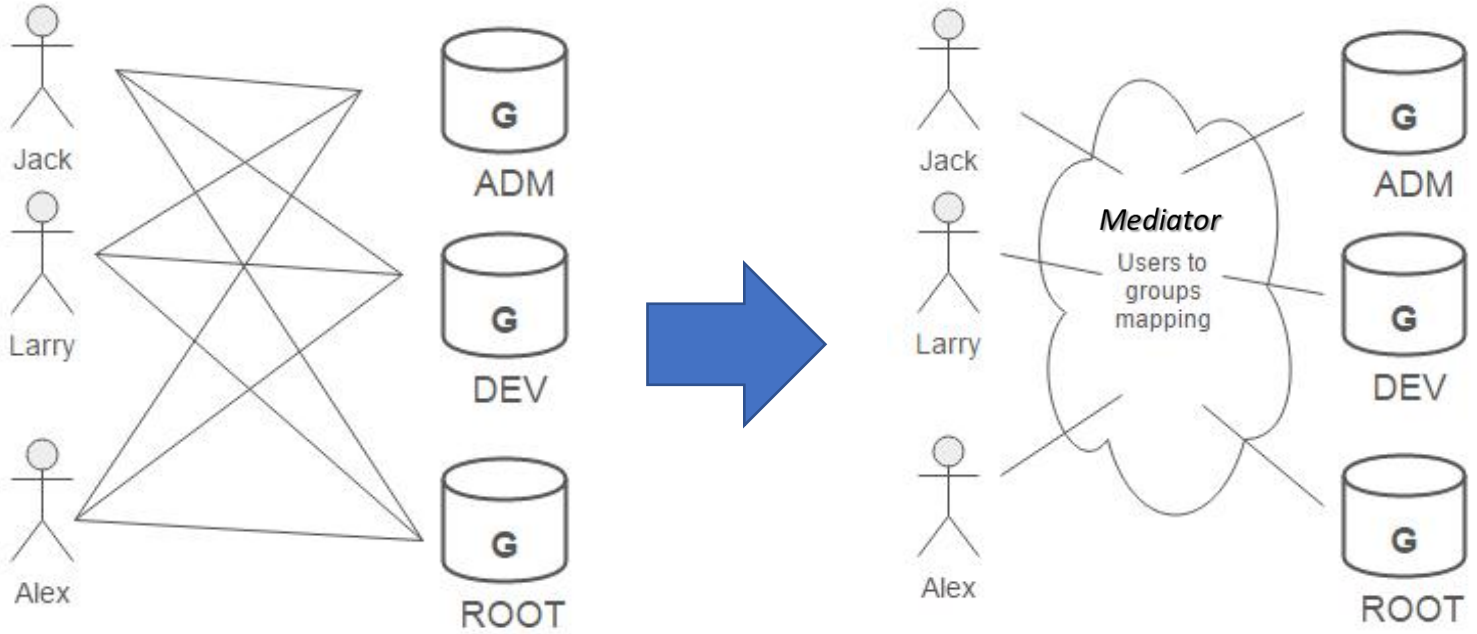
客戶端只會使用到Façade去操作

```
CUP freeze.
HardDrive read ...
Meory load:0 length:10000
CUP jump:0
CUP execute...
```

Mediator

中介者模式





把原本彼此錯亂的溝通透過一個中介者來管理

把直接的溝通交給中介者來轉達

Mediator和Colleague之間會互相知道

```java
//Abstract Mediator
interface Mediator {
    void book();
    void view();
    void search();
    void registerView(BtnView v);
    void registerSearch(BtnSearch s);
    void registerBook(BtnBook b);
    void registerDisplay(LblDisplay d);
}
```

Mediator算是集合了所有Colleague之間需要的功能

```java
//Concrete mediator
class ParticipantMediator implements Mediator {

    BtnView btnView;
    BtnSearch btnSearch;
    BtnBook btnBook;
    LblDisplay show;

    //....
```

Mediator知道所有的Colleague

```java
//A concrete colleague
class BtnView extends JButton implements Command {

    Mediator med;
```

Colleague也知道Mediator

- Colleague

```java
public void execute() {
    med.search();
}
```

- Mediator

```java
public void search() {
    btnSearch.setEnabled(false);
    btnView.setEnabled(true);
    btnBook.setEnabled(true);
    show.setText("searching...");
}
```

Colleague執行過程中如果需要其他Colleague幫忙則會呼叫Mediator去通知其他Colleague幫忙

Façade vs Mediator

Façade是希望提供一個介面對外提供操作

使用者不用知道任何底下的子系統運作

Mediator是當系統內部之間運作非常複雜

彼此的耦合太高的時候需要一個中介者來調節系統之間的運作

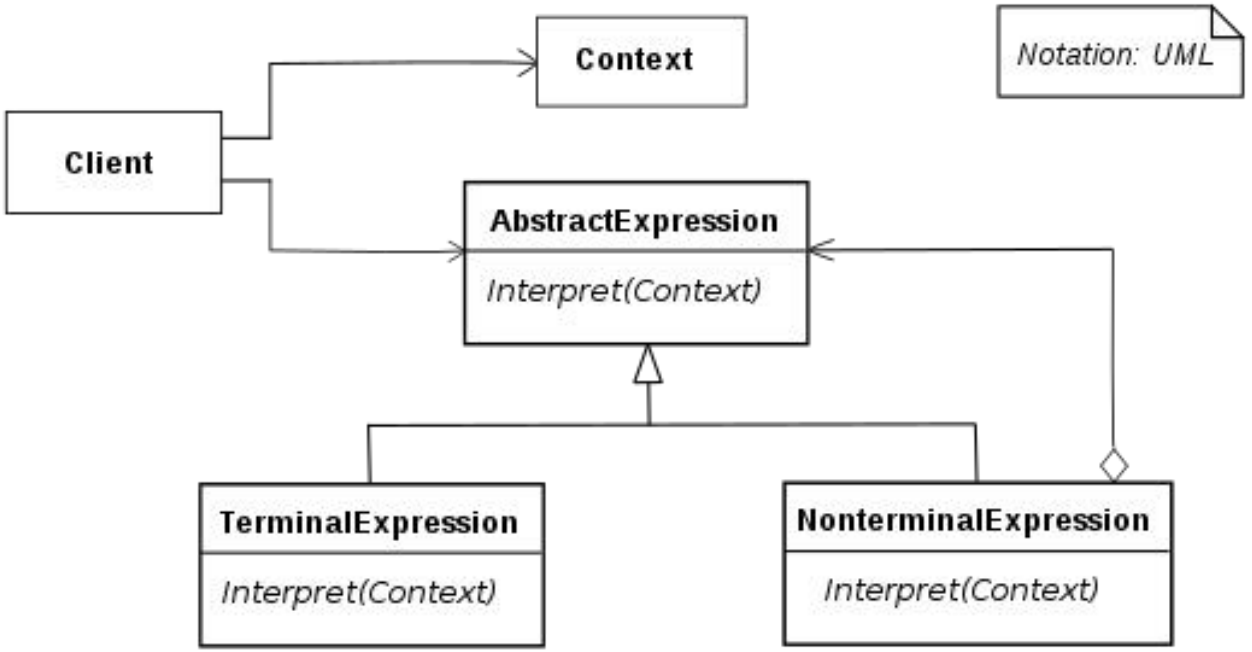
Interpreter

解釋器模式

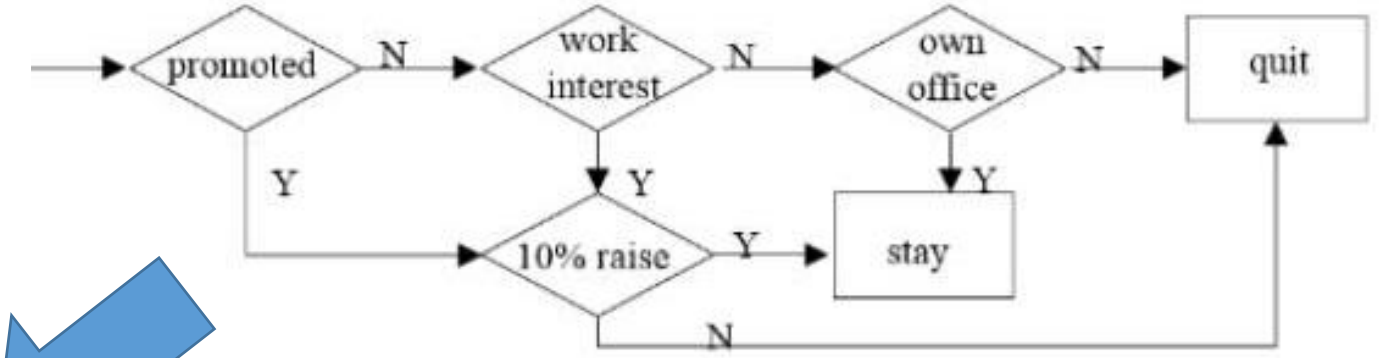
在處理一些複雜的問題上我們希望可以透過分析器把問題丟進去，答案會自然產生

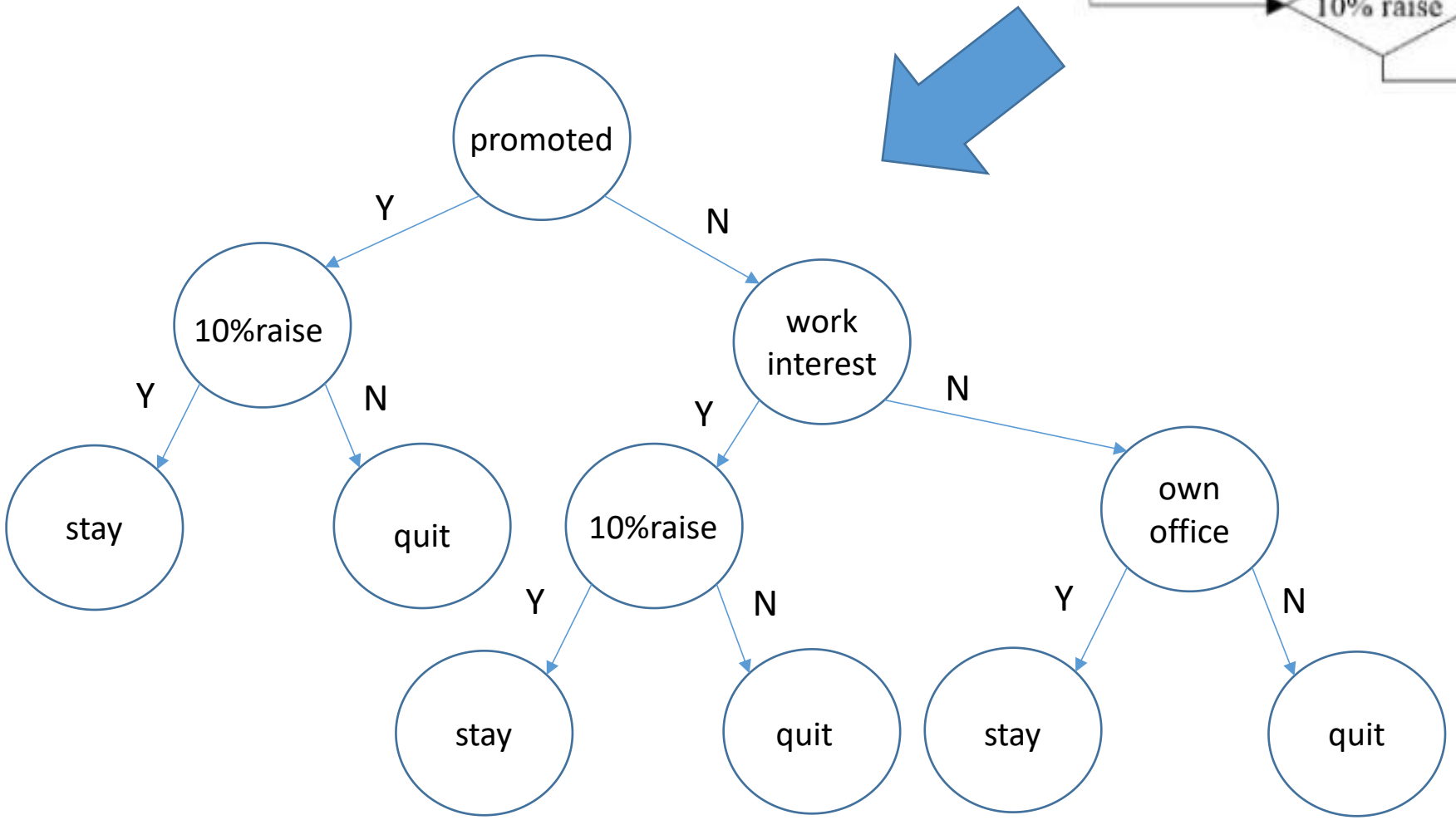
然而當問題變數改變時不必修改原本寫好的分析器

就會使用到Interpreter



我們試著為這個決策原則建立一棵樹
並使用Interpreter求出答案





轉換成程式怎麼寫？

1.看到的任何樹節點都是一個Experssion

```
interface Experssion{
    public boolean interpret(Map<String,String> context);
}
```

2.總共有六個情況 ，四個nonTerminal(promoted、raise、workinterest、ownoffice)，兩個Terminal(Stay、Q

```
class promoted implements Experssion{
    private String name="promoted";
    private Experssion y;
    private Experssion n;
    promoted(Experssion y,Experssion n){
        this.y=y;
        this.n=n;
    }
    public boolean interpret(Map<String,String> context){

        if (("Y").equals(context.get(name))){
            return y.interpret(context);
        }else{
            return n.interpret(context);
        }
    }
}
```

因為是nonTerminal所以會有左右節點(依情況而定，有時候可能是單邊節點)

解析的方法，根據此節點的結果分別繼續往左右節點走，直到Terminal節點得出結果。

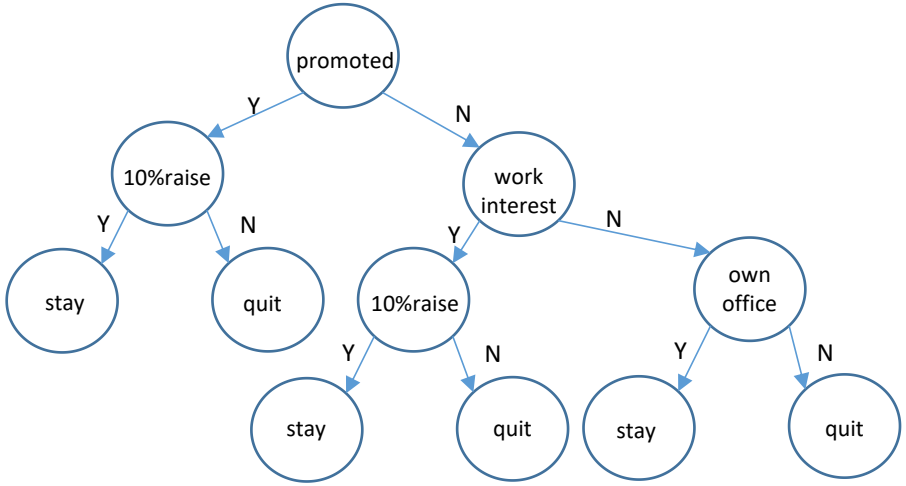
```
class stay implements Experssion{
    public boolean interpret(Map<String,String> context){
        return true;
    }
}
```

Terminal節點，輸出結果。

3.Interpreter最重要的地方，建立結構(樹)

```
class Evaluator{
    public Experssion evaluate(){
        Experssion r = new raise(new stay(),new quit());
        Experssion wi=new workinterest(new raise(new stay(),new quit()),new ownoffice(new stay(),new quit()));
        Experssion p=new promoted(r,wi);
        return p;
    }
}
```

根據樹的形狀建立整個結構



4.輸入與執行

```java
public class Interpreter{
    public static void main(String[] args){

        Evaluator evaluator=new Evaluator();
        Experssion Handle=evaluator.evaluate();
        Map<String,String> context=new HashMap<String,String>();
        Scanner scan=new Scanner(System.in);
        String s="";
        System.out.println("請輸入決策：(Ex:Y N - -) 輸入0結束");
        s=scan.nextLine();
        while((!s.equals("0"))){

        String[] s2=s.split(" ");
        context.put("promoted",s2[0]);
        context.put("raise",s2[1]);
        context.put("workinterest",s2[2]);
        context.put("ownoffice",s2[3]);
        result(Handle.interpret(context));
        s=scan.nextLine();
        }
    }
    public static void result(boolean b){
        if (b){
            System.out.println("Stay.");
        }else{
            System.out.println("Quit.");
        }
    }
}
```

Map分別用來存四個NonTerminal的情境抉擇狀況

輸入並用空格分割

將輸入分別填入Map中，並根據result結果輸出答案

```
請輸入決策：(Ex:Y N - -) 輸入0結束
N Y Y -
Stay.
Y Y - -
Stay.
```

Composite

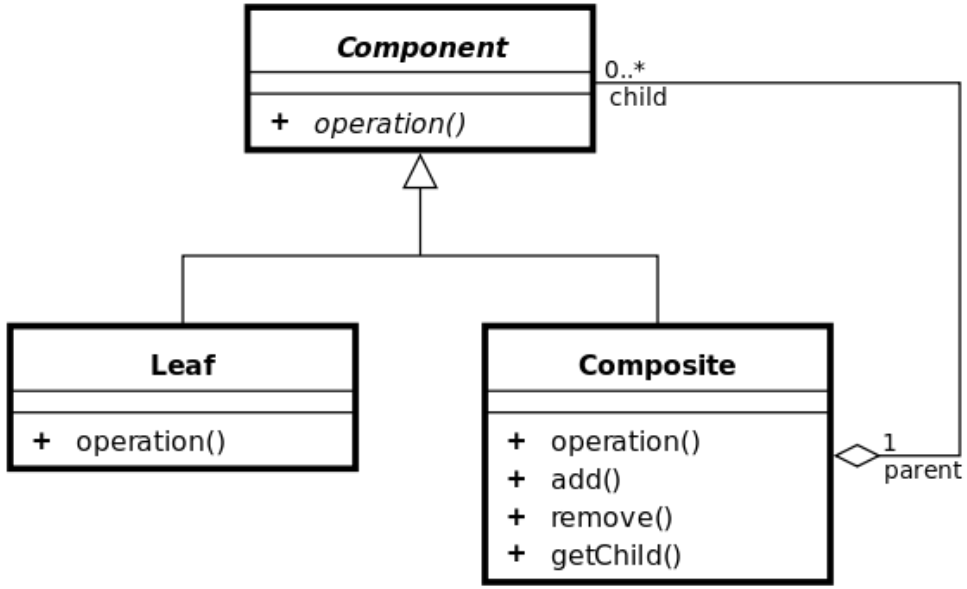
組合模式

Composite Pattern會將物件組織成樹狀結構

並且讓外界以一致性(都視為Component)的方式對待個別類別物件和組合類別物件。

個別類別 Leaf 和組合類別 Composite 都繼承 Component，透過這樣的繼承關係，才能使得巡訪時，可將個別類別物件及組合類別物件視為相同的類別。

- 階層關係(因為樹狀一層一層)

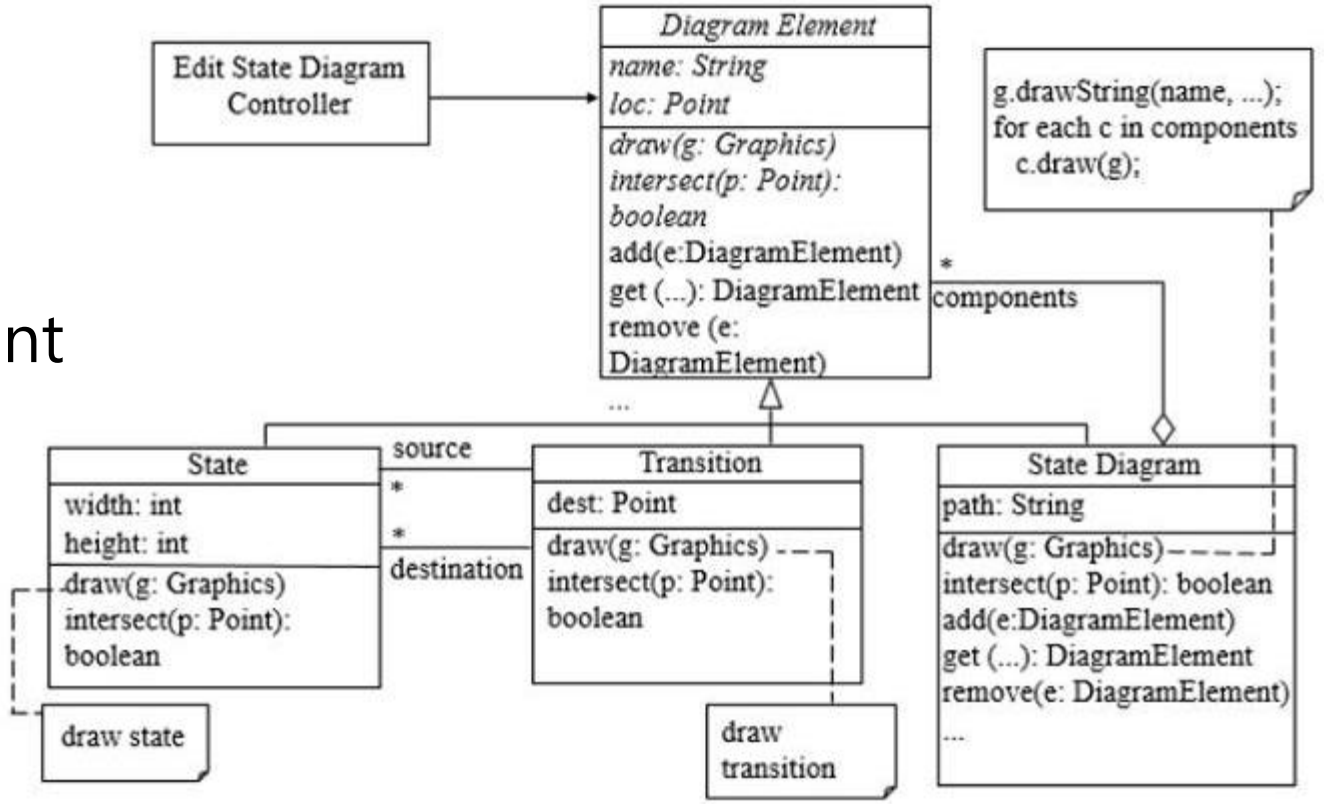


試著實作右邊的範例

Diagram Element 是Component

State、Transition是Leaf

State Diagram是Composite



1.建立抽象類別DiagramElement(Component)

```
abstract class DiagramElement{
    public String name;
    public Point loc;
    abstract void draw(Graphics g);
    public boolean intersect(Point p){
        return true;
    }
    public void add(DiagramElement e){
        System.out.println("I can't do that.");
    }
    public DiagramElement get(){
        System.out.println("I can't do that.");
        return null;
    }
    public void remove(DiagramElement e){
        System.out.println("I can't do that.");
    }
    public DiagramElement get(String name){
        System.out.println("I can't do that.");
        return null;
    }
}
```

事先實作好方法，防止Leaf使用
到這些功能時不會出現錯誤
也可以直接實作成空的方法→{}

2.建立State、Transition(Leaf)

```
class State extends DiagramElement{
    public void draw(Graphics g){
        //draw State
        System.out.println("Draw a State.");
    }
    public boolean intersect(Point p){
        //do State instrsect decide
        //we presume it is true
        return true;
    }

}
class Transition extends DiagramElement{
    public void draw(Graphics g){
        //draw Transition
        System.out.println("Draw a Transition.");
    }
    public boolean instrsect(Point p){
        //do Transition instrsect decide
        //we presume it is true
        return true;
    }

}
```

沒什麼好講的很簡單~

3.建立StateDiagram(Composite)

```
class StateDiagram extends DiagramElement{
    public String Path;
    private ArrayList<DiagramElement> des=new ArrayList<DiagramElement>();
    public void draw(Graphics g){
        //g.drawString(name,....);
        Iterator<DiagramElement> itr = des.iterator();
        while (itr.hasNext()){
            DiagramElement e=itr.next();
            e.draw(g);
        }
    }
    public boolean instrsect(Point p){
        //do StateDiagram instrsect decide
        //we presume it is true
        return true;
    }
    public void add(DiagramElement e){
        des.add(e);
    }
    public void remove(DiagramElement e){
        des.remove(e);
    }
    public DiagramElement get(String name){
        Iterator<DiagramElement> itr = des.iterator();
        DiagramElement oute=null;
        boolean checknull=true;
        while (itr.hasNext()){
            DiagramElement e=itr.next();
            if (name.equals(e.name)){
                oute=e;
                checknull=false;
            }
        }
        if ( checknull){
                System.out.println("Can't find this Element.");
        }
        return oute;
    }
}
```

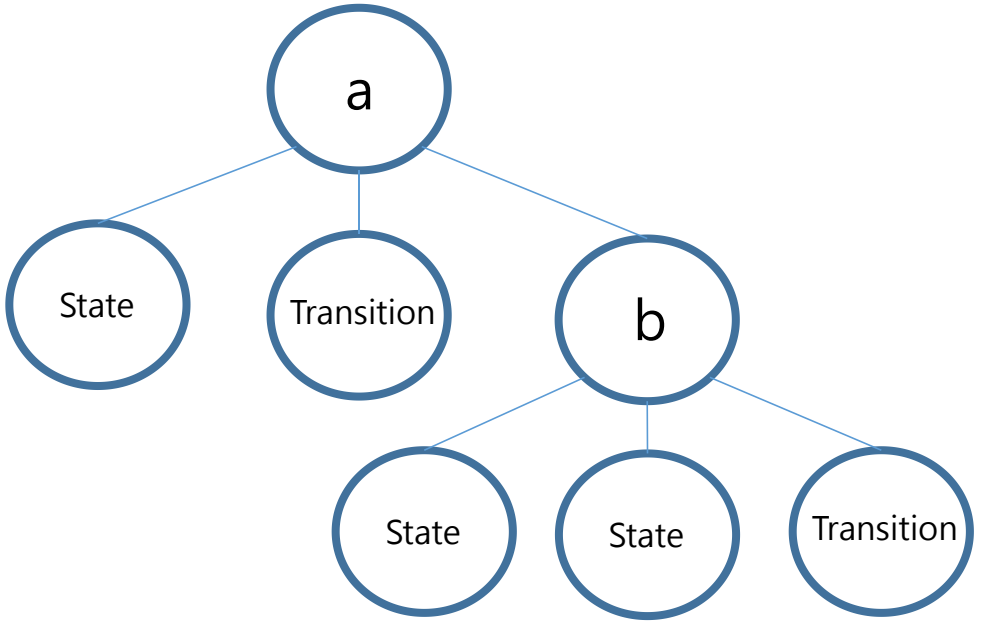
Composite是組合，所以需要一個ArrayList來儲存多個元件
因為Composite、Leaf本身都繼承了Component，都可以視為Component

因此ArrayList宣告儲存的型態只要宣告為Component類別就可以同時儲存Composite及Leaf

只是要注意如果使用到父類別未定一的方法會出錯(這時候需要轉型)

```java
public class EditStateDiagramController {
    public static void main(String[] args){
        DiagramElement b=new StateDiagram();
        b.add(new State());
        b.add(new State());
        b.add(new Transition());
        DiagramElement a=new StateDiagram();
        a.add(new State());
        a.add(new Transition());
        a.add(b);
        a.draw(null);
    }
}
```

```
Draw a State.
Draw a Transition.
Draw a State.
Draw a State.
Draw a Transition.
```



都好了之後就可以很簡單的建立一顆樹

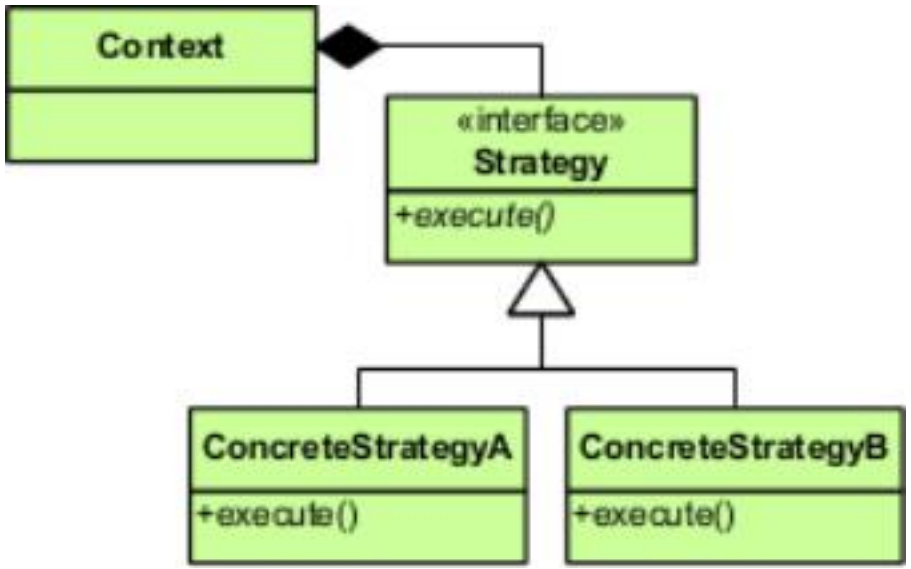
Strategy

策略模式

為了達到相同的目的，物件可以因地制宜，讓行為擁有多種不同的實作方法。

比如每個人都要「交個人所得稅」，但是「在美國交個人所得稅」和「在中國交個人所得稅」就有不同的算稅方法。

將實作的方法從原本的物件獨立出來成為一個Class，透過繼承的方式可以在不修改原本程式碼的情況下輕易地擴充新的程式碼



- 吻合OCP原則(Open-Closed Principl)
- Runtime下改變(Override)
- Dynamic Binding

OCP=軟體實體必須能夠延伸但不能修改，" 對擴展開放，修改則封閉"

```java
interface Strategy{
    public void execute();
}
```

1.先宣告一個Interface(或abstract Class)

```java
class StrategyA implements Strategy{
    public void execute(){
        System.out.println("Using StrategyA.");
    }
}
class StrategyB implements Strategy{
    public void execute(){
        System.out.println("Using StrategyB.");
    }
}
```

2.再讓實際運作的方法(演算法)實現

```
class Context{
    Strategy strategy;
    public void setStrategy(Strategy s){
        strategy=s;
    }
    public void execute(){
        strategy.execute();
    }
}
public class StrategyDemo{
    public static void main(String[] arg){
        Context c = new Context();
        c.setStrategy(new StrategyA());
        c.execute();
        c.setStrategy(new StrategyB());
        c.execute();
    }
}
```

3.在需要切換運作方法的物件內
建立Strategy物件

4.隨時可以切換Strategy來改變實際的方法

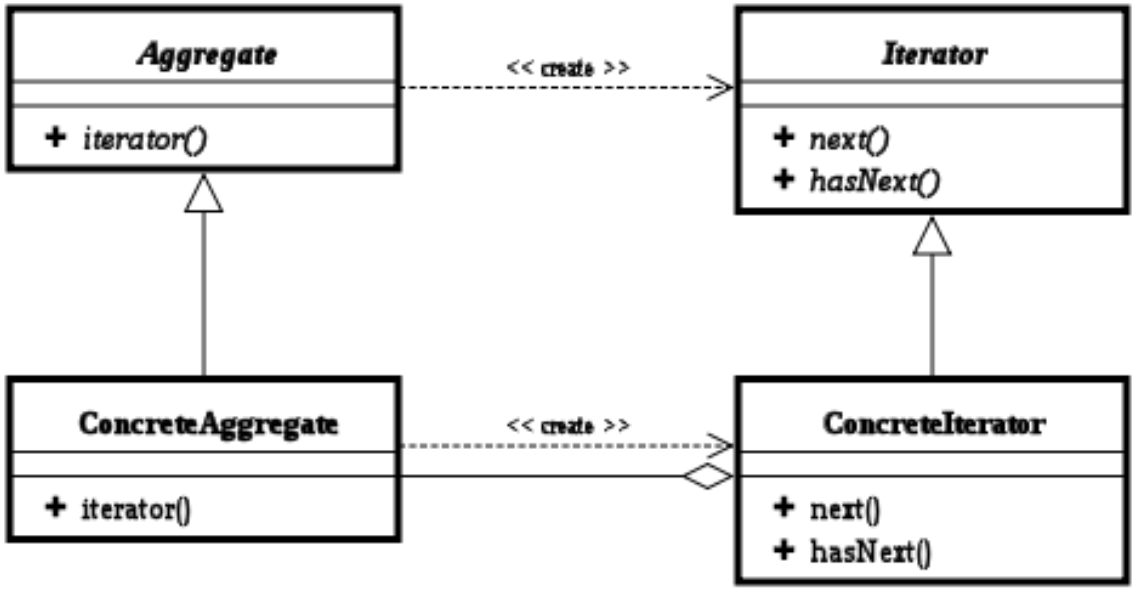
5.當Context物件被呼叫execute時
真正去執行的是Strategy所以當Strategy
被改變時，運作的方法也就不一樣了

Iterator

走訪器模式

提供方法走訪集合內的物件

走訪過程不需知道集合內部的結構



```
Iterator<DiagramElement> itr = des.iterator();
while (itr.hasNext()){
    DiagramElement e=itr.next();
    e.draw(g);
}
```

Java的Collection物件都有內建iterator()方法
直接拿來用吧..

Visitor

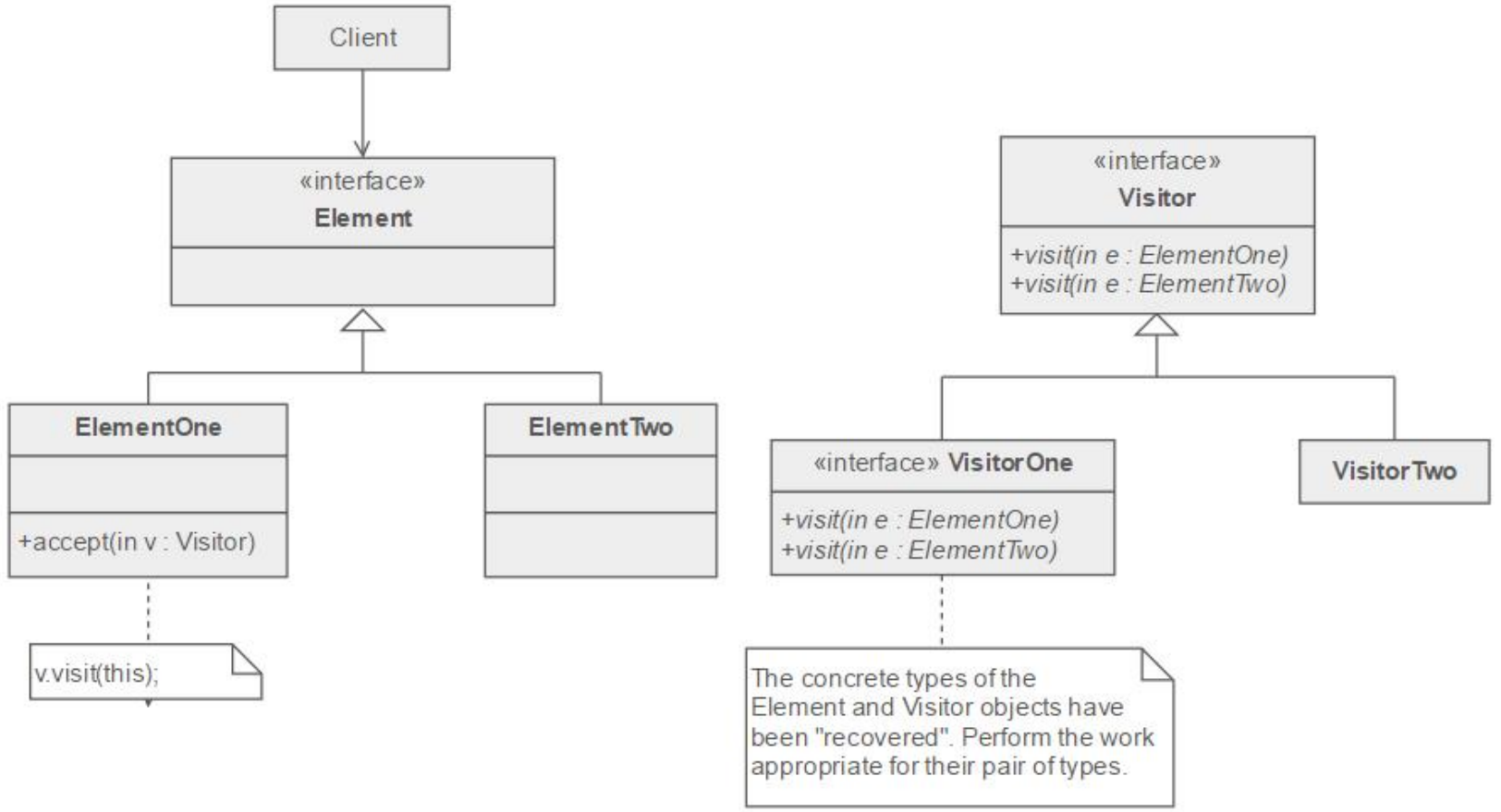
拜訪者模式

當你有很多元件(element)且數量固定

而這些元件常常需要被執行某些操作

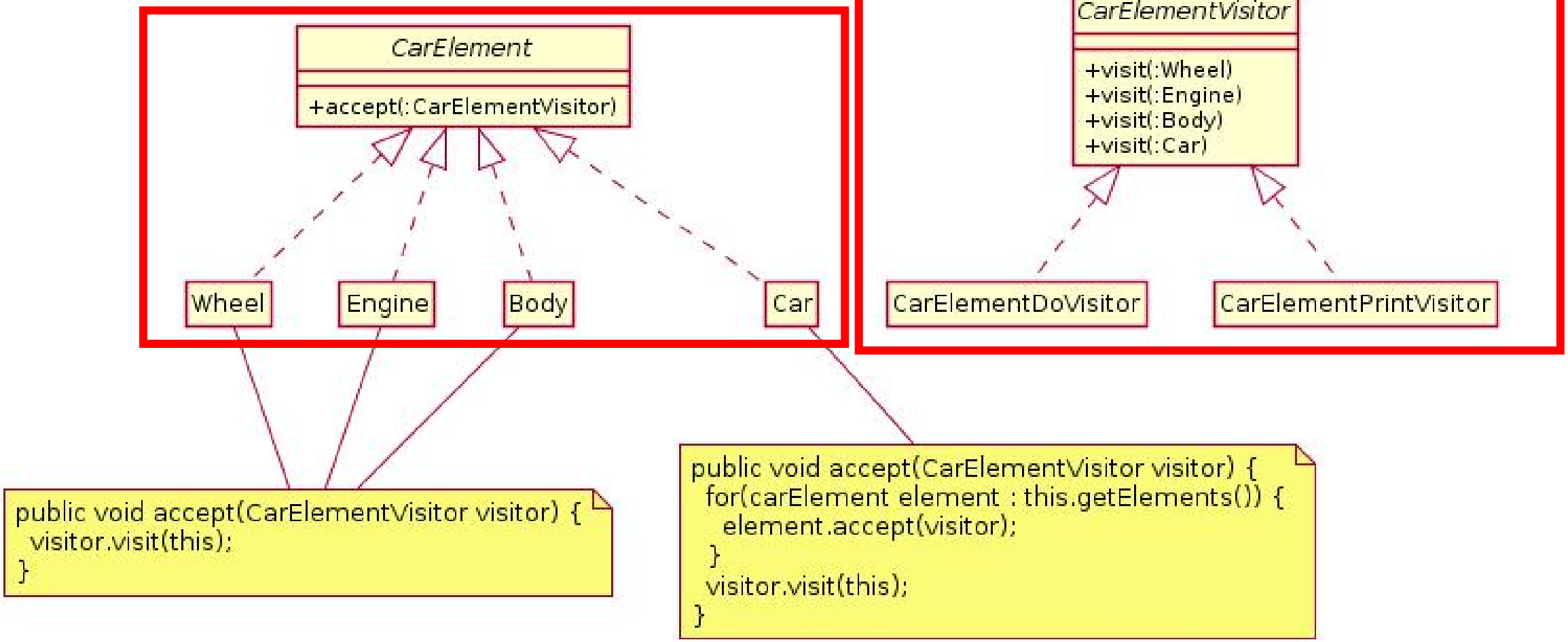
就可以使用Visitor

透過訪問者的方式來對這些元件進行操作



Override

Overload



```
interface ICarElement {
    void accept(final ICarElementVisitor visitor);
}

interface ICarElementVisitor {
    void visit(final Body body);
    void visit(final Car car);
    void visit(final Engine engine);
    void visit(final Wheel wheel);
}
```

先把Element跟Visitor的方法定義好

Element只有accept的方法(Override)

Visitor則要根據有幾個Element就會
有幾個Visit方法(Overload)

```
Body body=new Body();
body.accept(new CarElementPrintVisitor());
```

```
class Body implements ICarElement {
    public void accept(final ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

在告訴Element 來訪的Visitor (accept方法)之後

Element會把自己傳給Visitor(visite方法)

第一次Dispatch

```
class CarElementDoVisitor implements ICarElementVisitor {
    public void visit(final Body body) {
        System.out.println("Moving my body");
    }
```

最後Visitor會根據收到的是哪一個element
做出相對應的結果

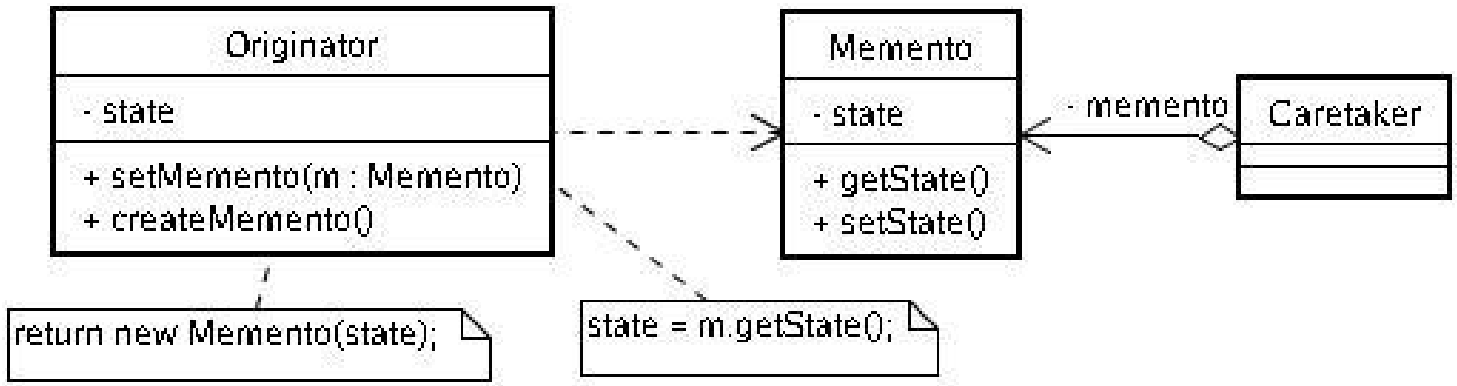
第二次Dispatch

Memento

備忘錄模式

Memento就是用來備份資料以供日後還原

- Originator負責建造Memento
- Memento負責儲存需要備份的東西
- Caretaker負責儲存這些備份下來的Memento



```java
class Memento {
    private String state;

    public Memento(String stateToSave) { state = stateToSave; }
    public String getSavedState() { return state; }
}
```

Memento 儲存state
可以用getSavedState()方法取出儲存的State

```java
class Originator {
    private String state;
    /* lots of memory consumptive private data that is not necessary to define the
     * state and should thus not be saved. Hence the small memento object. */

    public void set(String state) {
        System.out.println("Originator: Setting state to "+state);
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }
    public void restoreFromMemento(Memento m) {
        state = m.getSavedState();
        System.out.println("Originator: State after restoring from Memento: "+state);
    }
}
```

放置State的地方

提供Set方法可以改變State

需要備份時建造一個新的Memento

還原的時候從Memento取出State並還原

```java
class Caretaker {
    private ArrayList<Memento> savedStates = new ArrayList<Memento>();

    public void addMemento(Memento m) { savedStates.add(m); }
    public Memento getMemento(int index) { return savedStates.get(index); }
}
```

利用ArrayList儲存Memento
AddMemento增加備份
需要的時候可以用Index來 getMemento

```java
public class MementoExample {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State3");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State4");

        originator.restoreFromMemento( caretaker.getMemento(1) );
    }
}
```

Originator.saveToMemento()回傳一個新的Memento備份物件後丟到Caretaker裡面保存

需要再從Caretaker裡面取出來還原

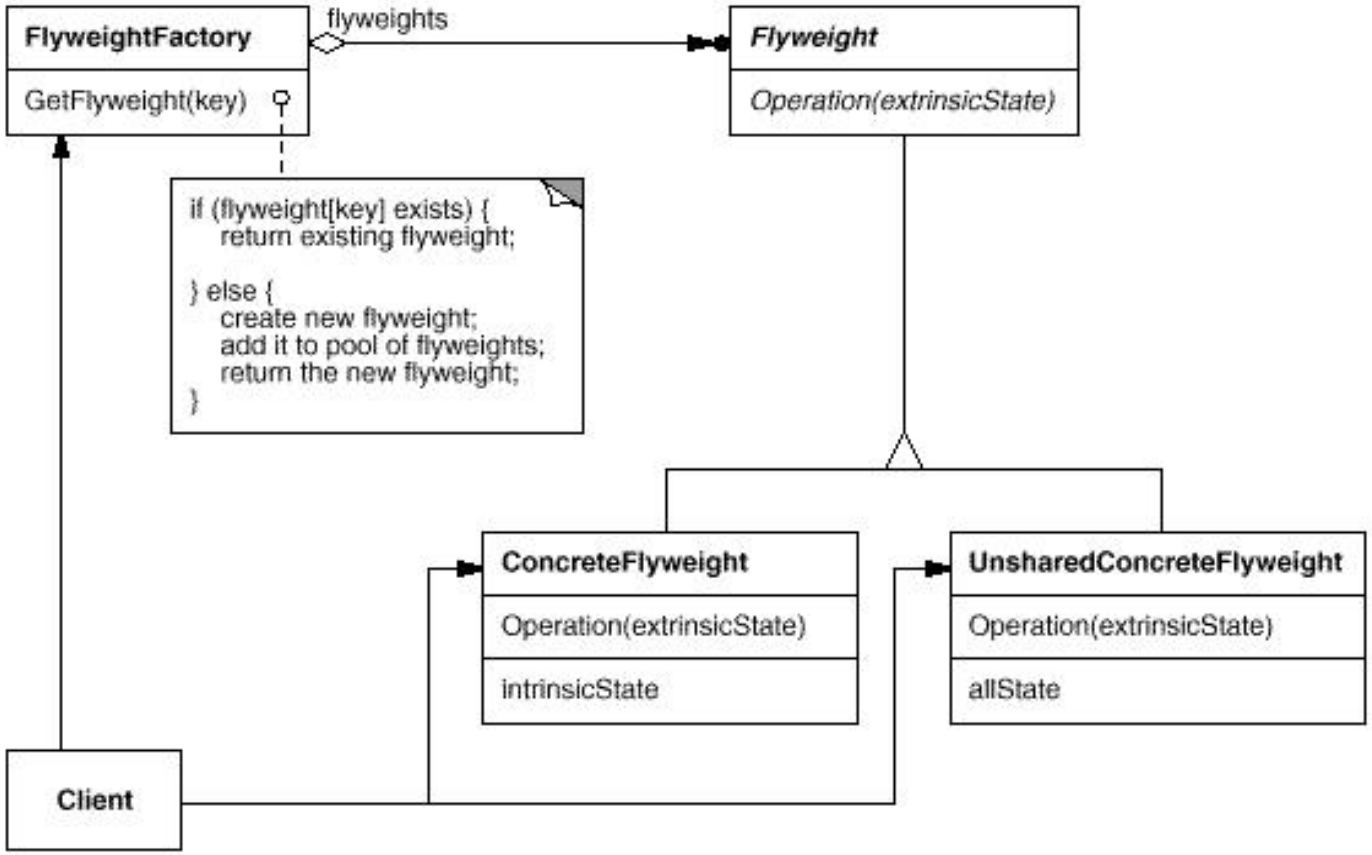
```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

Flyweight

享元模式

共享物件，用來儘可能減少記憶體使用量以及分享資訊給儘可能多的相似物件。

- Flyweight：所有的具體享元類的父類別，為這些類規定出需要實現的公共接口。
- ConcreteFlyweight：實現Flyweight接口，並為內部狀態拉回存儲空間。(也可以有不被共享的Flyweight)
- FlyweightFactory：負責創建和管理享元角色。
- Client：需要存儲所有享元對象的外部狀態。



Intrinsic:可被共享的 Extrinsic:不被共享的

```java
interface Flyweight
{
    public void operation( String extrinsicState );
}


class ConcreteFlyweight implements Flyweight {
    private String intrinsicState;
    public ConcreteFlyweight(String state){
        intrinsicState=state;
    }
    public void operation( String extrinsicState )
    {
        System.out.println(extrinsicState+" "+intrinsicState);
    }
}
```

儲存在內部的intrinsic，在使用得時候就可以共享出去

不被共享的在使用時才取得

```
class FlyweightFactory {
    private Hashtable flyweights = new Hashtable();
    public Flyweight getFlyweight( String key ) {
        Flyweight flyweight = (Flyweight) flyweights.get(key);

        if( flyweight == null ) {
            flyweight = new ConcreteFlyweight(key);
            flyweights.put( key, flyweight );
        }

        return flyweight;
    }
}
```

Factory使用Hashtable(或HashMap)
來存放建造出來的Flyweight物件

在要取得Flyweight時先從
HashTable裡面找
如果找不到再建造一個新的

如此一來才能減少記憶體空間的使用

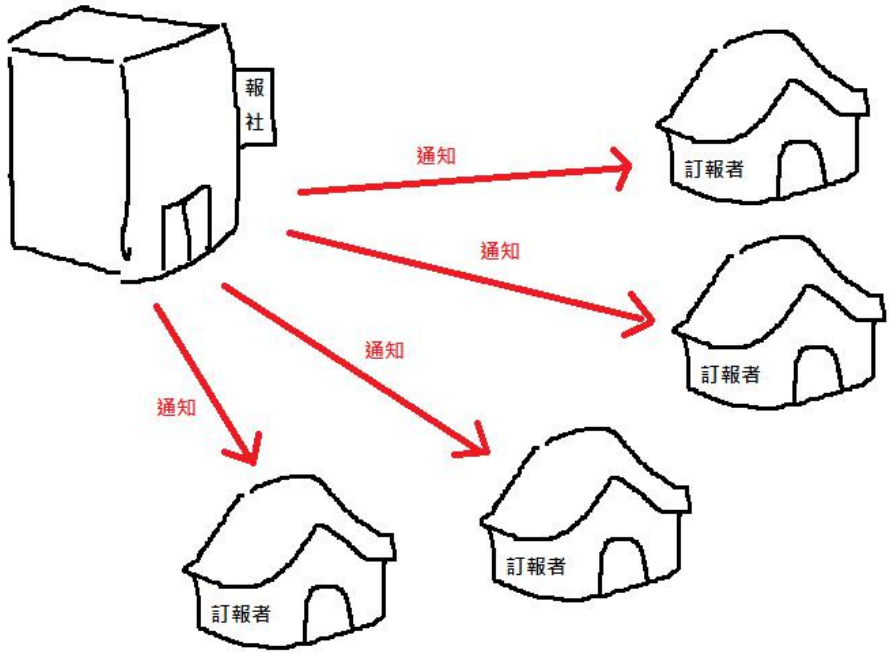
```
FlyweightFactory factory=new FlyweightFactory();

Flyweight flyweight=factory.getFlyweight("A");
flyweight.operation("red");
```

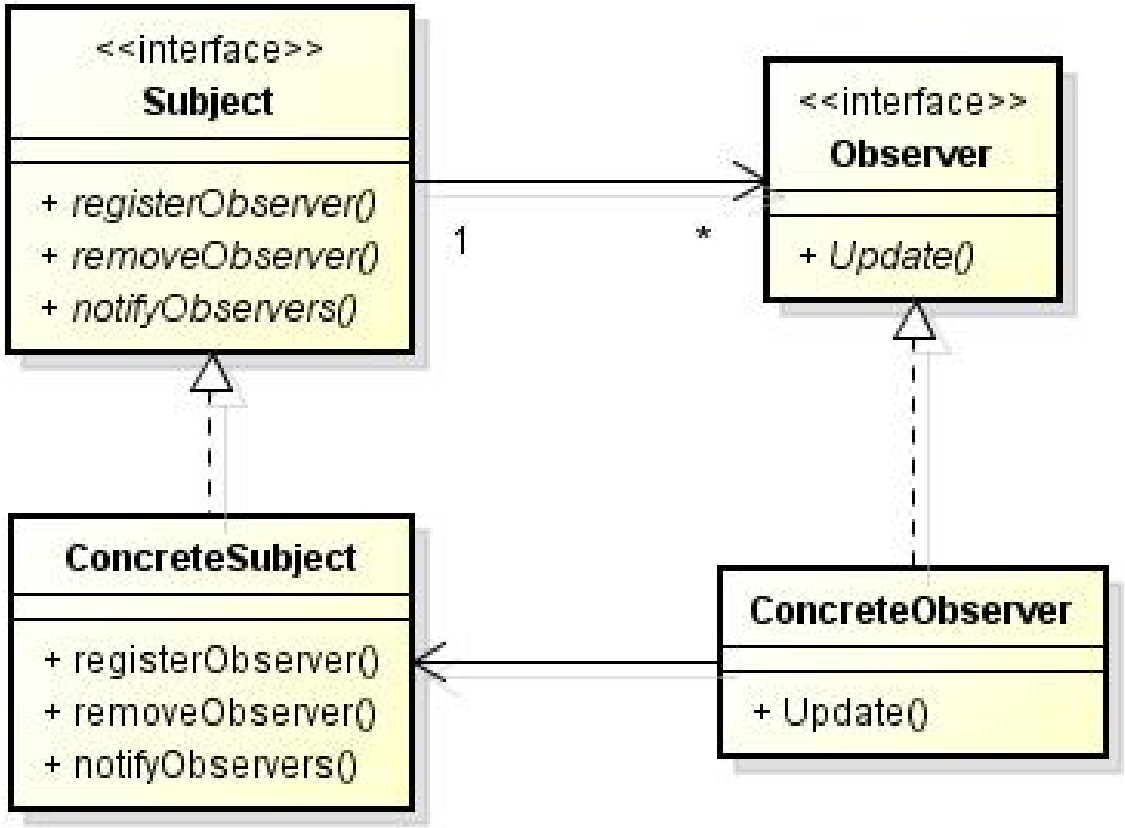
```
red A
```

Observer

觀察者模式

Observer很簡單
就是當Subject改變時
要通知所有的
Observer要更新了~~





```java
interface Subject{
    void notifyAllobserver();
    void register(Observer o);
    void remove(Observer o);

}
interface Observer{
    public void update(String news);
}
```

先把Subject、Observer的方法定義出來

```java
class newspaperOffice implements Subject{
    private String news="";
    private ArrayList<Observer> observers=new ArrayList<>();
    public void register(Observer o){
        observers.add(o);
    }
    public void remove(Observer o){
        observers.remove(o);
    }
    public void notifyAllobserver(){
        for (Observer o:observers){
            o.update(news);
        }
    }
    public void updateNews(String news){
        this.news=news;
        System.out.println("Office updateNews:"+news);
        notifyAllobserver();
    }
}
```

用一個ArrayList來儲存有訂閱這個Subject的訂閱者(Observer)

註冊新的訂閱者(Observer)
或
刪除原有的訂閱者(Observer)

用一個迴圈跑過所有ArrayList裡面的訂閱者(Observer)
並把新的新聞通知她們更新

更新新聞
同時通知所有訂閱者更新

```java
class subscriber implements Observer{
    private String name;
    public subscriber(String name){
        this.name=name;
    }
    public void update(String news){
        System.out.println("I'm "+name+", I got news:"+news);
    }
}
```

收到更新時print出來

```java
newspaperOffice office=new newspaperOffice();
office.register(new subscriber("John"));
office.register(new subscriber("Hsuan"));
office.updateNews("葛仲珊揪20多人《大夥騎》 俏皮腳開開");
office.updateNews("衰！歐陽妮妮又傳摔傷十字韌帶斷裂將開刀");
```

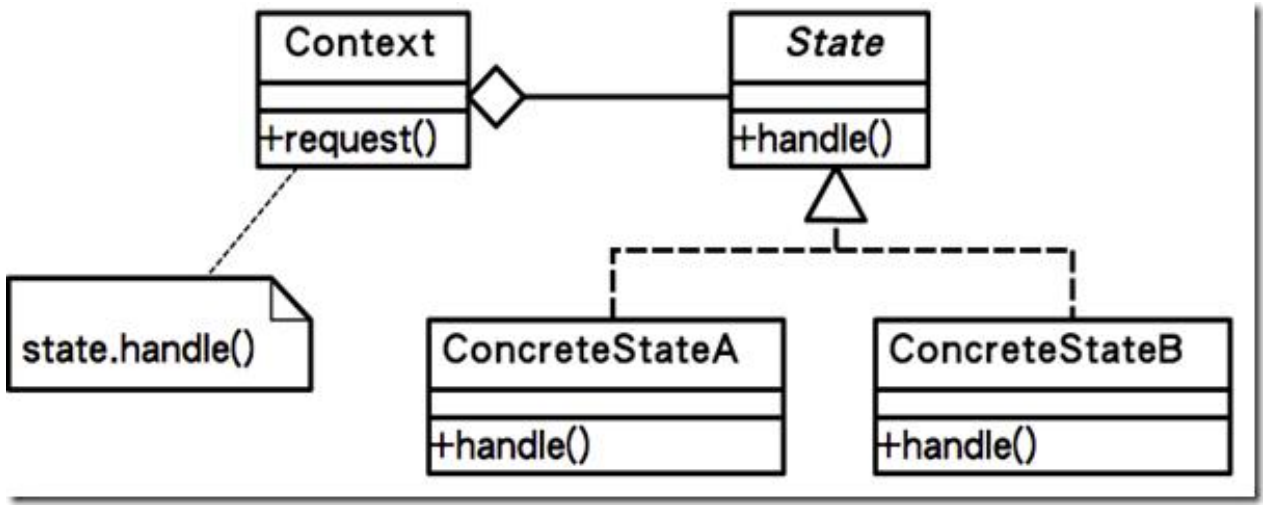
```
Office updateNews:葛仲珊揪20多人《大夥騎》 俏皮腳開開
I'm John, I got news:葛仲珊揪20多人《大夥騎》 俏皮腳開開
I'm Hsuan, I got news:葛仲珊揪20多人《大夥騎》 俏皮腳開開
Office updateNews:衰！歐陽妮妮又傳摔傷十字韌帶斷裂將開刀
I'm John, I got news:衰！歐陽妮妮又傳摔傷十字韌帶斷裂將開刀
I'm Hsuan, I got news:衰！歐陽妮妮又傳摔傷十字韌帶斷裂將開刀
```

只要報社更新新聞就會通知
所有訂閱者收到更新

State

狀態模式

一個物件的行為會因為物件自身狀態不同而表現出不同的反應動作。例如一個自動販賣機物件，具備「選擇貨品」的功能。同樣一個「選擇貨品」功能，會因為顧客有沒有投錢、投了多少錢，而有著不同的行為反應。



結構跟Strategy一模一樣
不過目的不一樣

- State是由自己轉變到下一個State
- Strategy是由使用者決定要切換到哪一個方法

```
interface State {
    void change(TrafficLight light);
}
```

這裡用的是一個紅綠燈的範例
會自動切換紅、綠、黃三個狀態

```
abstract class Light implements State {
    public abstract void change(TrafficLight light);
    protected void sleep(int second) {
        try {
            Thread.sleep(second);
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

因為號誌燈需要讀秒倒數所以把三個號誌燈繼承一個Light父類別讓燈有sleep()得方法可以倒數

```
class Red extends Light {
    public void change(TrafficLight light) {
        System.out.println("紅燈");
        sleep(5000);
        light.set(new Green()); // 如果考慮彈
    }
}
```

實作State的change方法
每個燈的sleep時間不一樣
Sleep完之後會切換燈號到下一個狀態

2.再來號誌燈會使用State的change方法，當State被改變，change的實際方法也改變

```
class Red extends Liah
    public void change(TrafficLight light) {
        System.out.println("紅燈");
        sleep(5000);
        light.set(new Green()); // 如果考慮
    }
}
```

3.在Sleep完之後會重新把號誌燈Set到下一個綠燈狀態

```
class TrafficLight {
    private State current = new Red();
    void set(State state) {
        this.current = state;
    }
    void change() {
        current.change(this);
    }
}
```

1.迴圈內會一直呼叫號誌燈要Change

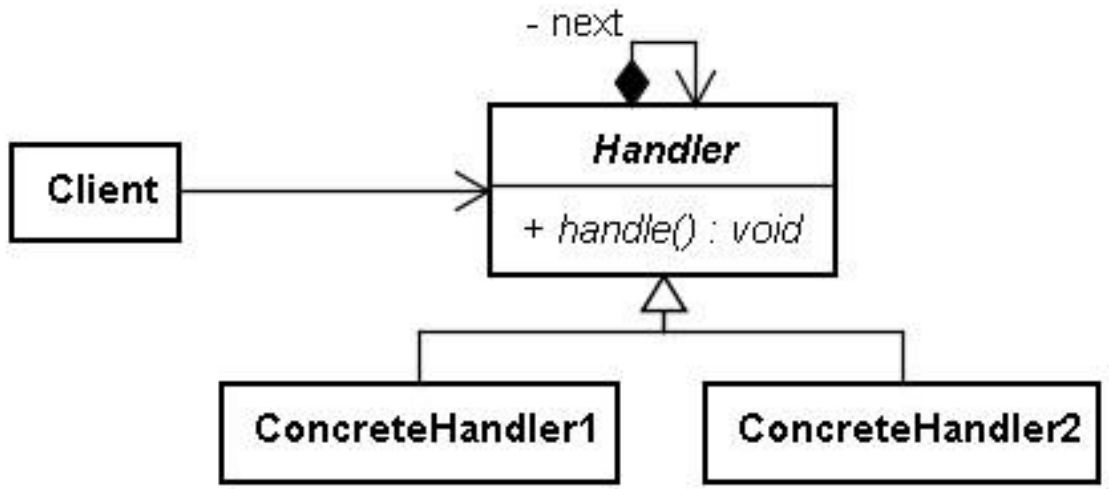
```
public class StatePattern
    public static void mai
        TrafficLight trafficLight = new TrafficLight();
        while(true) {
            trafficLight.change();
        }
    }
}
```

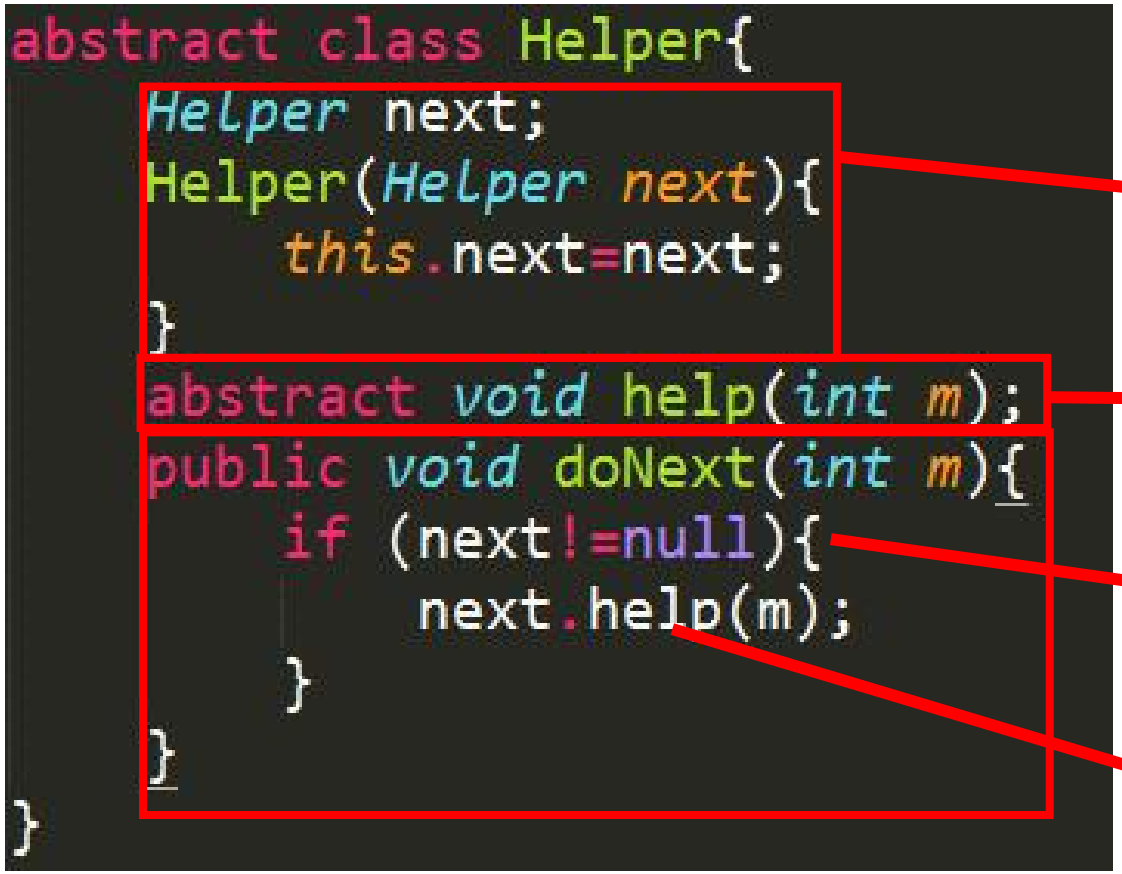
Chain of Responsibility

責任鏈模式

這個Pattern很簡單
就是把很多個處理器串在一起
當這個處理器無法處理就交給下一個

- 可以處理越多事情的Handler放越後面





每個Helper裡面要放下一個Helper
在建構子中放置下一個Helper

這個Helper實際要怎麼做等到
concrete再決定

如果下一個Helper不是空就可以交
給下一位

當這個Helper無法處理的時候要交
給下一個Helper處理

```java
class H_1000 extends Helper{
    H_1000(Helper next){
        super(next);
    }
    public void help(int m){
        if (m>=1000 && m!=0){
        System.out.println("1000 = "+(m / 1000));
        }
        doNext(m % 1000);
    }
}
```

這是一個零錢處理器總共有1000
500、100、10、5、1六個處理器
現在這個是處理1000的，所以只負
責處理一千塊

用取餘數的方式把1000取完之後交
給下一個處理

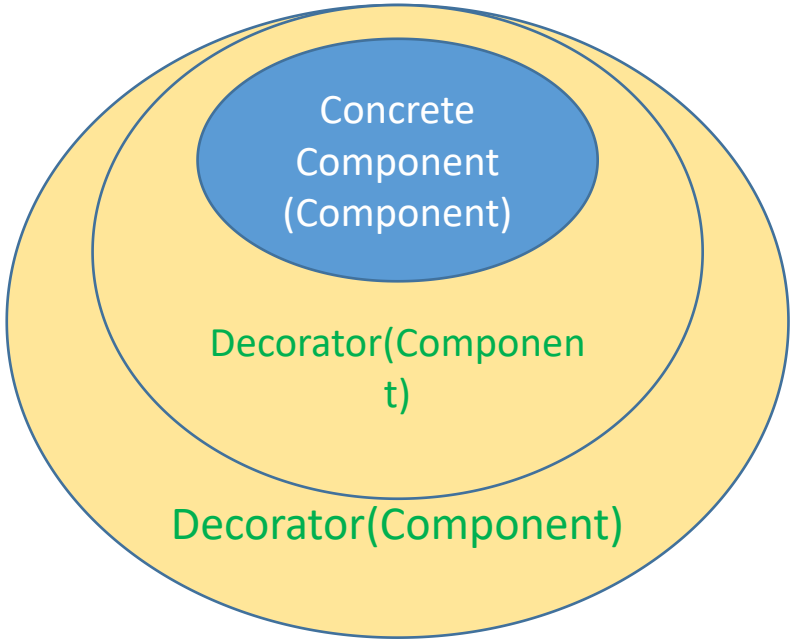
```java
public class CoR {
    public static void main(String[] aras){
        Helper H=new H_1000(new H_500(new H_100(new H_10(new H_5(new H_1(null))))));
        H.help(156437);
    }
}
```

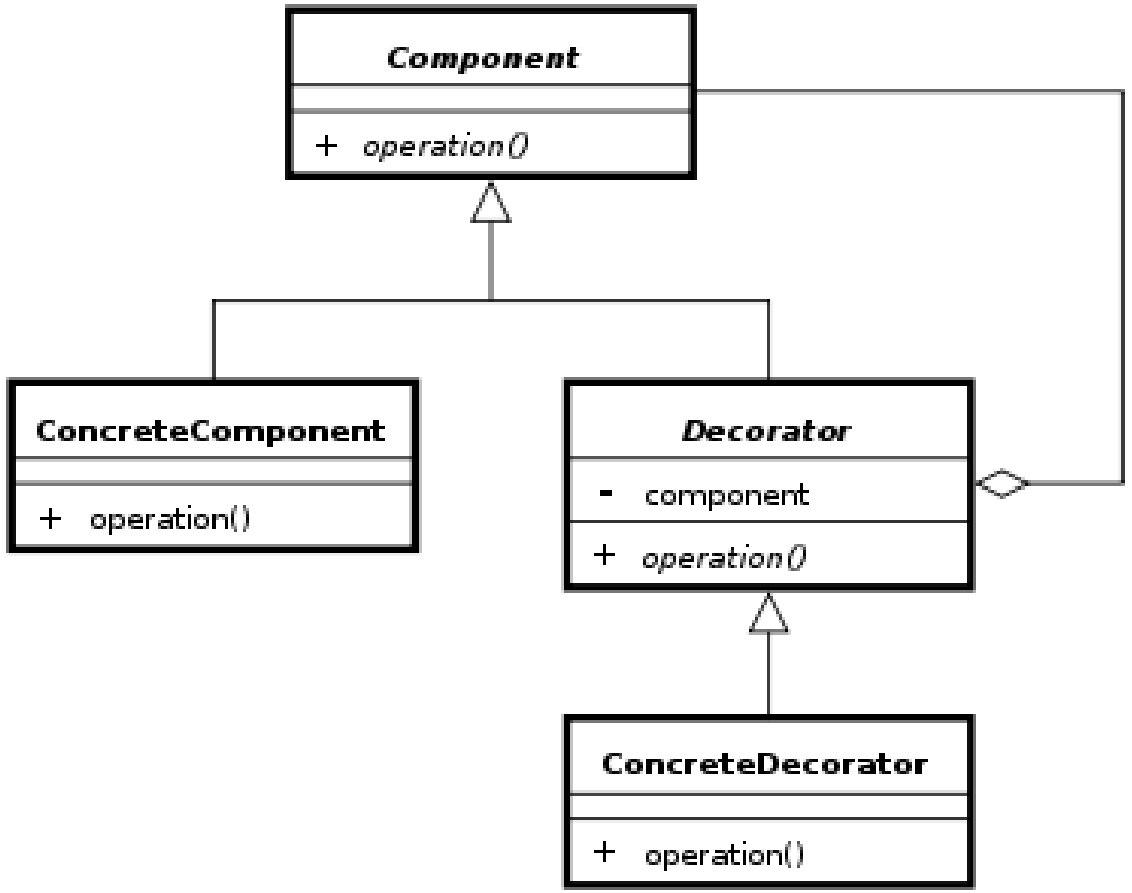
接著把所有Helper
串起來
如果沒有下一個就
放null

```
1000 = 156
100 = 4
10 = 3
5 = 1
1 = 2
```

Decorator

裝飾模式





跟Composite很相似的結構
重點在他可以一層一層的疊上去
用Decorator去包裝ConcreteComponent

- 讓原本的Component很容易擴充新的功能
  而不用修改原本寫好的

```java
interface Meal {
    public String getContent();
    public double getPrice();
}
```

每個餐點都有取得內容跟價錢兩個方法

```java
abstract class AbstractSideDish implements Meal{
    protected Meal meal;
    public AbstractSideDish(Meal m){
        this.meal = m;
    }
}
```

Abstract Decorator 因為要包裝餐點所以裡面放置一個Meal

```java
class FriedChicken implements Meal{
    private String content="烤雞";
    private double price=79;

    @Override
    public String getContent() {
        return content;
    }

    @Override
    public double getPrice() {
        return price;
    }
```

Concrete component (被包裝的)
不能包裝別人
放在最底層的

```java
class SideDishOne extends AbstractSideDish{

    public SideDishOne(Meal m){
        super(m);
    }

    @Override
    public String getContent() {
        return meal.getContent()+" |加購|可樂+薯條";
    }

    @Override
    public double getPrice() {
        return meal.getPrice() + 30;
    }
}
```

Concrete Decorator
在呼叫方法的時候會呼叫內部一層
的相同方法再加上自己的
以達到包裝(擴充)

有點類似遞迴的概念

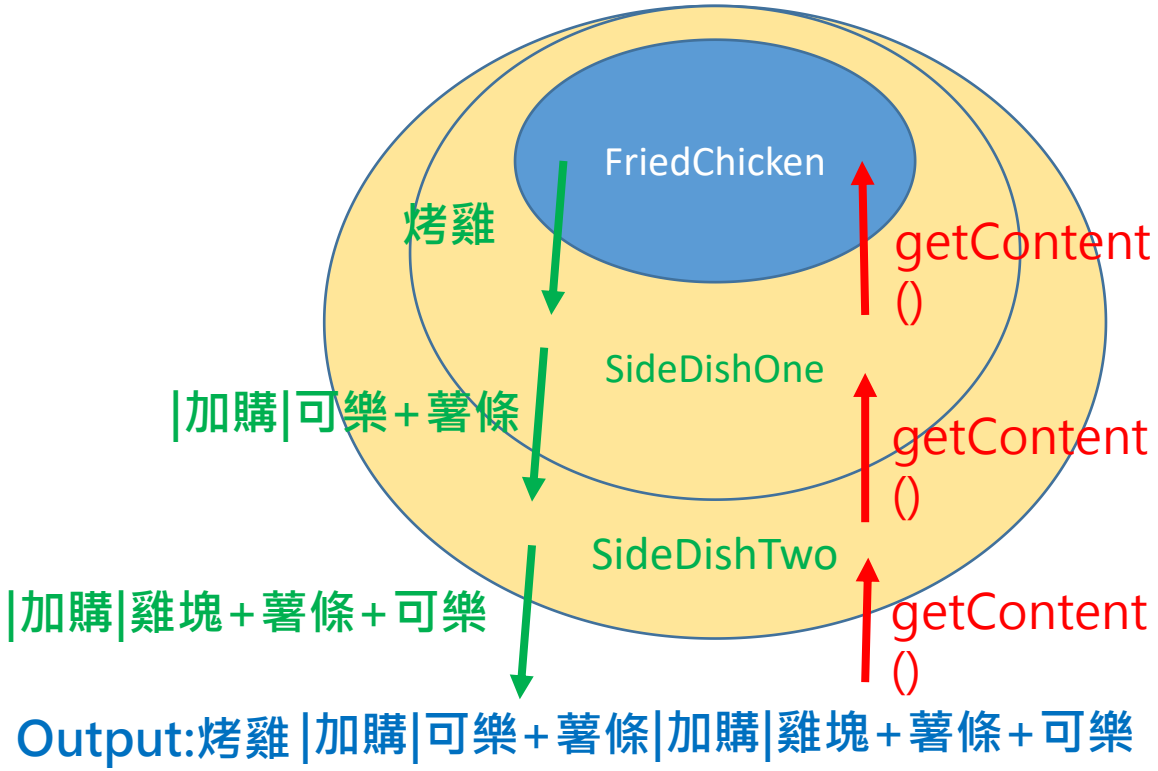
```java
    Meal meal=new FriedChicken();
    meal=new SideDishOne(meal);
    System.out.println(meal.getContent());
    System.out.println(meal.getPrice());
    meal=new SideDishTwo(meal);
    System.out.println(meal.getContent());
    System.out.println(meal.getPrice());
```

從這個例子來看
主餐(FriedChicken)總共包了兩層
先包SideDishOne再包SideDishTwo

```
烤雞 |加購|可樂+薯條
109.0
烤雞 |加購|可樂+薯條 |加購|雞塊+薯條+可樂
178.0
```

```
Meal meal=new FriedChicken();
meal=new SideDishOne(meal);
System.out.println(meal.getContent());
System.out.println(meal.getPrice());
meal=new SideDishTwo(meal);
System.out.println(meal.getContent());
System.out.println(meal.getPrice());
```

從這個例子來看
主餐(FriedChicken)總共包了兩層
先包SideDishOne再包SideDishTwo




```
烤雞 |加購|可樂+薯條
109.0
烤雞 |加購|可樂+薯條 |加購|雞塊+薯條+可樂
178.0
```

Price也是同樣的原理

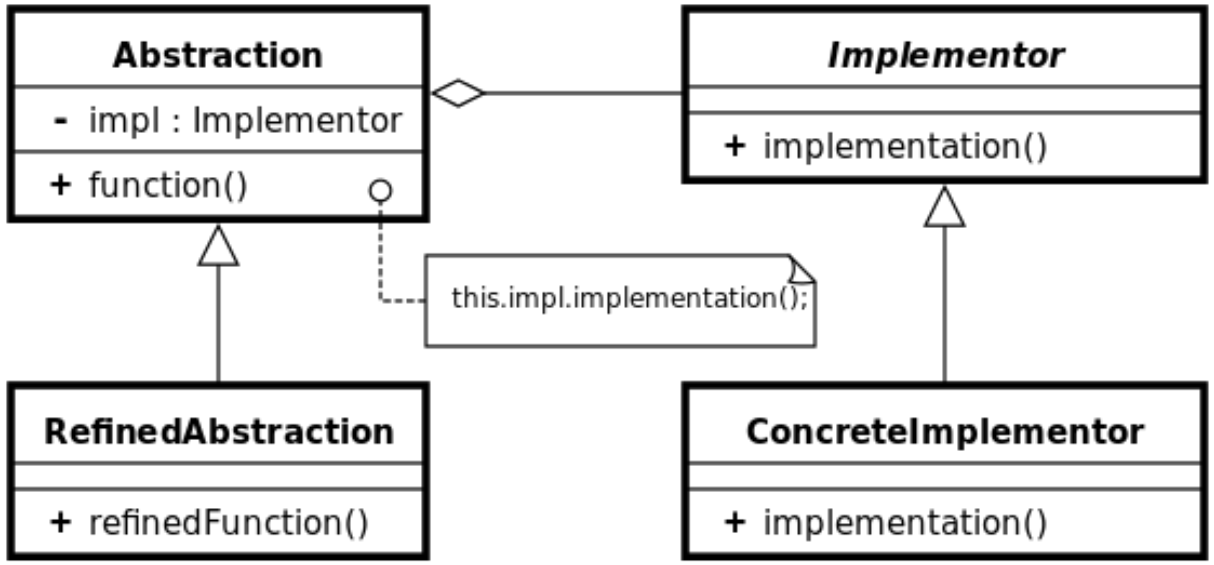
Bridge

橋接模式

將抽象部分與它的實現部分分離，使它們都可以獨立地變化。

假設左邊有三種框架，右邊有四種實作那總共就有3x4=12種變化
所以只要一方增加了就可以多出很多種變化

例如：軟體可以選擇要用什麼語言



跟Strategy比較

- Strategy是行為pattern強調的是讓使用者可以選擇怎樣的方式去做
- Bridge是結構pattern他強調的是把架構跟實作分離

小敏會強調Bridge是架構和實作的所有組合都能夠實現

```java
interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}

class RedCircle implements DrawAPI{
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("畫個圓[ 顏色: 紅色, radius: " + radius + ", x: " + x + ", " + y + "]");
    }
}

class GreenCircle implements DrawAPI{
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("畫個圓[ 顏色: 綠色, radius: " + radius + ", x: " + x + ", " + y + "]");
    }
}
```

其實就是跟Stragegy很相似的寫法

這是實作的部分 DrawAPI先定義好方法drawCircle

然後有兩個實際的方法RedCircle跟GreenCircle

```java
abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}

class Circle extends Shape{
    private int x, y, radius;
    protected Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public void draw() {
        drawAPI.drawCircle(radius,x,y);
    }
}
```

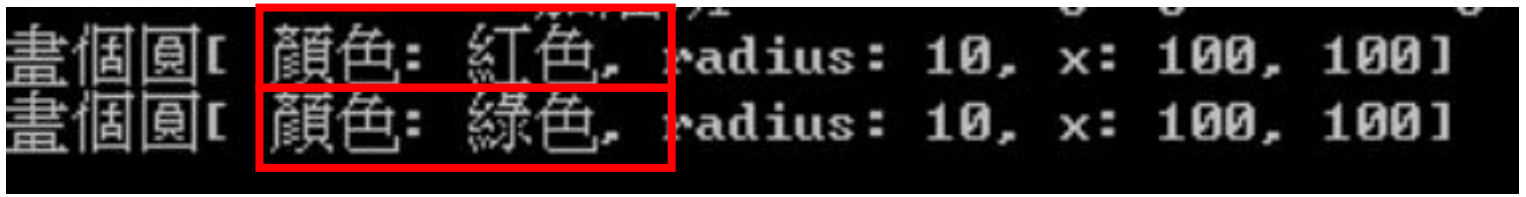
架構的地方
在裡面放置一個DrawAPI

根據放進來的API做對應得畫圖方法

在這個例子中只有一種Circle
配兩種顏色(Red,Green)所以只有兩種結果

```java
public class BridgePattern {
    public static void main(String[] args) {
        Shape redCircle = new Circle(100,100, 10, new RedCircle());
        Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

        redCircle.draw();
        greenCircle.draw();
    }
}
```

在這個例子中只有一種Circle
配兩種顏色(Red,Green)所以只有兩種
結果



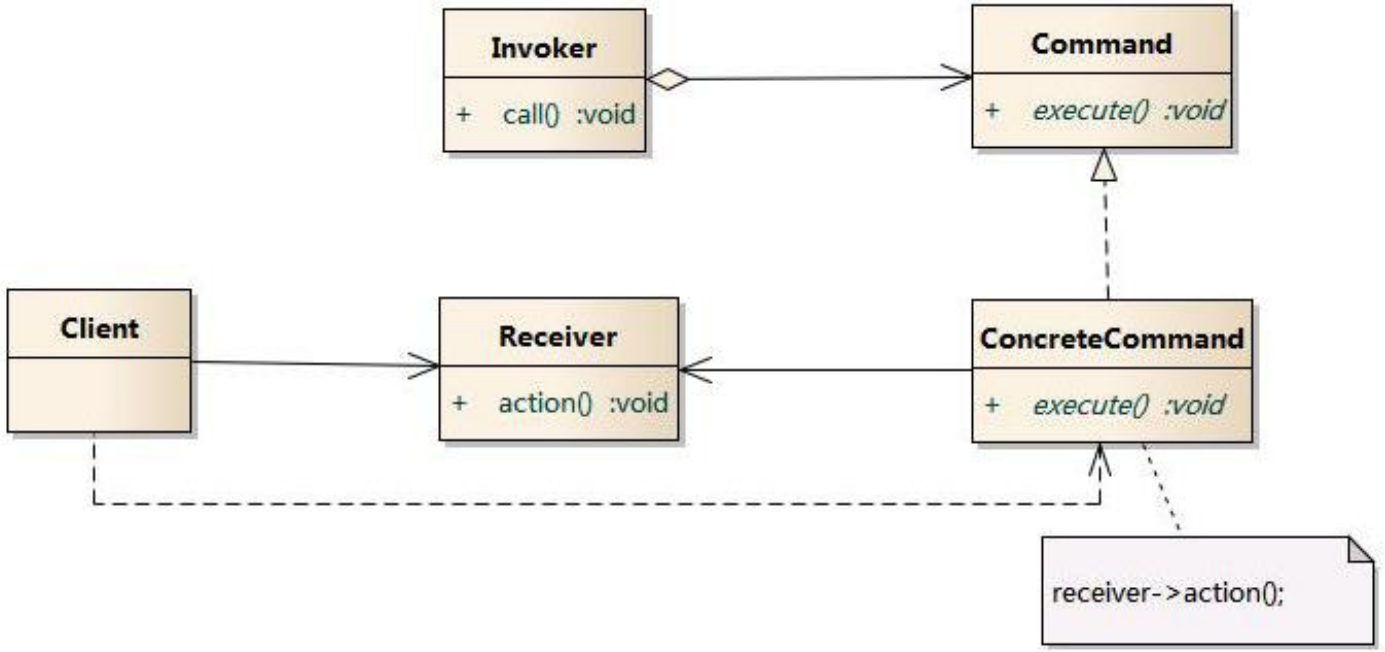
Command

命令模式

簡單來說就是把一些常用的、常常重複的
指令包裝成Command

需要用到的時候就透過Command來執行
增加ReUse

Invoker是存放、執行Command的
Command是操作Reciver的
Reciver是真正有功能的



- 降低耦合度
- 很容易增加新的命令
- 可以從小的命令(micro Command)組成
  一個大的命令(combined Command)
- 實作Undo、Redo方便

```java
//Command
interface Command{
  public void execute();
}
```

最基本的Command，只有execute功能

```java
//Concrete Command
class LightOnCommand implements Command{
  //reference to the light
  Light light;
  public LightOnCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOn();
    System.out.println("LightOn.");
  }
```

Reciver 用來執行動作

真正的動作是由Reciver執行

```java
class Light{
  private boolean on;
  public void switchOn(){
    on = true;
  }
  public void switchOff(){
    on = false;
  }
}
```

實際被Command操作的Reciver

```
class RemoteControl{
  private Command command;
  public void setCommand(Command command){
    this.command = command;
  }
  public void pressButton(){
    command.execute();
  }
}
```

Invoker 裡面存放要被執行的Command

執行Command

```
public class CommandDemo{
  public static void main(String[] args)    {
    RemoteControl control = new RemoteControl();
    Light light = new Light();
    Command lightsOn = new LightOnCommand(light);
    Command lightsOff = new LightOffCommand(light);
    //switch on
    control.setCommand(lightsOn);
    control.pressButton();
    //switch off
    control.setCommand(lightsOff);
    control.pressButton();
  }
}
```

把Command建造出來並把Reciver丟進去

在Set進Invoker就可以執行Command

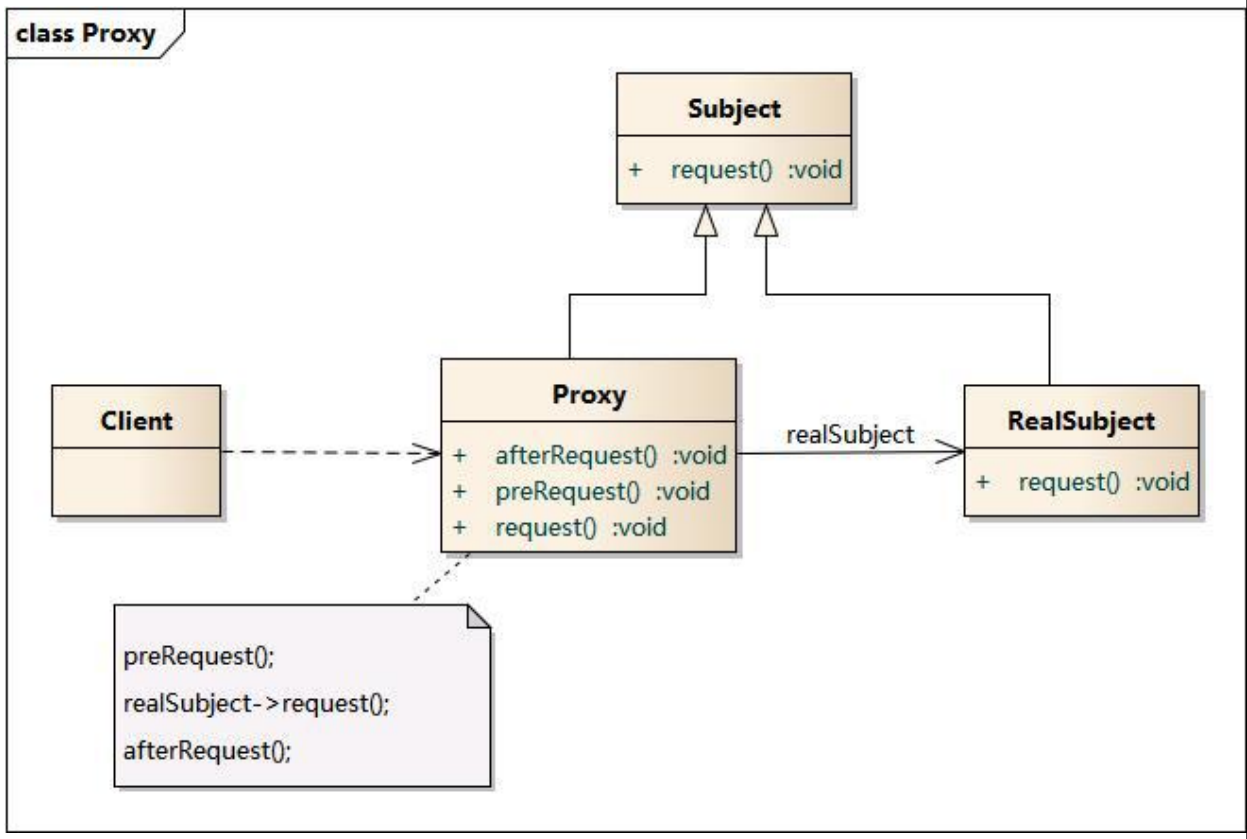
```
LightOn.
LightOff.
```

Proxy

代理模式

透過一個代理人來代替Real的東西
根據功能不同大致可以分四種

- 虛擬代理(Virtual Proxy)用比較不消耗資源的代理物件來代替實際物件，實際物件只有在真正需要才會被創造
- 遠程代理(Remote Proxy)在本地端提供一個代表物件來存取遠端網址的物件
- 保護代理(Protect Proxy)限制其他程式存取權限
- 智能代理(Smart Reference Proxy)為被代理的物件增加一些動作



Proxy應改比較少問實作，畢竟有很多種。不過就先放一個簡單的Proxy

```
interface Image {
    void display();
}
```

定義好Image的方法

```
class ProxyImage implements Image {

    private RealImage realImage;
    private String fileName;

    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```

在讀取時先判斷圖片是否已經有了，如果沒有就開始讀取

讀取完畢就可以直接顯示

```java
class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);

    }

    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }
}
```

在建構的時候便開始讀取

顯示真正的圖片

```java
Image image = new ProxyImage("test.jpg");
image.display();
System.out.println("");
image.display();
```

我們就可以透過Proxy來讀取
而不用操作真正的Image

```
Loading test.jpg
Displaying test.jpg

Displaying test.jpg
```

Builder

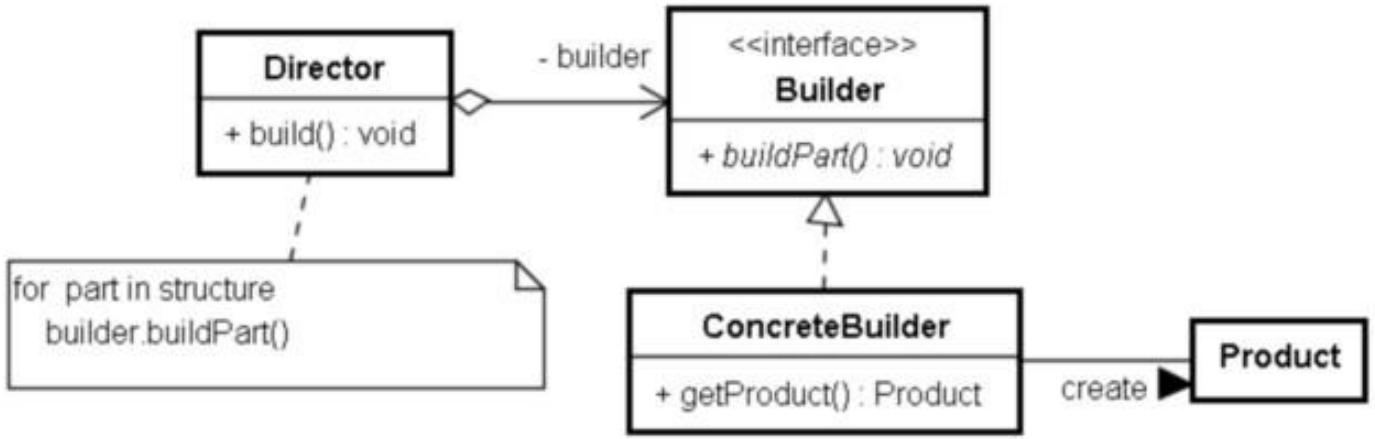
建造者模式

這個模式用來建造由複雜組成的產品

Builder可以製造許多part
然後組成一個Product

每一個ConcreteBulider會根據Director的規定建立產品

Director裡面會規定如何建造
再透過ConcreteBulider來取得產品



```java
class Car {
    private int wheels;
    private String color;
    @Override
    public String toString() {
        return "Car [wheels = " + wheels + ", color = " + color + "]";
    }

    public int getWheels() {
        return wheels;
    }

    public void setWheels(final int wheels) {
        this.wheels = wheels;
    }

    public String getColor() {
        return color;
    }

    public void setColor(final String color) {
        this.color = color;
    }
}
```

要產出的Product

toString的方法是
如果直接Print物件
系統會自統呼叫toString方法
如果沒有複寫則會印出記憶體位置..等資料

其他都是設定跟取得值的方法

```java
interface CarBuilder {
    CarBuilder setWheels(final int wheels);

    CarBuilder setColor(final String color);

    Car build();
}
```

定義好Builder的方法

```java
class CarBuilderImpl implements CarBuilder {
    private Car car;

    public CarBuilderImpl() {
        car = new Car();
    }

    @Override
    public CarBuilder setWheels(final int wheels) {
        car.setWheels(wheels);
        return this;
    }

    @Override
    public CarBuilder setColor(final String color) {
        car.setColor(color);
        return this;
    }

    @Override
    public Car build() {
        return car;
    }
}
```

ConcreteBulider會建造好產品
再根據Director的指令
Set好產品的參數(或是組成產品)

然後Director就可以取得產品

```java
class Director{
    private CarBuilder builder;

    public Director(final CarBuilder builder) {
        this.builder = builder;
    }

    public Car construct() {
        return builder.setWheels(4).setColor("Red").build();
    }
}
```

Director根據不同的Builder取得不同的產品

Director會決定建造的方法來構成產品的內部構造(參數、組成方式)

```java
public class BuliderDemo {

    public static void main(final String[] arguments) {
        CarBuilder builder = new CarBuilderImpl();
        Director director = new Director(builder);
        System.out.println(director.construct());
    }
}
```

使用者透過Director就可以取得產品

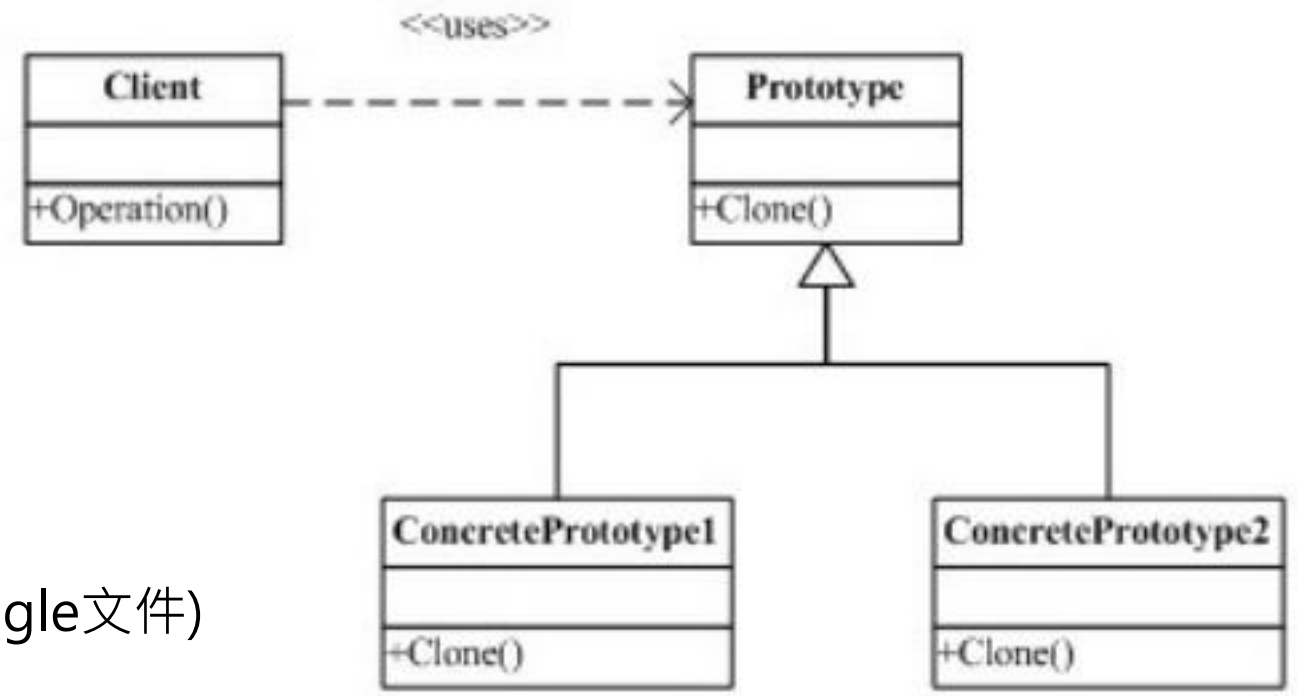
```
Car [wheels = 4, color = Red]
```

Prototype

原型模式

這個Pattern簡單來說就是用來複製
Prototype是原體，
ConcretePrototype就是複製體

很簡單，問題在於如何去複製



老師之前上課有提到Copy-on-Write
這個在多人運行的平台下比較會看到(向google文件)

- Copy-on-Write(寫入時複製):就是當你要修改的時候先把原本的複製一份起來，而你改的會是複製的那一份，別人再看的時候還是原本的，如果你真的要儲存的時候，才會把修改的地方在原始的檔案上修改
- 可以參考Wiki

淺層複製 (Shallow Copy)

當原型被修改

複製體也會跟著改

```
public Prototype ShallowClone(){
    return this;
}
```

物件還是參照原型

深層複製 (Deep Copy)

當原型被修改

複製體不會跟著改

```
public Prototype DeepClone(){
    Prototype clone=new ConcretePrototype();
    ((ConcretePrototype)clone).setNumber(number);
    return clone;
}
```

製造一個新的物件

```java
public class PrototypeDemo{
    public static void main(String[] arg){
        ConcretePrototype o=new ConcretePrototype();
        o.setNumber(5);
        ConcretePrototype shallow=(ConcretePrototype)o.ShallowClone();
        ConcretePrototype deep=(ConcretePrototype)o.DeepClone();
        System.out.println("修改前..");
        System.out.println("原型："+o.getNumber()
            +"  ShallowClone:"+shallow.getNumber()
            +"  DeepClone:"+deep.getNumber());

        o.setNumber(10);
        System.out.println("修改後..");
        System.out.println("原型："+o.getNumber()
            +"  ShallowClone:"+shallow.getNumber()
            +"  DeepClone:"+deep.getNumber());
    }
}
interface Prototype{
    Prototype ShallowClone();
    Prototype DeepClone();
}
class ConcretePrototype implements Prototype{
    private int number;
    public void setNumber(int n){number=n;}
    public int getNumber(){return number;}
    public Prototype ShallowClone(){
        return this;
    }
    public Prototype DeepClone(){
        Prototype clone=new ConcretePrototype();
        ((ConcretePrototype)clone).setNumber(number);
        return clone;
    }
}
```

```
修改前..
原型：5  ShallowClone:5  DeepClone:5
修改後..
原型：10  ShallowClone:10  DeepClone:5
```

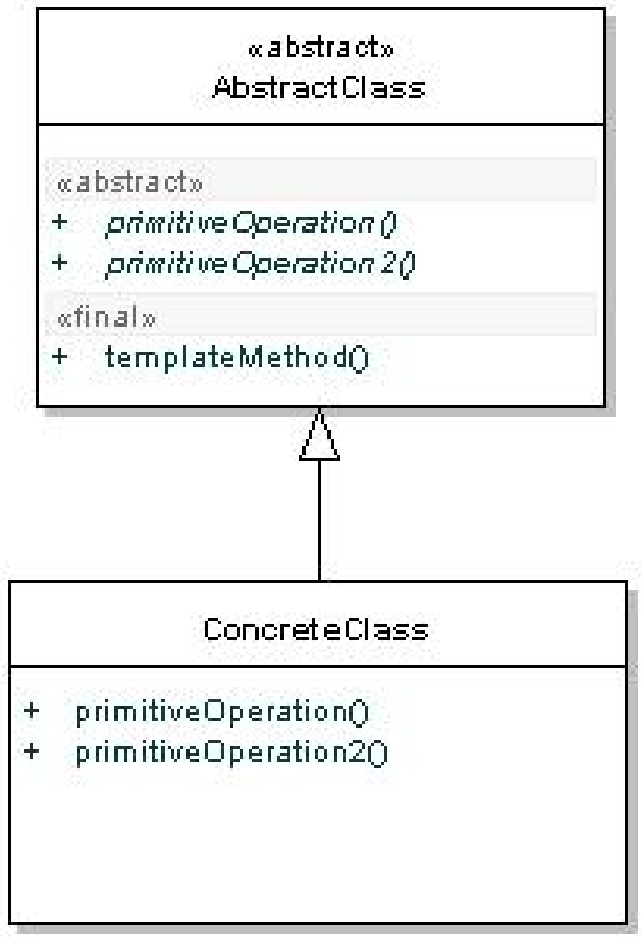
Template

樣板模式

Template 顧名思義就是提供一個固定的樣板

你可以自行修改樣板的方法來達到不一樣的效果

- Template Method必須是Final因為要定義好執行順序
- Primitive是Abstract必須被複寫的方法(會根據情況改變-切換演算法)
- Hook有兩種說法
  1.老師一直說的他是一個Boolean值，當Template執行的過程中可以根據這個掛勾決定要不要執行這一段
  2.另外就是網路上說的Hook是一個預設為空的方法(Concrete)，子類別可以選擇要不要付寫這個方法來擴充功能
- Concrete methods偶爾會有如果是通用的方法就可以先實作好



常拿來跟Strategy比較
因為兩個都是切換演算法的Pattern

- Strategy是Runtime
- Template是Compiler Time

```java
abstract class Game {
    /* Hook methods. Concrete implementation may differ
    protected int playersCount;
    protected int number;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        number = 0;
        while (!endOfGame()) {
            makePlay(number);
            number += 1;
        }
        printWinner();
    }
}
```

先定義一套Template
並把Template Method寫好

這裡的是一個遊戲 從0開始跑
跑到endofGame()回傳true則停止遊戲

Hook Method

```java
class Monopoly extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Initialize money
        System.out.println("initializeGame.");
    }
    void makePlay(int player) {
        // Process one turn of player
        System.out.println("makePlay. "+number);
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
        if (number>15){
            return true;
        }else{
            return false;
        }
    }
    void printWinner() {
        System.out.println("Winner:"+number % playersCount);
    }

}
```

```java
public class TemplateDemo{
    public static void main(String[] arg){
        Game game=new Monopoly();
        game.playOneGame(7);
    }
}
```

Contrete Template
根據複寫掉的方法
會執行不一樣的方法

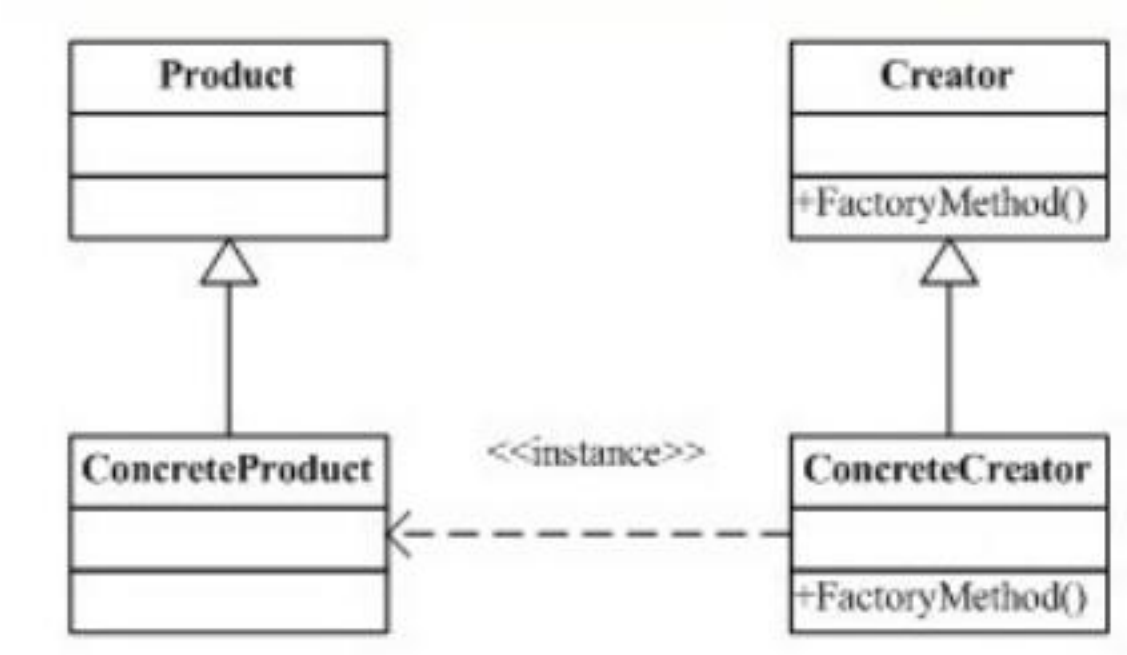
這裡設定超過15就停止

```
initializeGame.
makePlay. 0
makePlay. 1
makePlay. 2
makePlay. 3
makePlay. 4
makePlay. 5
makePlay. 6
makePlay. 7
makePlay. 8
makePlay. 9
makePlay. 10
makePlay. 11
makePlay. 12
makePlay. 13
makePlay. 14
makePlay. 15
Winner:2
```

Factory Method

工廠方法

一個工廠對應一個ConcreteFactory
多一個產品就要多一種ConcreteFactory



```
abstract class Shape {

  public abstract void draw();
  public String name;
  public Shape(String aName){
    name = aName;
  }
}

class Circle extends Shape {
  public void draw() {
    System.out.println("It will draw a circle."+name);
  }

  public Circle(String aName){
    super(aName);
  }
}

class Square extends Shape {
  public void draw() {
    System.out.println("It will draw a square."+name);
  }

  public Square(String aName){
    super(aName);
  }
}
```

定義好Product並實作好
Concrete Product的Method

```
abstract class ShapeFactory {
  protected abstract Shape factoryMethod(String aName);

}


class CircleFactory extends ShapeFactory {

  protected Shape factoryMethod(String aName) {
    return new Circle(aName + " (created by CircleFactory)");
  }
}


class SquareFactory extends ShapeFactory {

protected Shape factoryMethod(String aName) {
    return new Square(aName + " (created by SquareFactory)");
  }
}
```

再來定義好工廠的方法

並把Concrete Factory根據對應的Product實作好

```
ShapeFactory sf1 = new SquareFactory();
ShapeFactory sf2 = new CircleFactory();
sf1.factoryMethod("Shape one").draw();
sf2.factoryMethod("Shape two").draw();
```

就可以透過工廠製造產品，不用透過建構子(封裝細節)

```
It will draw a square.Shape one (created by SquareFactory)
It will draw a circle.Shape two (created by CircleFactory)
```

Abstract Factory

抽象工廠

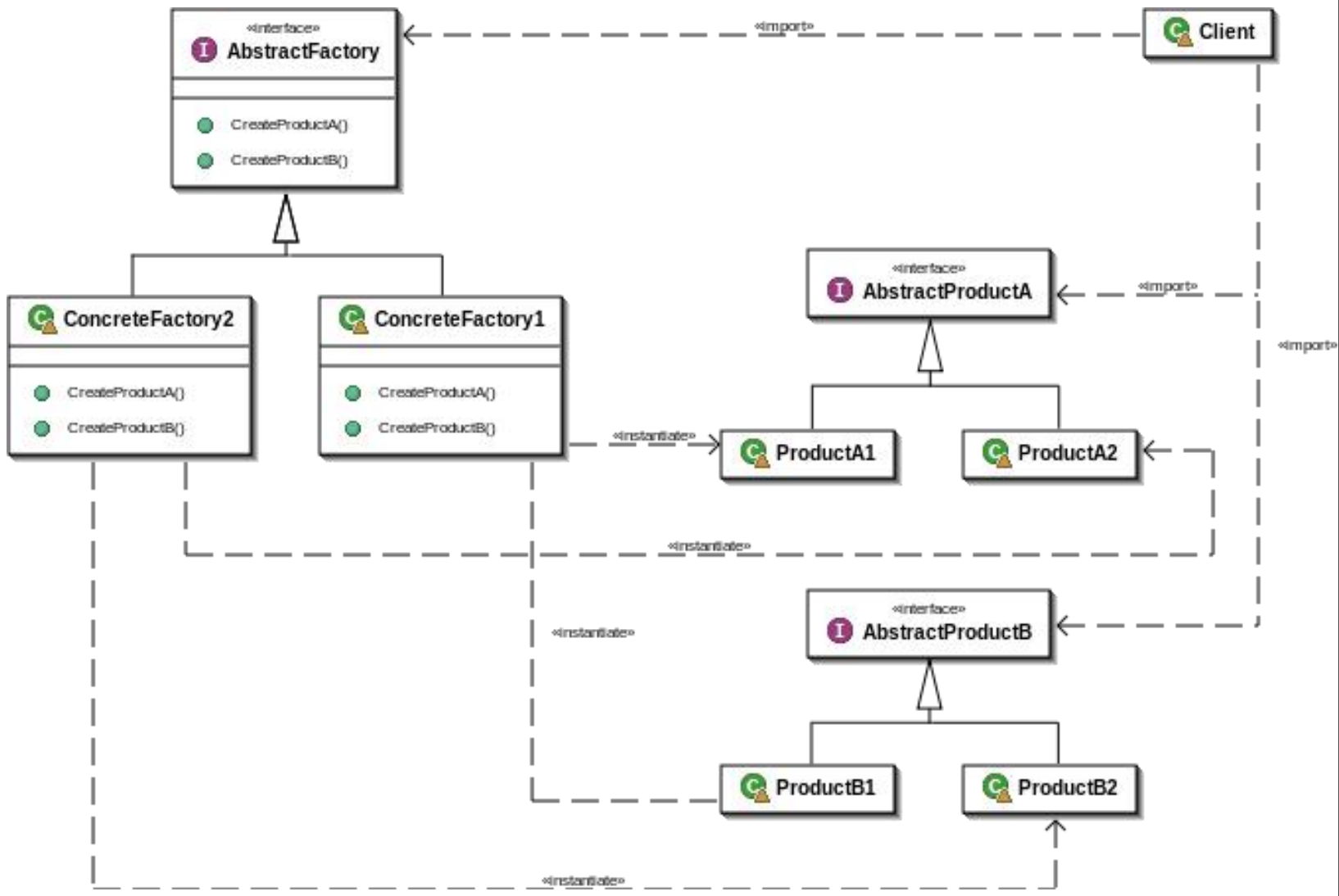
相似的產品
透過不同工廠生產
變成另一樣不同的產品

就像Nike的拖鞋今天換成addias工廠生產
就變成addias拖鞋

以右圖來說有A、B兩種產品

透過工廠1生產的就是產品A1、B1

透過工廠2生產的就是產品A2、B2



```
abstract class Product{
    abstract public String open();
}
class MacNoteBook extends Product{

    public String open(){
        return "Open Macbook.";
    }
}
class WindowsNoteBook extends Product{

    public String open(){
        return "Open WindowsNoteBook.";
    }
}
```

首先建立好產品
在這裡只有Product一種產品
那因為有兩個工廠所以實體建造出來會有
兩個Concrete Product

```
interface AbstractFactory{
    Product createnoteBook();
}

class MacFactory implements AbstractFactory{
    public Product createnoteBook(){
        return new MacNoteBook();
    }
}
class WindowsFactory implements AbstractFactory{
    public Product createnoteBook(){
        return new WindowsNoteBook();
    }
}
```

再來是工廠的部分
不同工廠就建造相對應的產品

```java
public class AbstractFactoryDemo{
    public static void main(String[] arg){
        AbstractFactory F1=new MacFactory();
        AbstractFactory F2=new WindowsFactory();
        System.out.println(F1.createnoteBook().open());
        System.out.println(F2.createnoteBook().open());
    }
}
```

```
Open Macbook.
Open WindowsNoteBook.
```

就可以透過不同工廠產生
不一樣廠牌但相似的產品

可以參考

https://openhome.cc/Gossip/DesignPattern/

http://design-patterns.readthedocs.io/zh_CN/latest/index.html

https://en.wikipedia.org/wiki/Design_Patterns

https://skyyen999.gitbooks.io/-study-design-pattern-in-java/content/