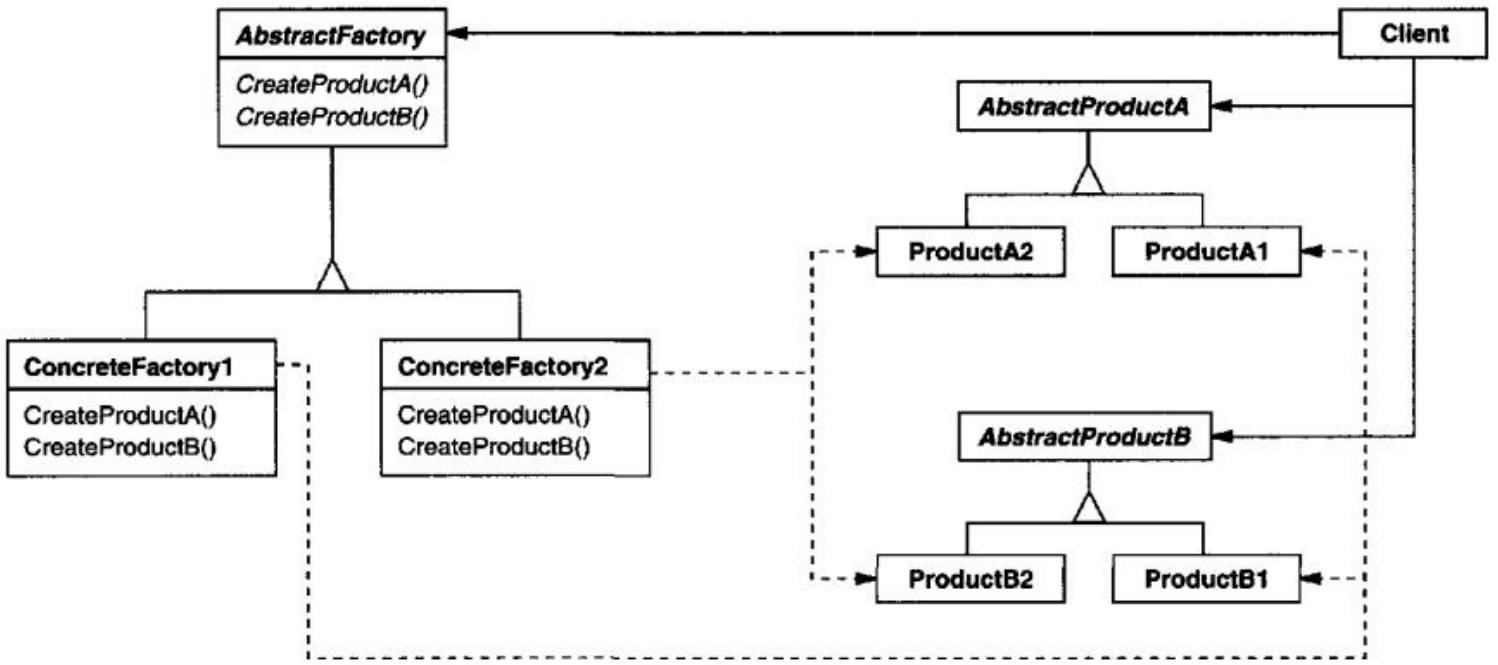
1. Abstract Factory 抽象工廠

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

提供用於創建相關或從屬對象族的接口，而無需指定其具體類。



相似產品 不同工廠生產 變成不同產品

Nike的拖鞋(A1)
原本是Nike工廠生產(F1)

換成Addias工廠生產(F2)
變成addias拖鞋(A2)

以圖來說有A、B兩種產品
Factory1生產的是產品A1、B1
Factory2生產的是產品A2、B2

```java
//AbstractA
interface Shape {
   void draw();
}
class Square implements Shape {
   public void draw() {
      System.out.println("Inside Square::draw()");}
}
class Circle implements Shape {
   public void draw() {
      System.out.println("Inside Circle::draw()");}
}
//AbstractB
interface Color {
   void fill();
}
class Red implements Color {
   public void fill() {
      System.out.println("Inside Red::fill()");}
}
class Green implements Color {
   public void fill() {
      System.out.println("Inside Green::fill()");}
}
public class main {
   public static void main(String[] args) {
      AbstractFactory shape = new ShapeFactory();
      shape.getShape("CIRCLE").draw();
      shape.getShape("SQUARE").draw();
      AbstractFactory color = new ColorFactory();
      color.getColor("RED").fill();
      color.getColor("Green").fill();
   }
}
```

```
Inside Circle::draw()
Inside Square::draw()
Inside Red::fill()
Inside Green::fill()
```

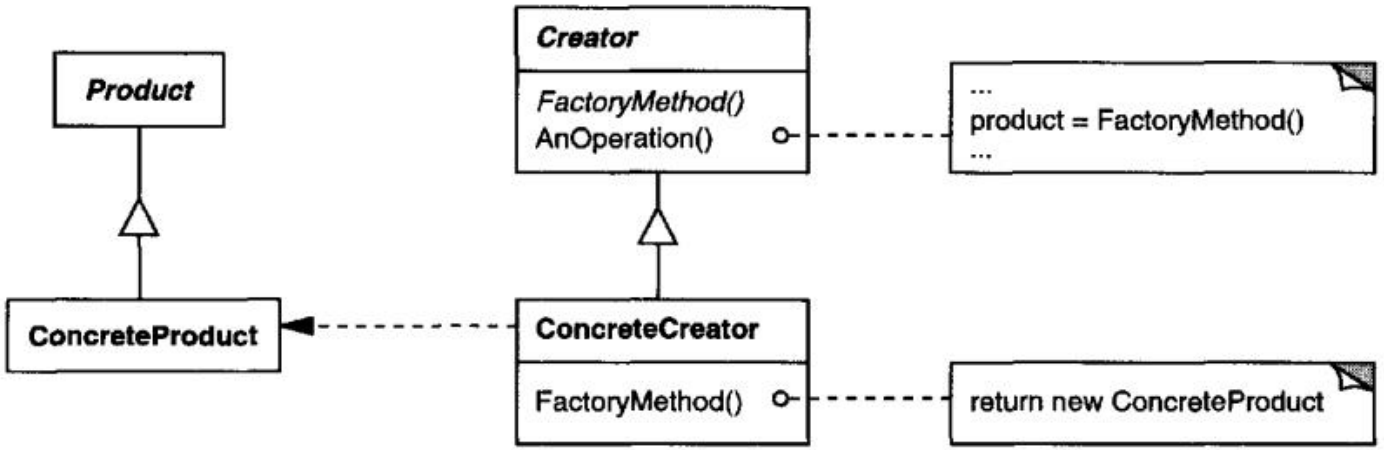
1.不同工廠就建造相對應的產品
2.只有Product一種產品 因為有兩個工廠
所以實體建造出來會有兩個Concrete Product
3.可以透過不同工廠產生不同廠牌但相似的產品

```java
abstract class AbstractFactory {
   public abstract Color getColor(String color);
   public abstract Shape getShape(String shape);
}
class ShapeFactory extends AbstractFactory {
   public Shape getShape(String in){
      if(in.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(in.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;}
   public Color getColor(String color) {  return null;}
}
class ColorFactory extends AbstractFactory {
   public Color getColor(String color) {
      if(color.equalsIgnoreCase("RED")){
         return new Red();
      } else if(color.equalsIgnoreCase("GREEN")){
         return new Green();
      }
      return null;}
   public Shape getShape(String in){ return null;}
}
```

2. Factory Method 工廠方法

Define an interface for creating an object, but let subclasses decide which class to instantiate.
Factory Method lets a class defer instantiation to subclasses.

定義用於創建對象的接口，但讓子類決定實例化哪個類。
Factory Method允許類將實例化延遲到子類。



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

```java
interface Shape {
   void draw();
}
class Rectangle implements Shape {
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");}
}
class Square implements Shape {
   public void draw() {
      System.out.println("Inside Square::draw() method.");}
}
class Circle implements Shape {
   public void draw() {
      System.out.println("Inside Circle::draw() method.");}
}
//Creator+ConcreteCreate
class ShapeFactory {
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

1.把Concrete Factory根據對應的Product實作
2.實作Concrete Product的Method
3.透過工廠製造產品，不用透過建構子(封裝細節)

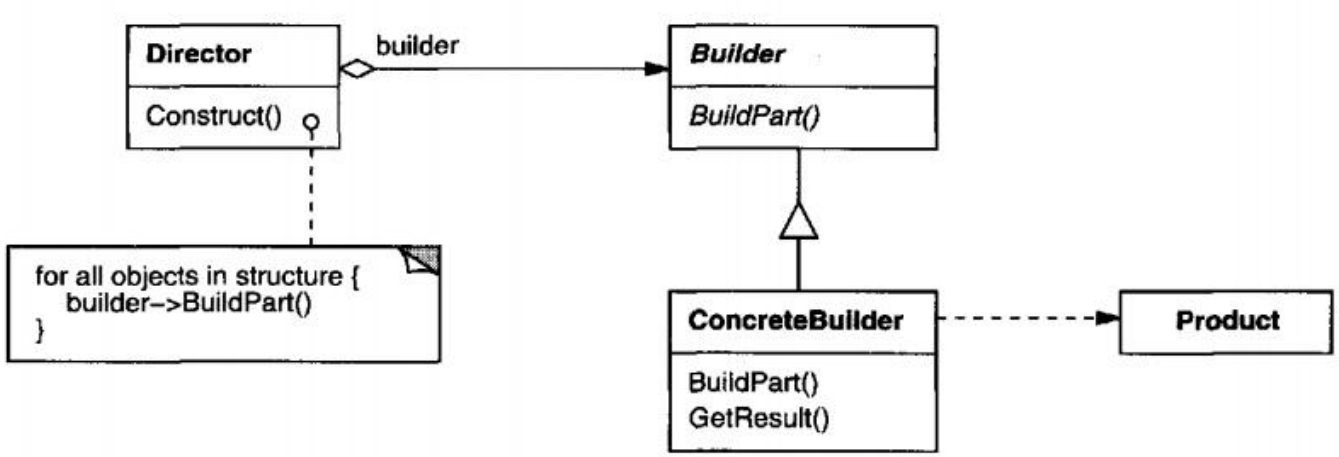
```java
public class main {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();
      shapeFactory.getShape("CIRCLE").draw();
      shapeFactory.getShape("RECTANGLE").draw();
      shapeFactory.getShape("SQUARE").draw();
   }
}
```

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

3. Builder 建造者模式 ***

Separate the construction of a complex object from its representation
so that the same construction process can create different representations.

將復雜對象的構造與其表示分開，以便將其複制相同的施工過程可以創建不同的表示。



這個模式用來建造由複雜組成的產品
Builder可以製造許多part 組成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());
    }
}
```

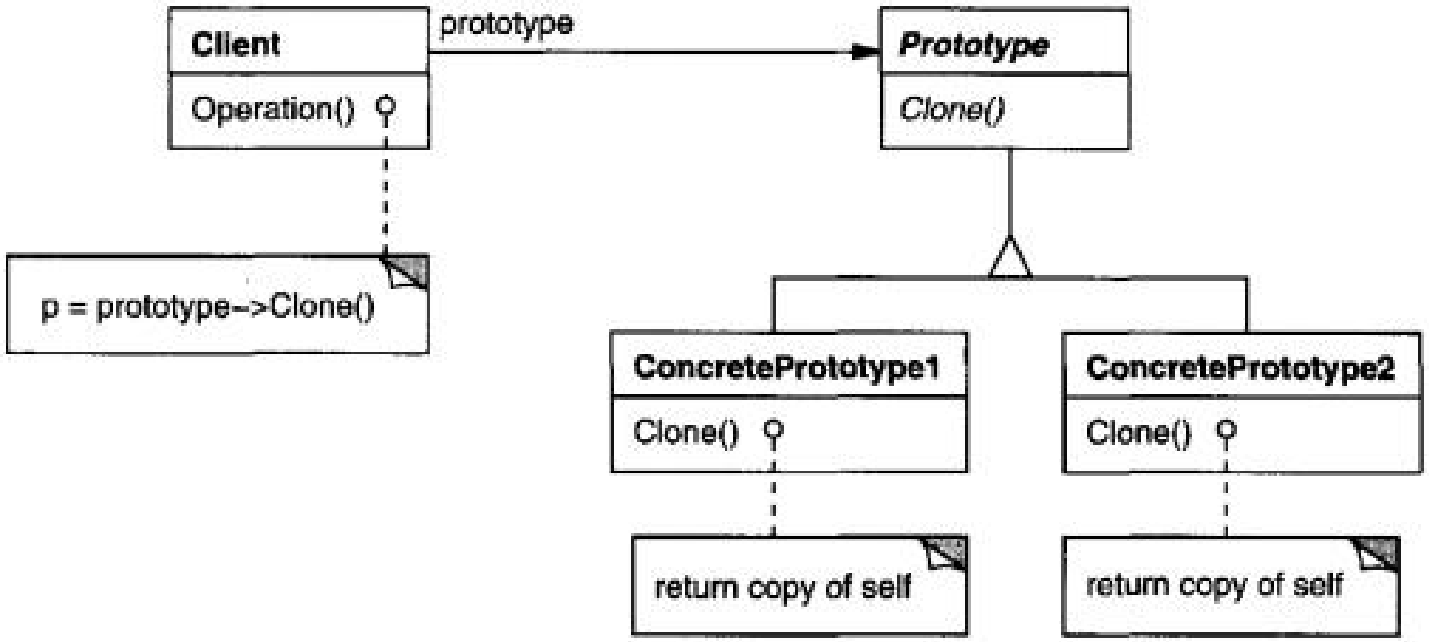
使用者透過Director就可以取得產品

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

4. Prototype 原型模式

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

使用原型實例指定要創建的對象類型，然後創建新對象複製此原型的對象。



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

```
interface Prototype{
    Prototype ShallowClone();
    Prototype DeepClone();
}
class Concrete implements Prototype{
    private int number;
    public void set(int n){
        number = n;
    }
    public int get(){
        return number;
    }
    public Prototype ShallowClone(){
        return this;
    }
    public Prototype DeepClone(){
        Prototype clone = new Concrete();
        ((Concrete)clone).set(number);
        return clone;
    }
}
```

Shallow Copy 當原型被修改，複製體也會跟著改

Deep Copy 當原型被修改，複製體不會跟著改

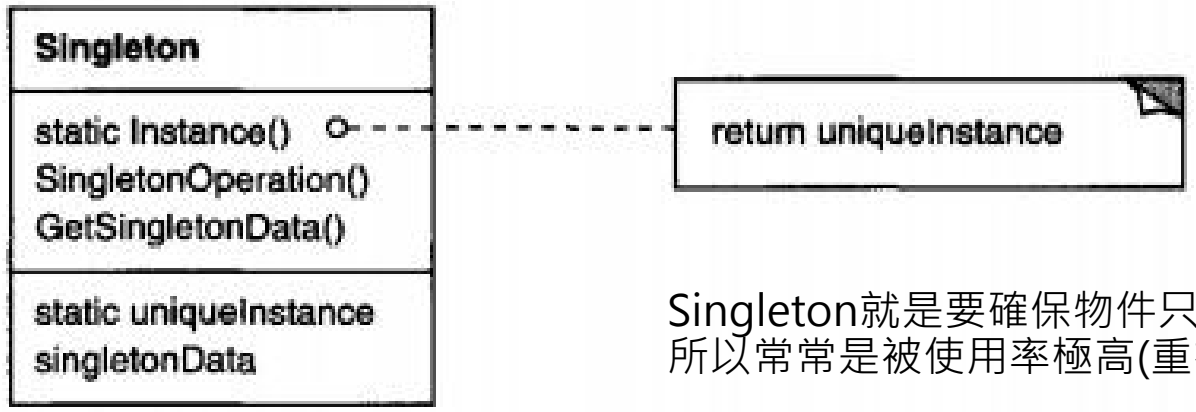
```
public class main{
    public static void main(String[] args) {
        Concrete o = new Concrete();
        o.set(5);
        Concrete shallow = (Concrete)o.ShallowClone();
        Concrete deep = (Concrete)o.DeepClone();
        System.out.println("before..");
        System.out.println("Prototype:"+o.get()
            +"   ShallowClone:"+shallow.get()
            +"   DeepClone:"+deep.get());

        o.set(10);
        System.out.println("After..");
        System.out.println("Prototype:"+o.get()
            +"   ShallowClone:"+shallow.get()
            +"   DeepClone:"+deep.get());
    }
}
```

```
before..
Prototype:5   ShallowClone:5   DeepClone:5
After..
Prototype:10   ShallowClone:10   DeepClone:5
```

5. Singleton 單例模式

Ensure a class only has one instance, and provide a global point of access to it.

確保一個類只有一個實例，並提供一個全局訪問點。



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

LazySingleton 被需要的時候才建造實體

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

Problem：在大量(多人)存取時可能會造成問題產生多個實體

若為否則在此時建造實體
在取得實體時判斷是否已經建造

EagerSingleton一開始就存在著實體

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

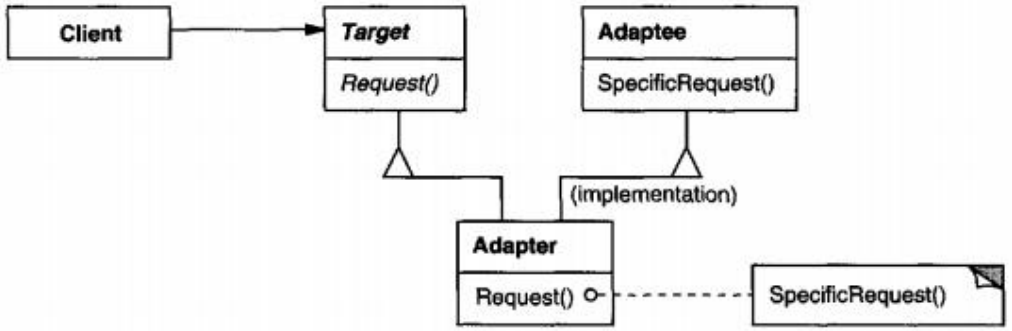
在一開始就先建造好實例
反正反覆存取一定會用到
需要的時候直接回傳同一個

6. 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

Convert the interface of a class into another interface clients expect.
Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
將類的接口轉換為客戶期望的另一個接口。 適配器由於不兼容的接口，類無法協同工作。

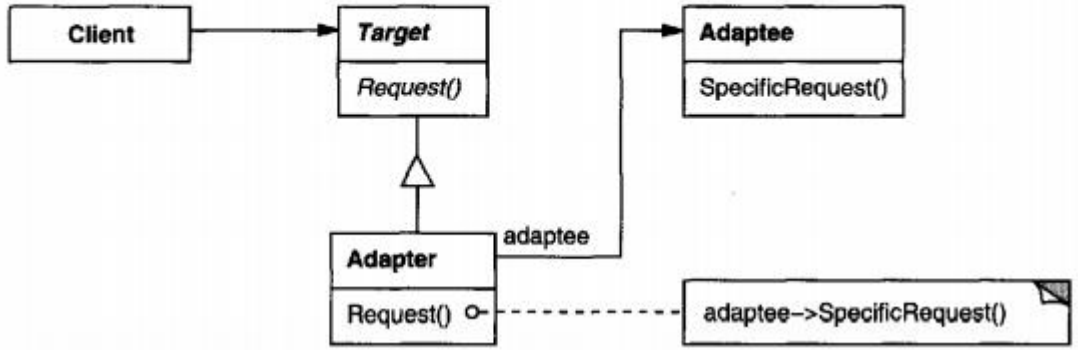
A class adapter uses multiple inheritance to adapt one interface to another:



Class的方式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來做成Target的方法

【Java不支援多重繼承故不討論】

An object adapter relies on object composition:



Object方式簡單來說就是透過aggregation
把Adaptee放在Adapter中繼承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

```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

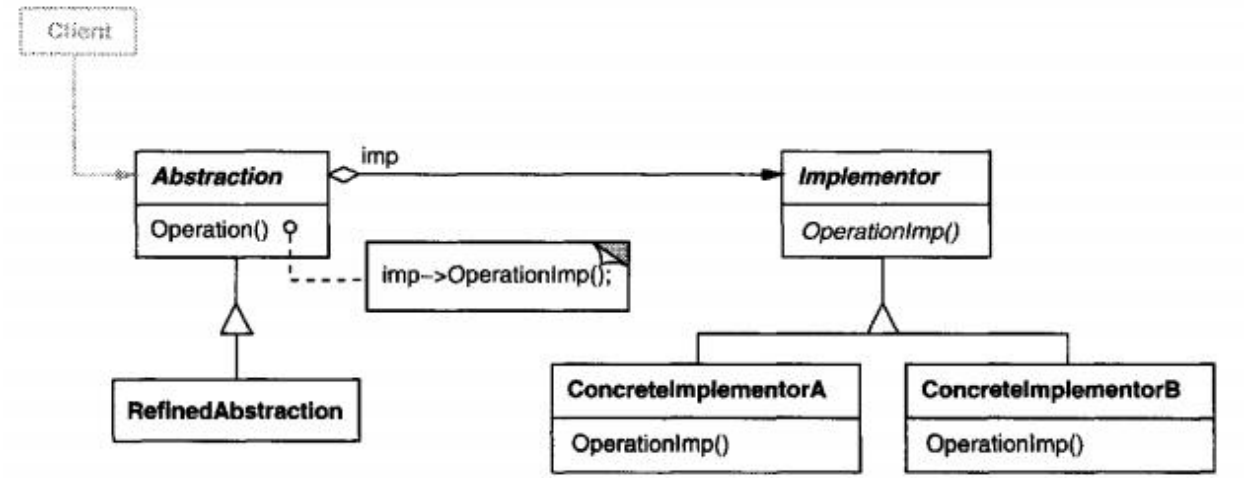
```java
public class AdapterDemo{
    public static void main(String[] arg){
        new Target().request();
        new Adapter().request();
    }
}
```

```
我是Target.
我是Adaptee裝成的Target.
```

7. Bridge 橋接模式 ***

Decouple an abstraction from its implementation so that the two can vary independently.

將抽象與其實現分離，以使兩者可以獨立變化。



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

Strategy是Behavioral pattern
強調的是/使用者可以選擇怎樣的方式去做

Bridge是Structural pattern
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:
    }
}

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:
    }
}
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);
    }
}
```

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

架構的地方
在裡面放置一個DrawAPI

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

根據放進來的API做對應得畫圖方法

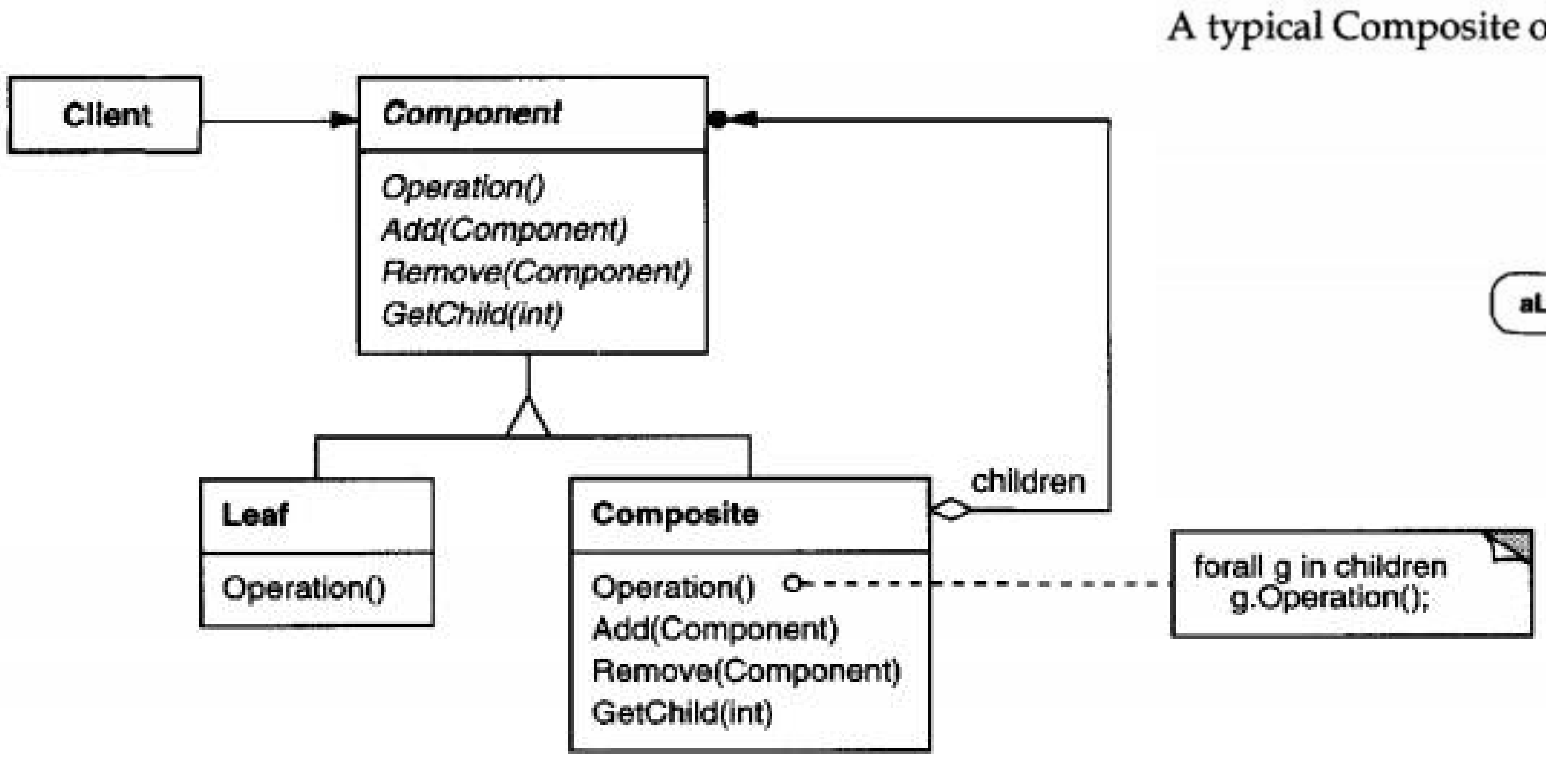
```java
public class BridgePattern {
    public static void main(String[] args) {
        Shape redCircle = new Circle(100,100, 10, new RedCircle());
        Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

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

```
畫個圓[ 顏色: 紅色, radius: 10, x: 100, 100]
畫個圓[ 顏色: 綠色, radius: 10, x: 100, 100]
```

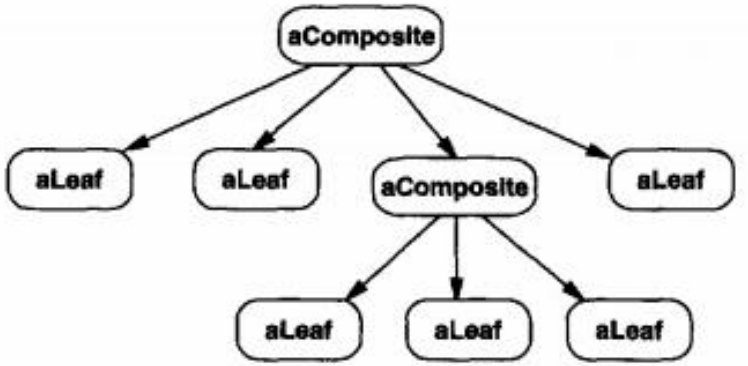
8. Composite 組合模式

Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.

將對象組織成樹狀結構產生出階層關係。讓外界一致性(都視為Component)對待個別類別物件和組合類別物件。



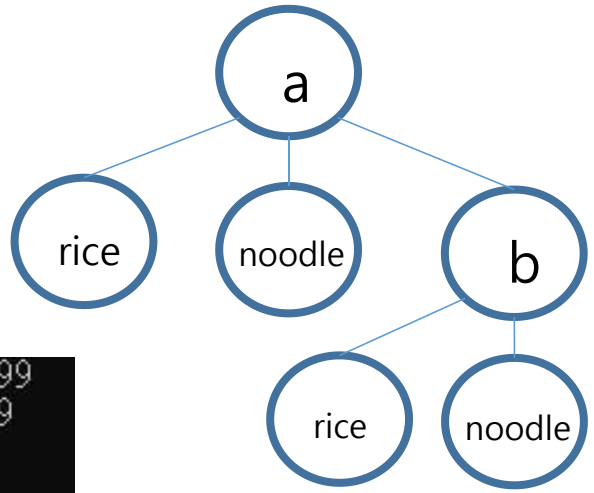
A typical Composite object structure might look like this:



```java
abstract class MenuComponent {
   public void add(MenuComponent m) {}
   public void remove(MenuComponent m) {}
   public String getName() {return "error";} //no void need return
   public double getPrice() {return 0;}      //no void need return
   public void print() {}
}

class Menu extends MenuComponent {  //composite
   ArrayList arr = new ArrayList();
   String name;
   public void add(MenuComponent m) {
      arr.add(m);
   }
   public void remove(MenuComponent m) {
      arr.remove(m);
   }
   public void print() {
      Iterator iterator = arr.iterator();
      while (iterator.hasNext()) {
         MenuComponent menuComponent = (MenuComponent)iterator.next();
         menuComponent.print();
      }
   }
}
```

```java
public class main {
   public static void main(String args[]) {
      MenuComponent a = new Menu();
      a.add(new rice("A_pancakeHouse", 1.99));
      a.add(new noodle("B_pancakeHouse", 2.99));
      MenuComponent b = new Menu();
      b.add(new rice("X_dinerMenu", 3.99));
      b.add(new noodle("Y_dinerMenu", 4.99));

      a.add(b);   //must have this!!!
      a.print();  //print top a

   }
}
```

```
rice     A_pancakeHouse , 1.99
noodle   B_pancakeHouse , 2.99
rice     X_dinerMenu , 3.99
noodle   Y_dinerMenu , 4.99
```



1.先作好方法，防止Leaf使用到功能時不會錯誤 可以作空→{}

2.Leaf (rice,noodle)

3.建立 composite (menu) ，組合需要ArrayList儲存多個元件，

因為Composite、Leaf 本身繼承，都可以視為Component

4.成功建立樹

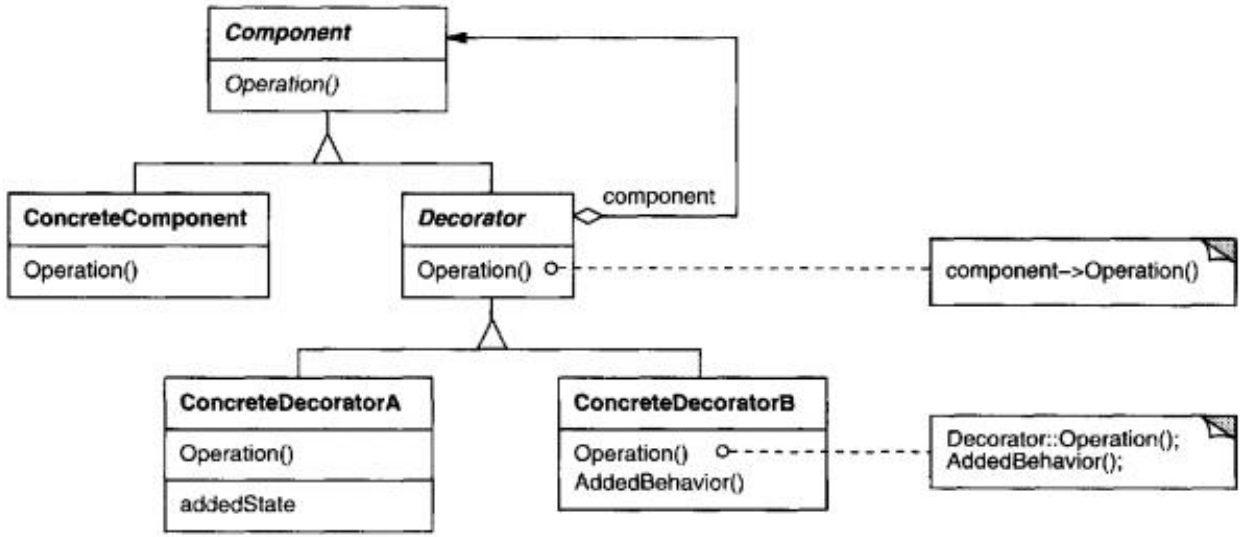
```java
import java.util.*;
import java.util.Iterator;
import java.util.ArrayList;
```

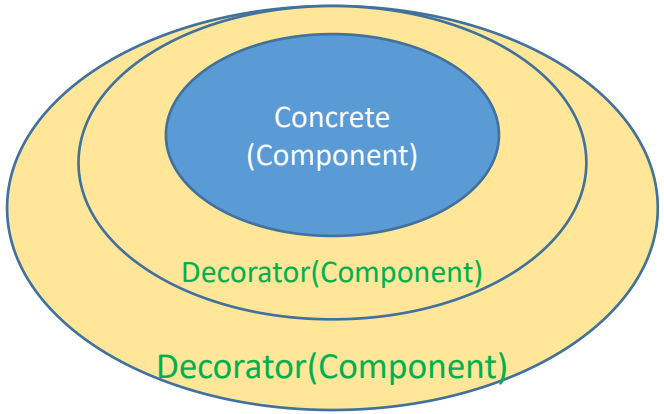
```java
class rice extends MenuComponent {
   String name;
   double price;
   public rice(String name, double price) {
      this.name = name;
      this.price = price;
   }
   public String getName() {return name;}
   public double getPrice() {return price;}
   public void print() {
      System.out.println("rice   " + getName() +" , "+getPrice());
   }
}

class noodle extends MenuComponent {
   String name;
   double price;
   public noodle(String name, double price) {
      this.name = name;
      this.price = price;
   }
   public String getName() {return name;}
   public double getPrice() {return price;}
   public void print() {
      System.out.println("noodle " + getName() +" , "+getPrice());
   }
}
```

9. Decorator 裝飾模式 ***

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

動態地將附加職責附加到對象。 裝飾者提供了一個靈活的子類化替代方法，用於擴展功能。





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

讓原本的Component很容易擴充而不用修改原本寫好的

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

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

```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;
    }
}
```

```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;
    }
}
```

```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());
```

1.每個餐點都有取得內容跟價錢兩個方法
2.Abstract Decorator 要包裝餐點所以裡面放置一個Meal
3.Concrete component (被包裝的)不能包裝別人放在最底層

4.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)

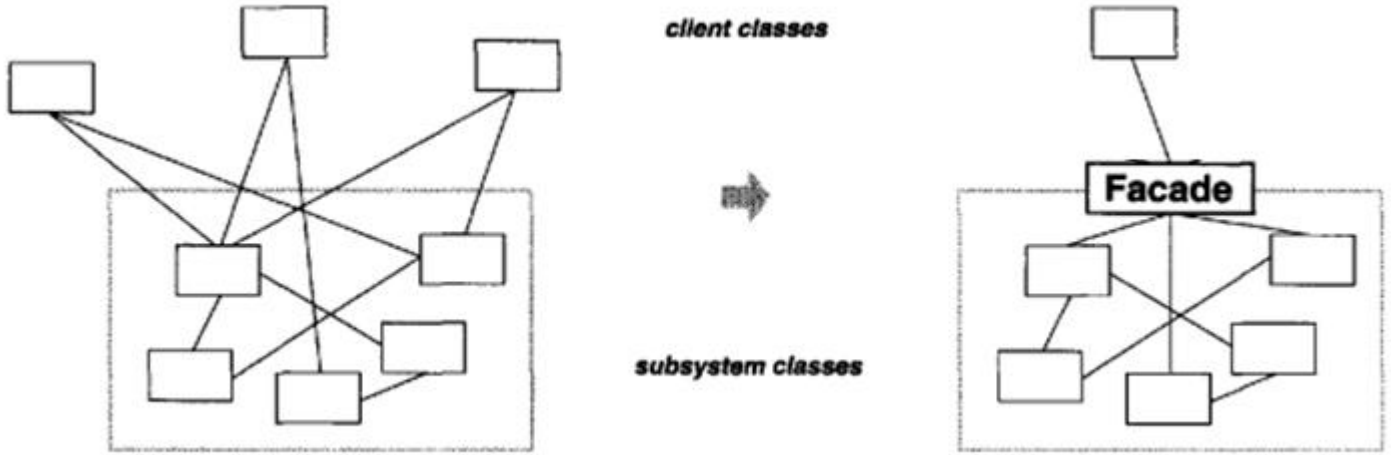
有點類似遞迴的概念

```
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
```

10. Facade 外觀模式

Provide a unified interface to a set of interfaces in a subsystem.
Facade defines a higher-level interface that makes the subsystem easier to use.

為子系統中的一組接口提供統一接口。 Facade定義了一個更高級別的界面，使子系統更易於使用。



- 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

1.Façade介面裡面包含所有子系統

```
class Computer {
    private CPU cpu;
    private Memory memory;
    private Disk disk;

    public Computer(){
        cpu = new CPU();
        memory = new Memory();
        disk = new Disk();
    }
    public void startup(){
        System.out.println("start the computer!");
        cpu.startup();
        memory.startup();
        disk.startup();
    }
}
```

2.總共三個子系統

```
class CPU {
    public void startup(){
        System.out.println("-cpu startup!");  }
}
class Memory {
    public void startup(){
        System.out.println("-memory startup!");}
}
class Disk {
    public void startup(){
        System.out.println("-disk startup!");  }
}

public class main{
    public static void main(String[] args) {
        Computer com = new Computer();
        com.startup();
    }
}
```

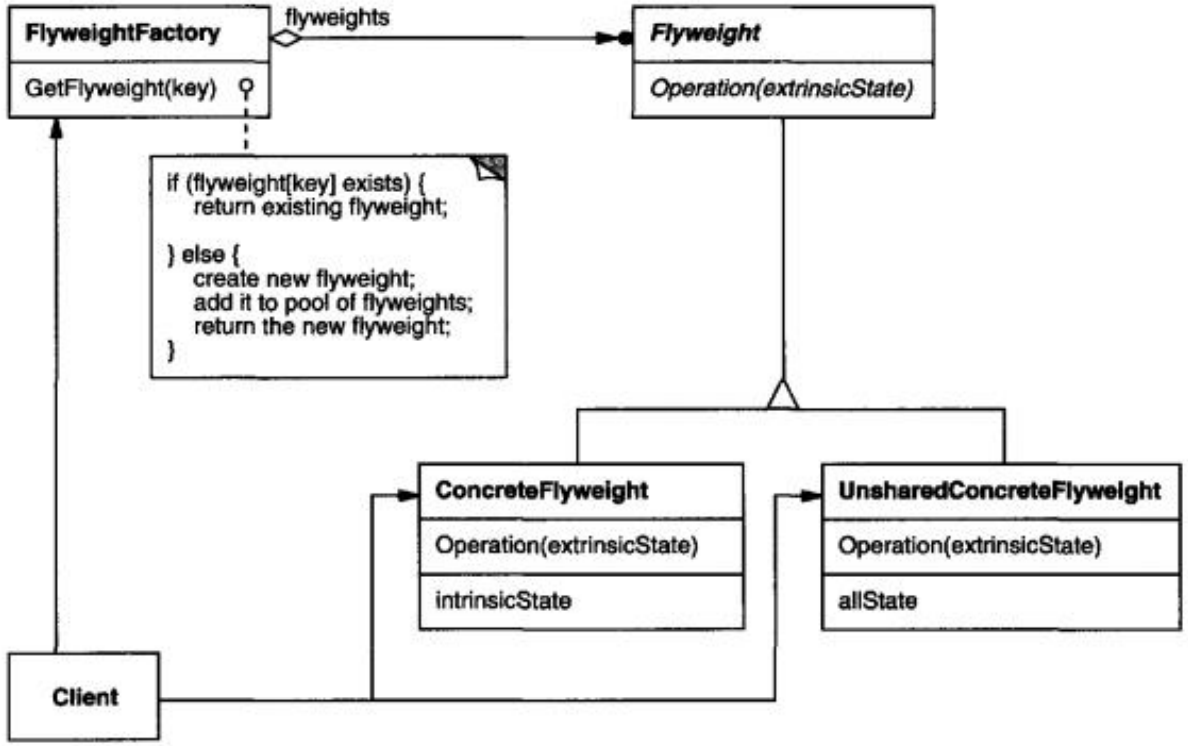
3.客戶端只會使用到Façade去操作
(我按下開機，後面怎麼執行不重要)

```
start the computer!
-cpu startup!
-memory startup!
-disk startup!
```

11. Flyweight 享元模式 ***

Use sharing to support large numbers of fine-grained objects efficiently.

使用共享可以有效地支持大量細粒度對象。



共享物件，用來盡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(所有具體享元的父類別)
  為這些類規定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。
- 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

```
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
使用時可以共享出去

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存放Flyweight物件

要取得Flyweight時先從HashTable裡面找
找不到再建造一個新的，減少記憶體空間的使用

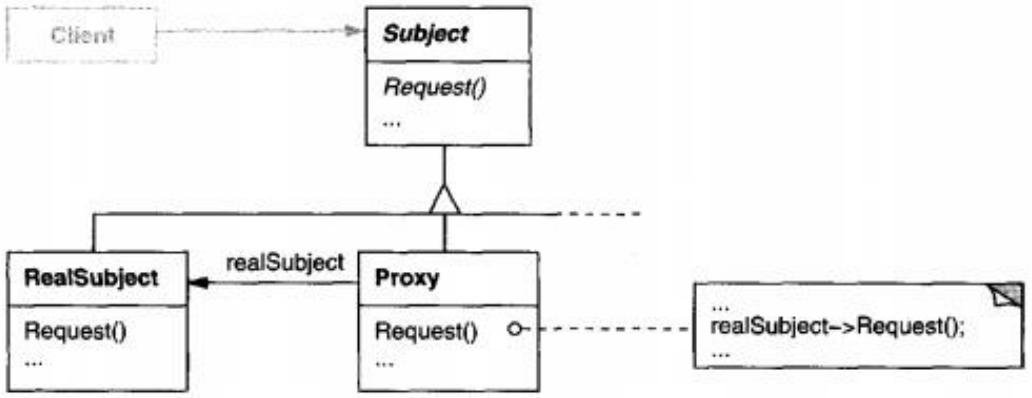
```
FlyweightFactory factory=new FlyweightFactory();

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

```
red A
```

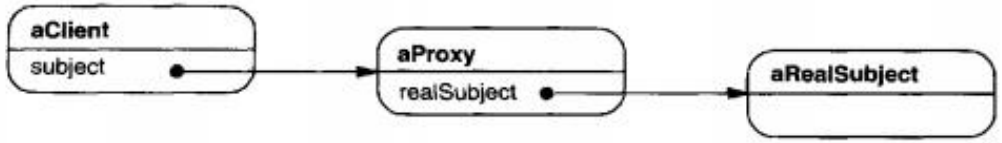
12. Proxy 代理模式***

Provide a surrogate or placeholder for another object to control access to it.

為另一個對象提供代理或占位符以控制對它的訪問。



Here's a possible object diagram of a proxy structure at run-time:



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(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(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(Protect Proxy)
  限制其他程式存取權限
- 智能(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(display)

```
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

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

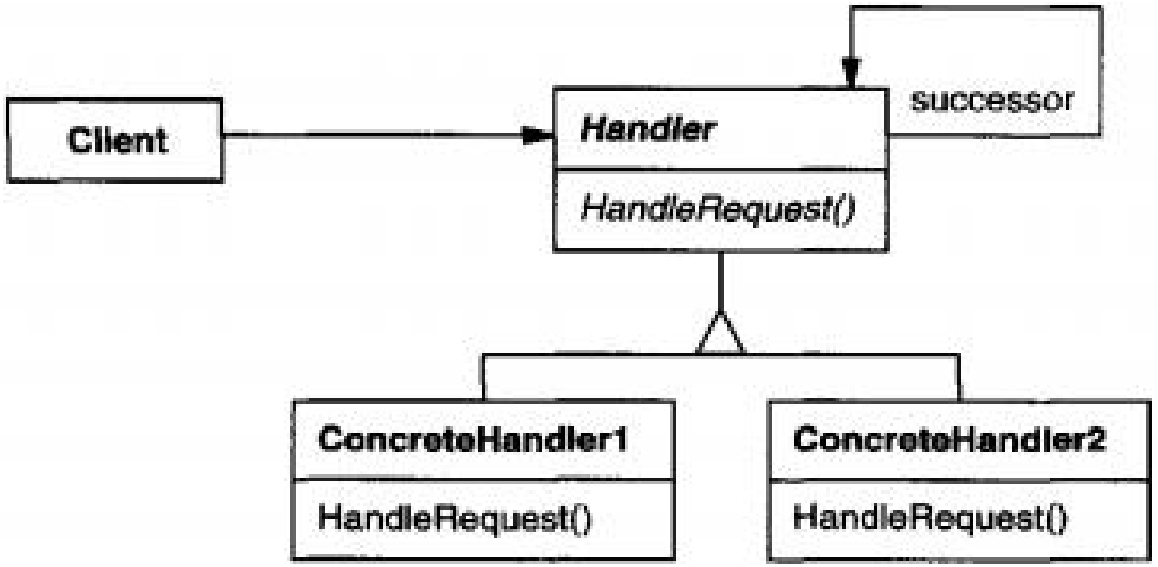
```
Loading test.jpg
Displaying test.jpg

Displaying test.jpg
```

我們就可以透過Proxy來讀取
而不用操作真正的Image

13. Chain of Responsibility 責任鏈模式

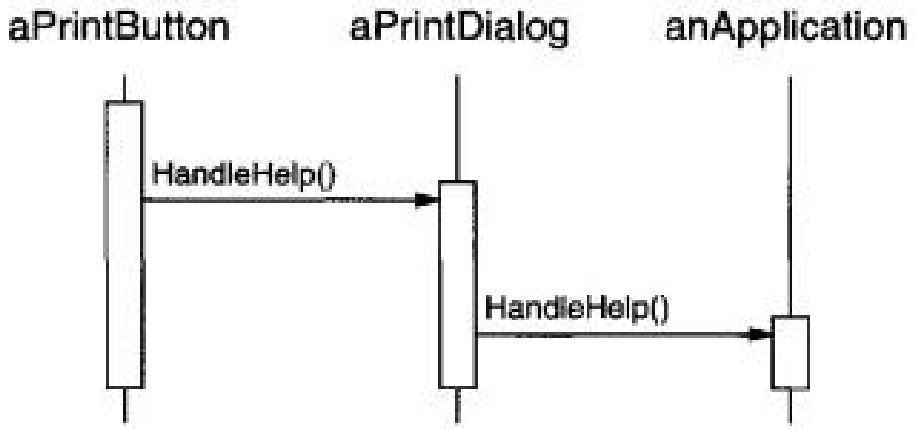
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

避免通過提供多個請求將發送者的請求與其接收者耦合對象有機會處理請求。
鏈接接收對象並通過沿鏈請求，直到對象處理它。



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



```java
abstract class Helper{
    Helper next;
    Helper(Helper next){
        this.next=next;
    }
    abstract void help(int m);
    public void doNext(int m){
        if(next !=null){
            next.help(m);
        }
    }
}

class h_10 extends Helper{
    public h_10(Helper next){
        super(next); }
    public void help(int m){
        if(m>=10){
            System.out.println("10 = "+(m/10));
        }
        doNext(m%10);
    }
}

class h_1 extends Helper{
    public h_1(Helper next){
        super(next); }
    public void help(int m){
        if(m>=1){
            System.out.println("1 = "+(m/1));
        }
        doNext(m%1);
    }
}
```

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

2.判斷下一個Helper 是不是Null
不是就交給下一位 next.help

3.零錢處理器現在這個是處理10的，只負責10塊
用取餘數，把10取完後交給next

4.把所有Helper串起，沒有下一個就放null

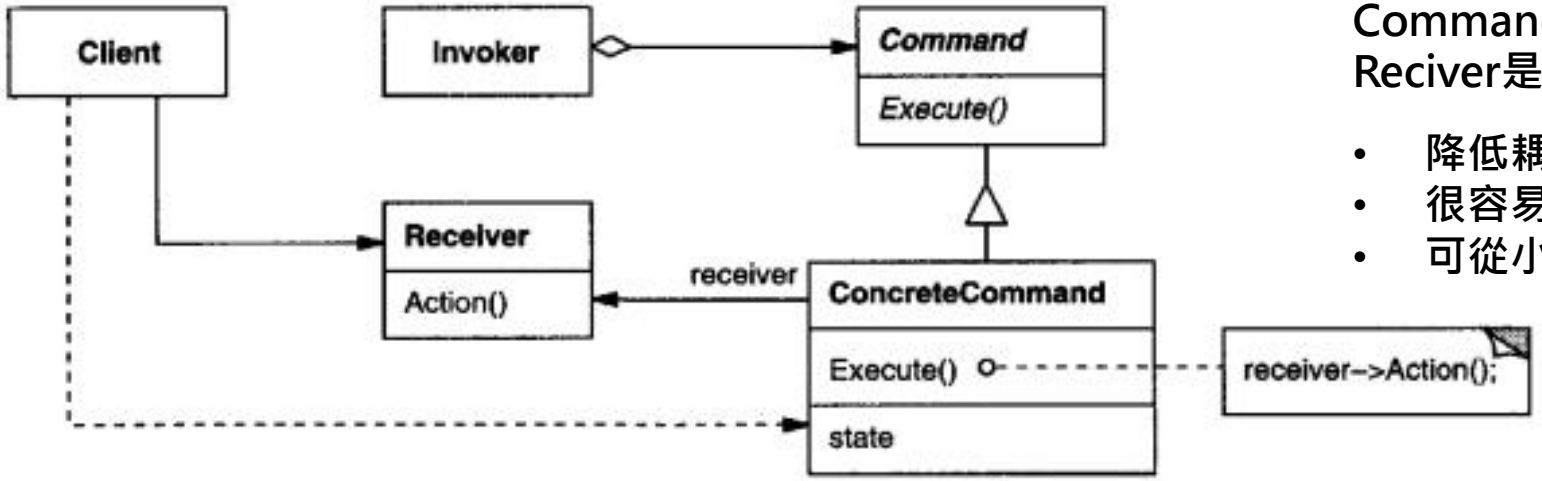
```java
public class main{
    public static void main(String[] args) {
    Helper h=new h_10(new h_1(null));
    h.help(1234);
    }
}
```

```
10 = 123
1 = 4
```

14. Command 命令模式

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

將請求封裝為對象，從而允許您使用參數化客戶端不同的請求，隊列或日誌請求，並支持可撤銷的操作。



把常用、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
Invoker用來執行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
- 可從小命令(micro)組成大命令(combined)

- Client -> Invoker ->command -> Receiver
- 客人　　服務生　　點餐　　廚師

```
class Invoker {
    Command command;
    public void set(Command command) {
        this.command = command;
    }
    public void Press() {
        command.execute();
    }
}
//Command
interface Command {
    public void execute();
}
//ConcreteCommand
class On implements Command {
    Light light;
    public On (Light light) {
        this.light = light;
    }
    public void execute() {
        light.on();
    }
}
class Off implements Command {
    Light light;
    public Off (Light light) {
        this.light = light;
    }
    public void execute() {
        light.off();
    }
}
```

1.Invoker存放要執行的Command

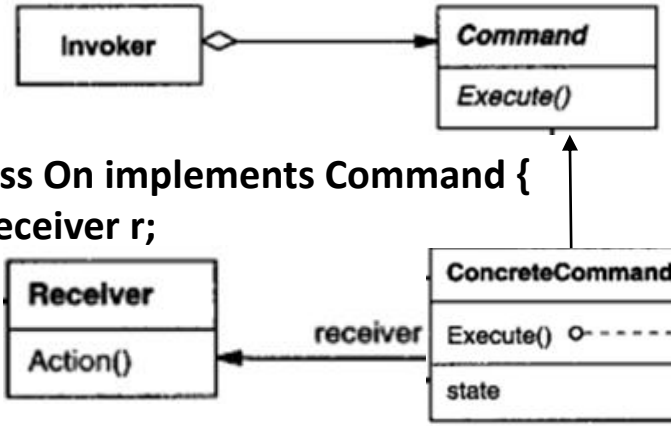
2.創造Command並把Reciver丟進去
Set進Invoker就可以執行Command

3.真正動作是Reciver執行

```
class Light {
    public Light() {}
    public void on() {
        System.out.println("Light is on");
    }
    public void off() {
        System.out.println("Light is off");
    }
}

public class main{
    public static void main(String[] args) {
        Light light = new Light();
        Command lights_on = new On(light);
        Command lights_off = new Off(light);
        Invoker i = new Invoker();

        i.set(lights_on);
        i.Press();
        i.set(lights_off);
        i.Press();
    }
}
```

```
LightOn.
LightOff.
```

```
class Invoker {
   Command command;
}
```

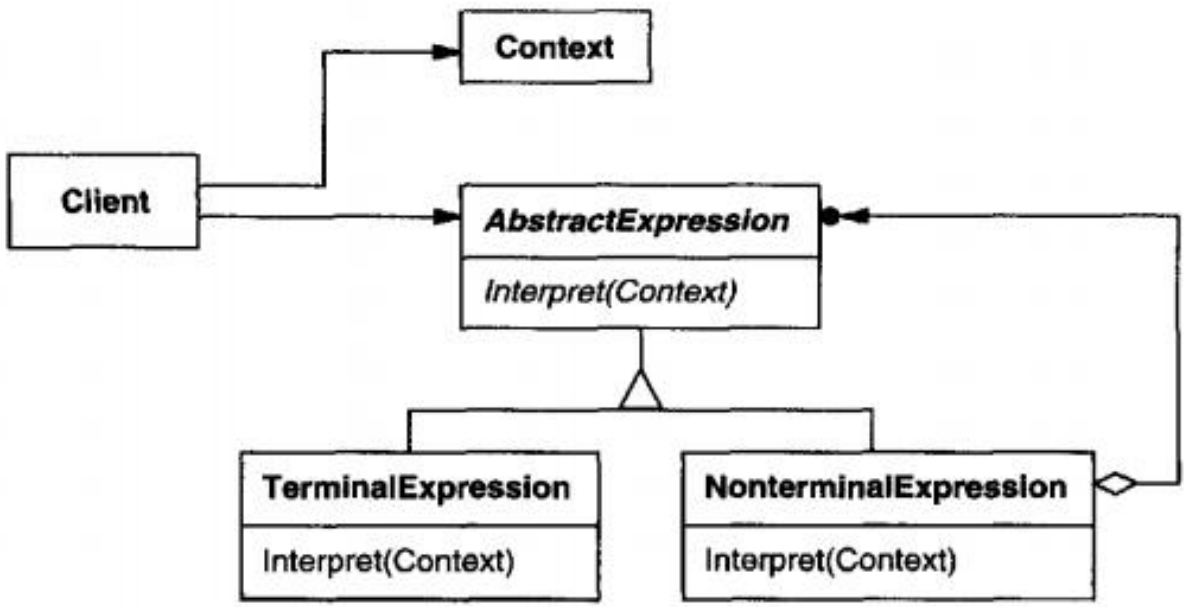



```
class On implements Command {
  Receiver r;
}
```

15. Interpreter 解釋器模式 ***

Given a language, define a represention forits grammar along with an interpreter
that uses the representation to interpret sentences in the language.

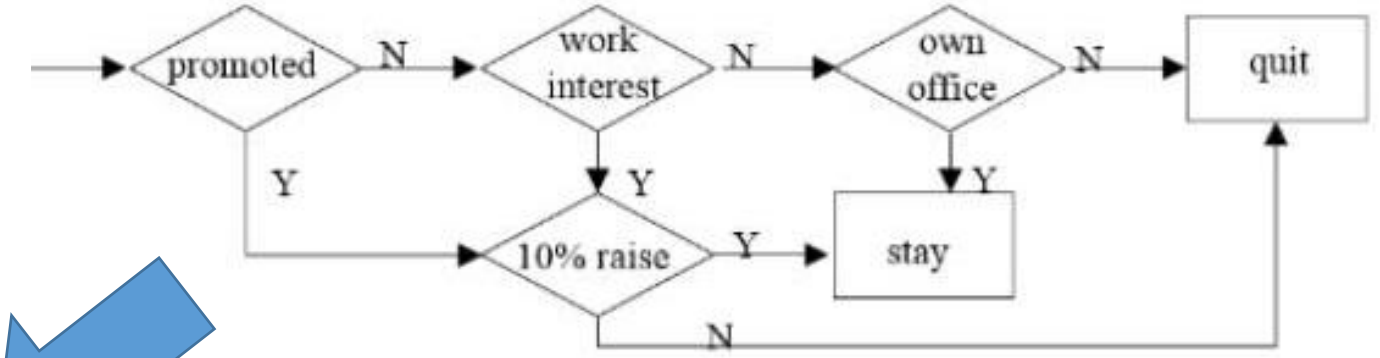
給定一種語言，定義其語法的表示以及解釋器
使用表示來解釋語言中的句子。

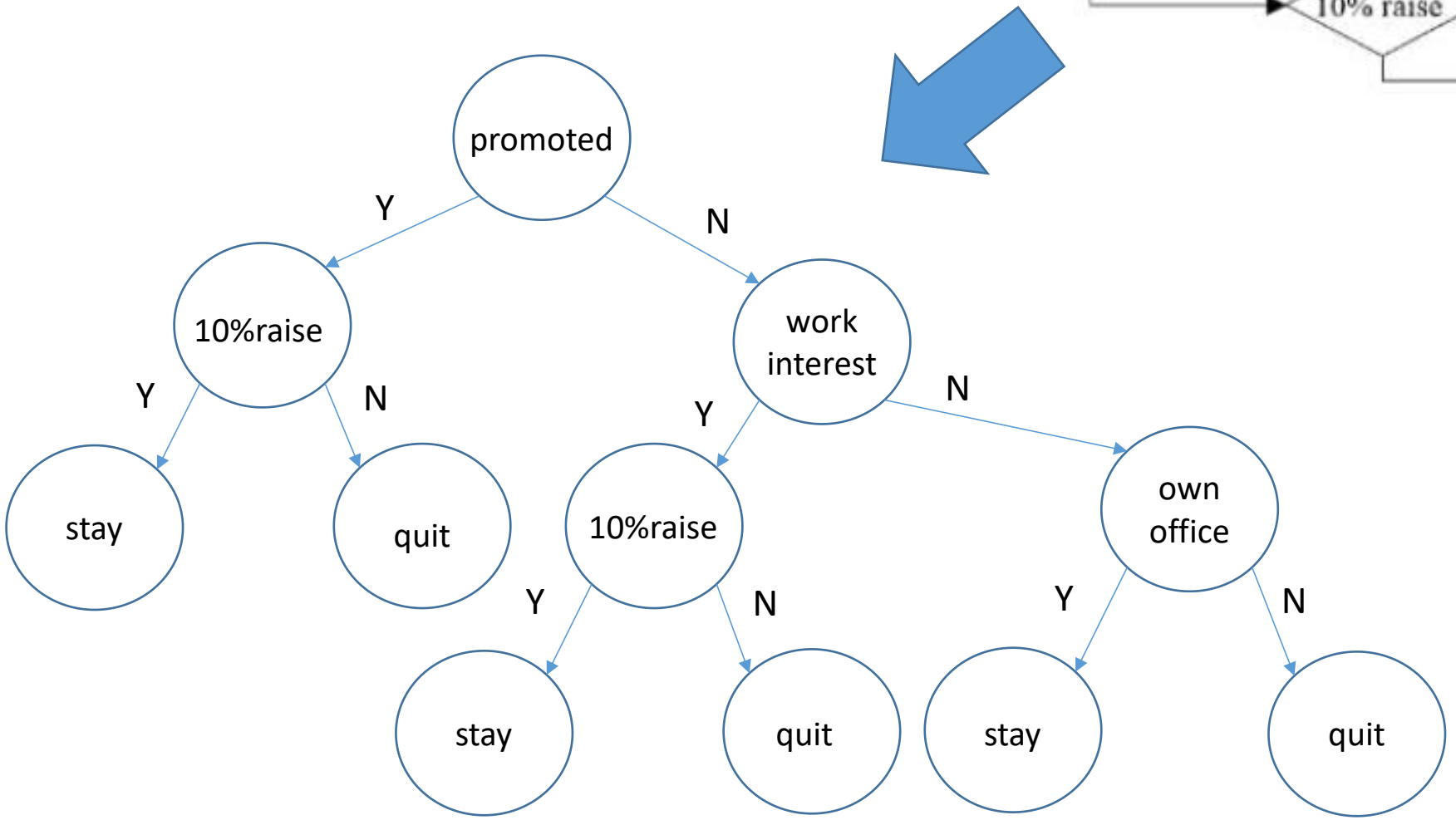


在處理一些複雜的問題上
我們希望可以透過分析器把問題丟進去
答案會自然產生

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、Quit)

```
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);
        }
    }
}
```

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

Terminal節點，輸出結果。

nonTerminal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。

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;
    }
}
```

4.輸入與執行

```
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.");
        }
    }
}
```

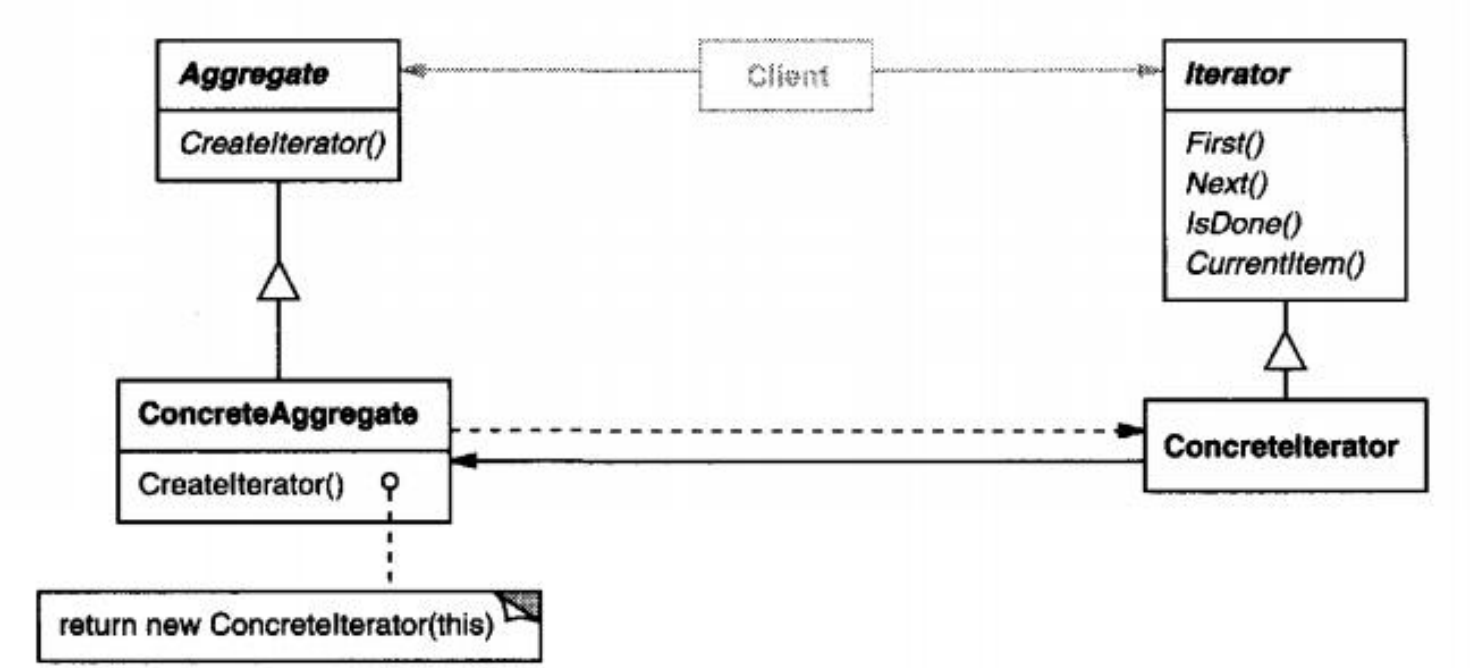
```
請輸入決策：(Ex:Y N - -)
N Y Y -
Stay.
Y Y - -
Stay.
```

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

16. Iterator 走訪器模式

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

提供一種順序訪問聚合對像元素的方法揭露其潛在的代表性。



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

```
import java.util.Iterator;
```

```
class Shape {
    private int id;
    private String name;
    public Shape(String name){
        this.name = name;}
    public String getName() {
        return name;}
    public void setName(String name) {
        this.name = name;}
    public String toString(){
        return " Shape: "+name;}
}

class ShapeStorage {
    private Shape[] shapes = new Shape[5];
    private int index;
    public void addShape(String name){
        int i = index++;
        shapes[i] = new Shape(name);
    }
    public Shape[] getShapes(){
        return shapes;
    }
}
```

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

```
class ShapeIterator implements Iterator<Shape>{
    private Shape[] shapes;
    int index;
    public ShapeIterator(Shape[]shapes){
        this.shapes = shapes;
    }
    public boolean hasNext() {
        if(index >= shapes.length)
            return false;
        return true;
    }
    public Shape next() {
        return shapes[index++];
    }
}

public class main {
    public static void main(String[] args) {
        ShapeStorage storage = new ShapeStorage();
        storage.addShape("Polygon");
        storage.addShape("Hexagon");
        storage.addShape("Circle");
        storage.addShape("Rectangle");
        storage.addShape("Square");

        ShapeIterator iterator = new ShapeIterator(storage.getShapes());
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
```

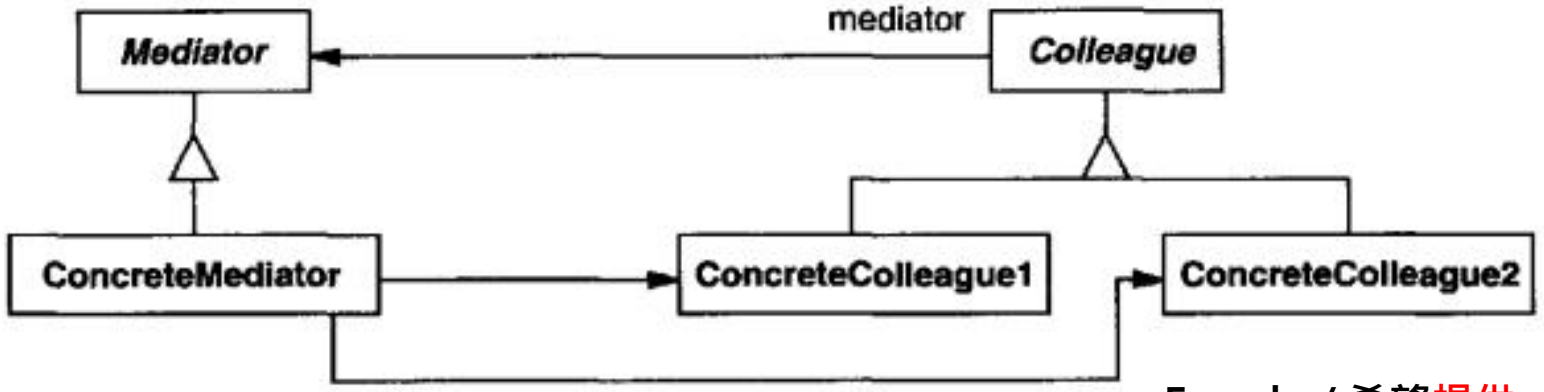
```
Iterator<DiagramElement> itr = des.iterator();
while (itr.hasNext()){
    DiagramElement e=itr.next();
    e.draw(g);
}
```

```
Shape: Polygon
Shape: Hexagon
Shape: Circle
Shape: Rectangle
Shape: Square
```

17. Mediator 中介者模式

Define an object that encapsulate show a set of objects interact.
Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

定義一個封裝顯示一組對象交互的對象。
Mediator通過保持對像明確地相互引用來促進鬆散耦合，它可以讓你獨立改變他們的互動。



Façade / 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(我按開機不管裡面)

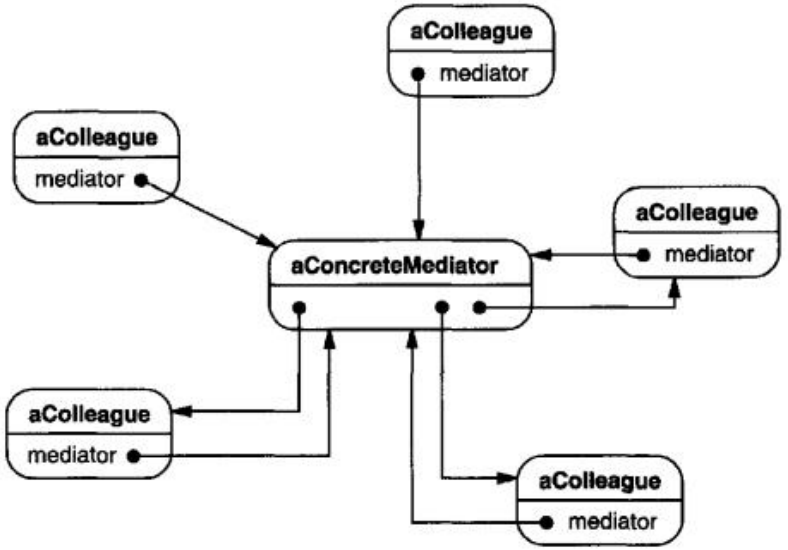
Mediator / 當系統內部之間運作非常複雜

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

```java
//colleague
class ChatRoom {
    public static void showMessage(User user, String message){
        System.out.println(" [" + user.getName() +"] : " + message);
    }
}
//mediator
class User {
    private String name;
    public String getName() {
        return name;
    }
    public User(String name){
        this.name   = name;
    }
    public void sendMessage(String message){
        ChatRoom.showMessage(this,message);     //this will catch who
    }
}
//client
public class main {
    public static void main(String[] args) {
        User robert = new User("Robert");
        User john = new User("John");
        robert.sendMessage("Hi! John!");
        john.sendMessage("Hello! Robert!");
    }
}
```

```
[Robert] : Hi! John!
[John] : Hello! Robert!
```

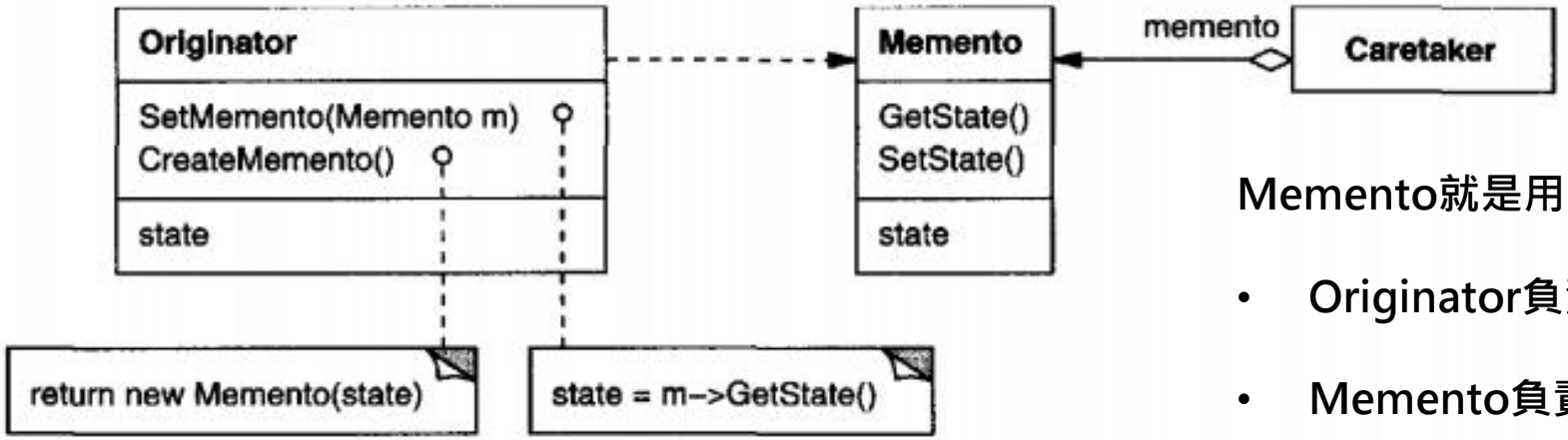
把原本彼此錯亂的溝通透過一個中介者來管理
把直接的溝通交給中介者來轉達
Mediator和Colleague之間會互相知道



18. Memento 備忘錄模式

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

在不違反封裝的情況下，捕獲並外化對象的內部狀態
以便稍後可以將對象恢復到此狀態。



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

```
import java.util.ArrayList;
import java.util.List;
```

```
class Memento {
    private String state;
    public Memento(String state){
        this.state = state;
    }
    public String gets(){
        return state;
    }
}

class Originator {  //tip
    private String state;
    public void sets(String state){
        System.out.println("Memento sets:" + state);
        this.state = state;
    }
    public String gets(){
        return state;
    }

    public Memento saves(){
        System.out.println("Memento save:");
        return new Memento(state);
    }
    public void getlist(Memento Memento){
        state = Memento.gets();
        System.out.println("Memento:"+state);
    }
}
```

1. **Memento 備份state (便條紙)**
2. **Originator建造memento (紙上的內容)**
3. **Caretaker 儲存這些備份 (便條紙一疊)**

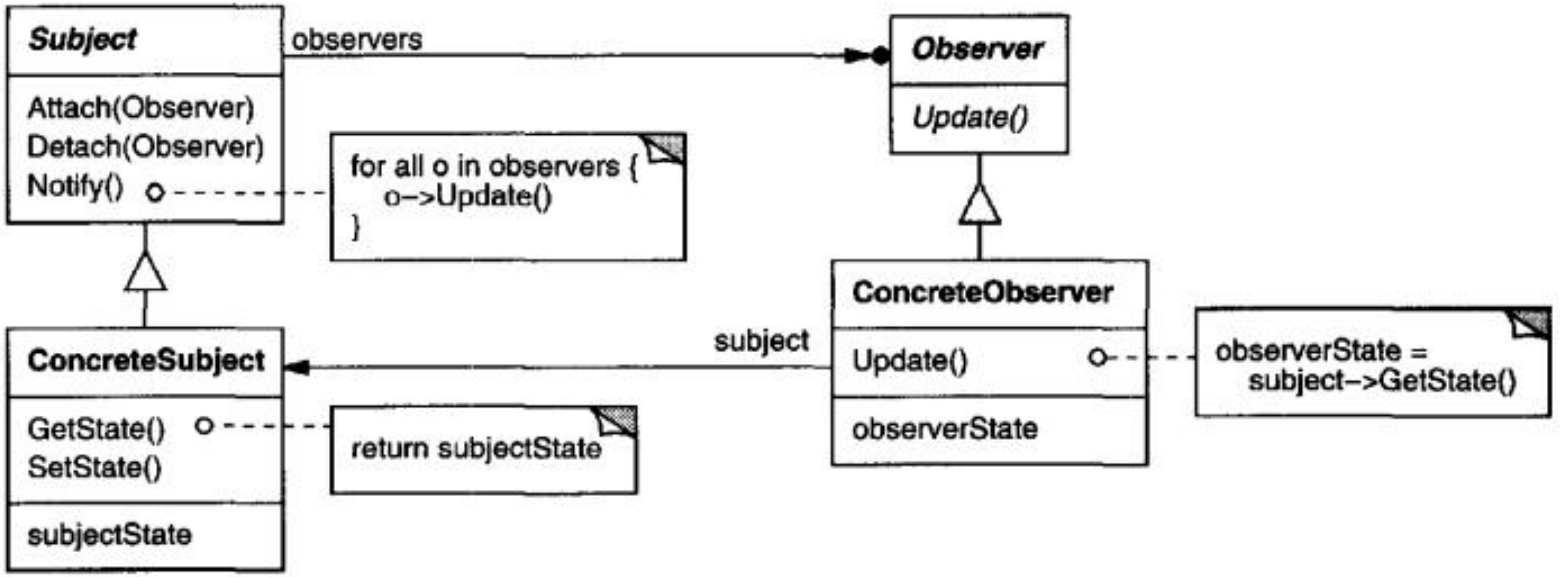
```
class CareTaker {   //tipssss
    private ArrayList<Memento> List = new ArrayList<Memento>();
    public void add(Memento state){
        List.add(state);
    }
    public Memento get(int index){
        return List.get(index);
    }
}

public class main{
    public static void main(String[] args) {
        Originator originator = new Originator();
        CareTaker careTaker = new CareTaker();
        originator.sets("State #1");
        originator.sets("State #2");
        careTaker.add(originator.saves());
        originator.sets("State #3");
        careTaker.add(originator.saves());
        originator.sets("State #4");

        originator.getlist(careTaker.get(0));
    }
}
```

```
Memento sets:State #1
Memento sets:State #2
Memento save:
Memento sets:State #3
Memento save:
Memento sets:State #4
Memento:State #2
```

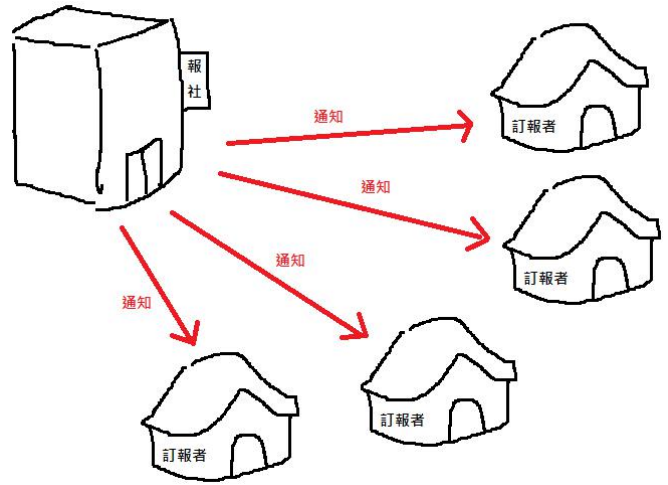
19. Observer 觀察者模式

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

定義對象之間的一對多依賴關係，以便一個對象改變狀態時，
所有家屬都會自動得到通知和更新。



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



import java.util.ArrayList;
import java.util.List;

```java
import java.util.ArrayList;
import java.util.List;

abstract class Observer {
    public abstract void update(String msg);
}
class Observer_1 extends Observer {
    public void update(String msg) {
        System.out.println("Youtube KanyeWest : " + msg);}
}
class Observer_2 extends Observer {
    public void update(String msg) {
        System.out.println("Youtube TaylorSwift :" + msg);}
}

//Youtuber
class Subject {
    private List<Observer> observers = new ArrayList<>();
    public void addAttach(Observer observer) {
        observers.add(observer);
    }
    public void notifyAll(String msg) {  //notify observer
        for (Observer observer : observers) {
            observer.update(msg);
        }
    }
}
```

1.用ArrayList儲存有訂閱這個
Subject的訂閱者(Observer)

2.用迴圈跑過所有ArrayList裡面的訂閱者
並把新的新聞通知她們更新

3.更新同時通知所有訂閱者更新

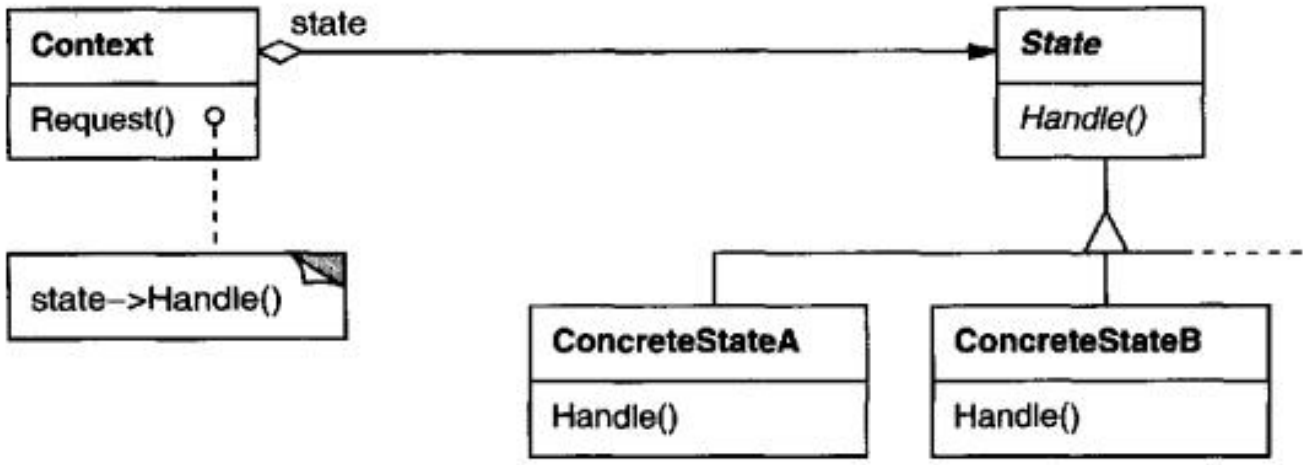
```java
public class main {
    public static void main(String[] args) {
        Observer_1 ob1 = new Observer_1();
        Observer_2 ob2 = new Observer_2();
        Subject subject = new Subject();
        subject.addAttach(ob1);
        subject.addAttach(ob2);
        subject.notifyAll("new video update");
    }
}
```

```
Youtube KanyeWest : new video update
Youtube TaylorSwift :new video update
```

20. State 狀態模式 ***

Allow an object to alter its behavior when its internal state changes.
The object will appear to change its class.

允許對像在其內部狀態更改時更改其行為。該對像似乎會更改其類。



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，「選擇貨品」的功能。
會因為顧客有沒有投錢、投多少錢，而有不同反應。

「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);
}
```

```
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
讓燈有sleep()得方法可以倒數

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

```
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
    }
}
```

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

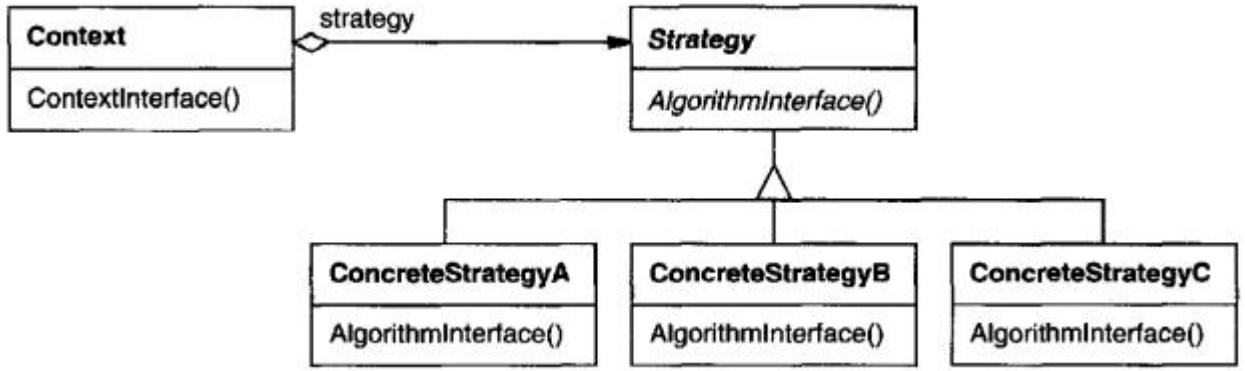
2.再來號誌燈會使用State的change方法
當State被改變，change的實際方法也改變
3.在Sleep完之後會重新把號誌燈Set到下一個綠燈狀態
1.迴圈內會一直呼叫號誌燈要Change

```
public class StatePattern {
    public static void main(String[] args) {
        TrafficLight trafficLight = new TrafficLight();
        while(true) {
            trafficLight.change();
        }
    }
}
```

21. Strategy 策略模式

Define a family of algorithms, encapsulate each one, and make them interchangeable.
Strategy lets the algorithm vary independently from clients that use it.

定義一系列算法，封裝每個算法，並使它們可互換。
策略允許算法獨立於使用它的客戶端。



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。

將實作的方法獨立出來成為一個Class，
透過繼承可以在不修改原程式碼的情況下擴充新程式碼

- 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();
}
class StrategyA implements Strategy{
    public void execute(){
        System.out.println("Using A");}
}
class StrategyB implements Strategy{
    public void execute(){
        System.out.println("Using B");}
}

class Context{
    Strategy strategy;
    public void set(Strategy s){
        strategy = s;
    }
    public void execute(){
        strategy.execute();
    }
}
```

1.先宣告一個Interface(或abstract) 底下實作

2.在需要切換運作方法的物件內建立Strategy物件

3.當Context物件被呼叫execute時
真正去執行的是Strategy

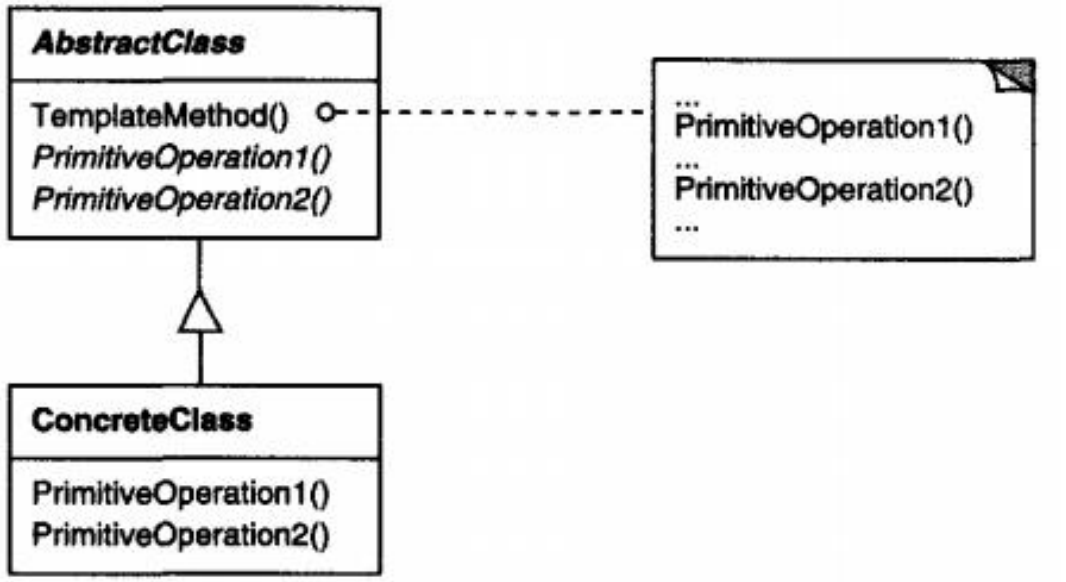
```java
public class main{
    public static void main(String[] args) {
        Context c = new Context();
        c.set(new StrategyA());
        c.execute();
        c.set(new StrategyB());
        c.execute();
    }
}
```

```
Using A
Using B
```

22. Template 樣板模式

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
Template Method letssubclassesredefine certain steps of an algorithm
without changing the algorithm's structure.

在操作中定義算法的骨架，將一些步驟推遲到子類。
模板方法letssubclasses重新定義算法的某些步驟而不改變算法的結構。



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掛勾決定要不要執行這一段
  2.網路上說Hook是一個預設為空的方法(Concrete)，子類別可以選擇是否覆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

```java
abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();
    public final void play(){
        initialize();
        startPlay();
        endPlay();
    }
}
class MapleStoy extends Game {
    void endPlay() {
        System.out.println("MapleStoy Finished!\n");}
    void initialize() {
        System.out.println("MapleStoy Initialized!");}
    void startPlay() {
        System.out.println("MapleStoy Started. Enjoy!");}
}
class GTA extends Game {
    void endPlay() {
        System.out.println("GTA Finished!\n");}
    void initialize() {
        System.out.println("GTA Initialized!");}
    void startPlay() {
        System.out.println("GTA Started. Enjoy!");}
}
```

1.先定義一套Abstract class 並固定好(final)版型

2.Contrete覆寫掉方法，執行不一樣的方法

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
public class main {
    public static void main(String[] args) {
        Game game = new MapleStoy();
        game.play();

        game = new GTA();
        game.play();
    }
}
```

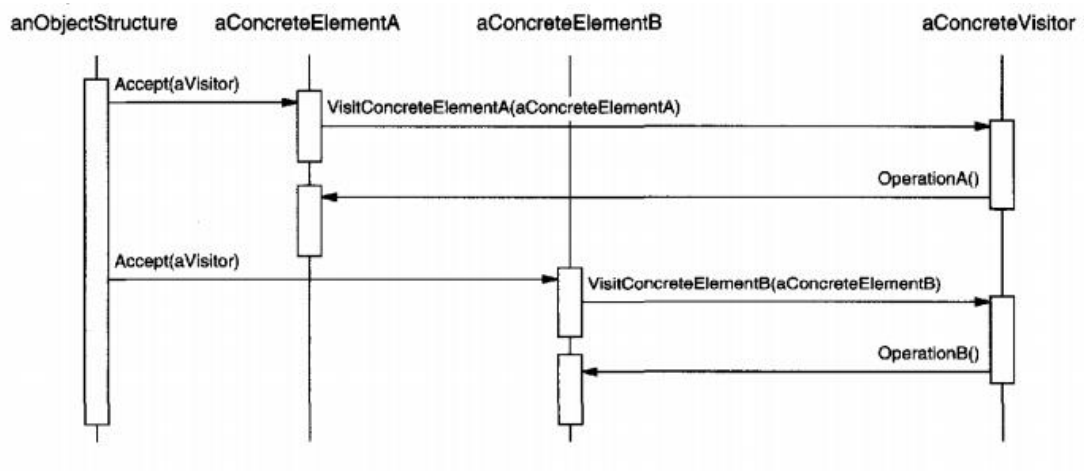
```
MapleStoy Initialized!
MapleStoy Started. Enjoy!
MapleStoy Finished!

GTA Initialized!
GTA Started. Enjoy!
GTA Finished!
```

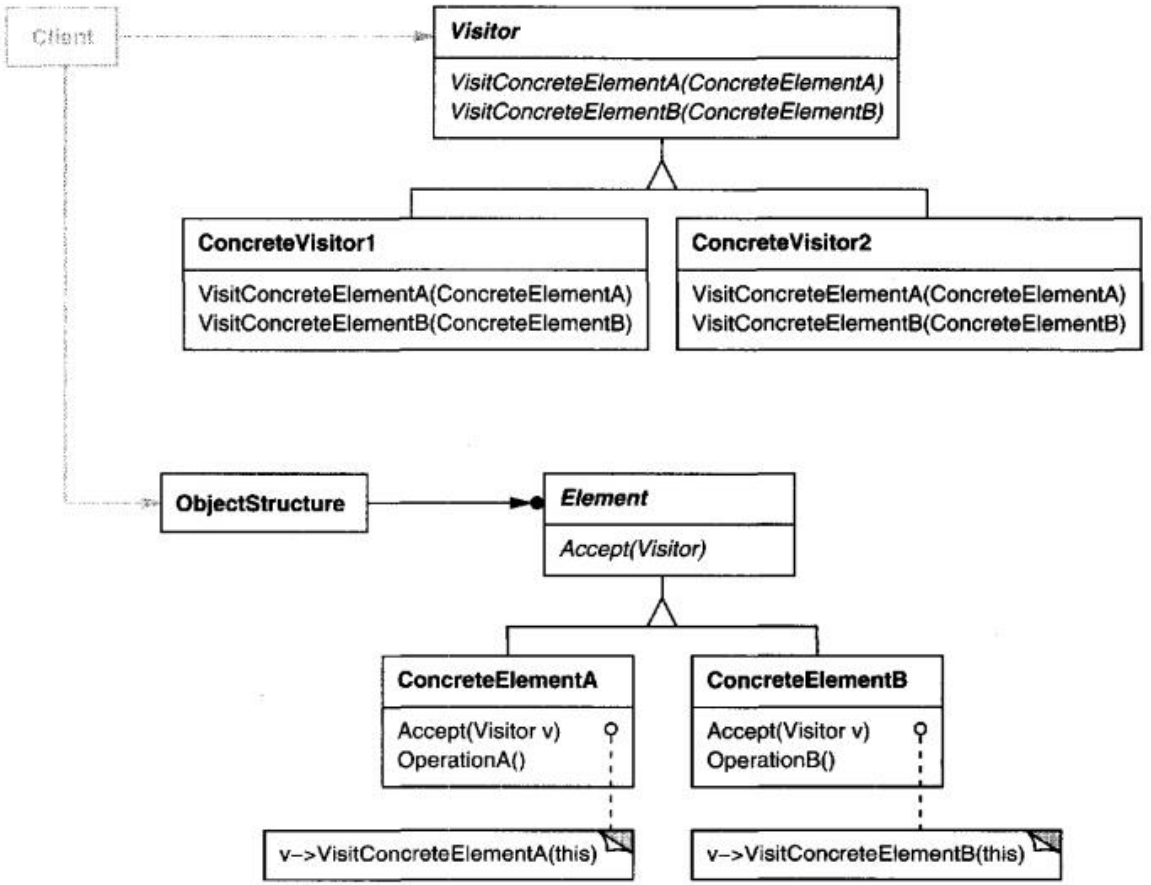
23. Visitor 參訪者模式

Represent an operation to be performed on the elements of an object structure.
Visitor lets you define a new operation without changing the classes of the elements on which it operates.

表示要對對象結構的元素執行的操作。
訪問者允許您定義新操作，而無需更改其操作的元素的類。

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





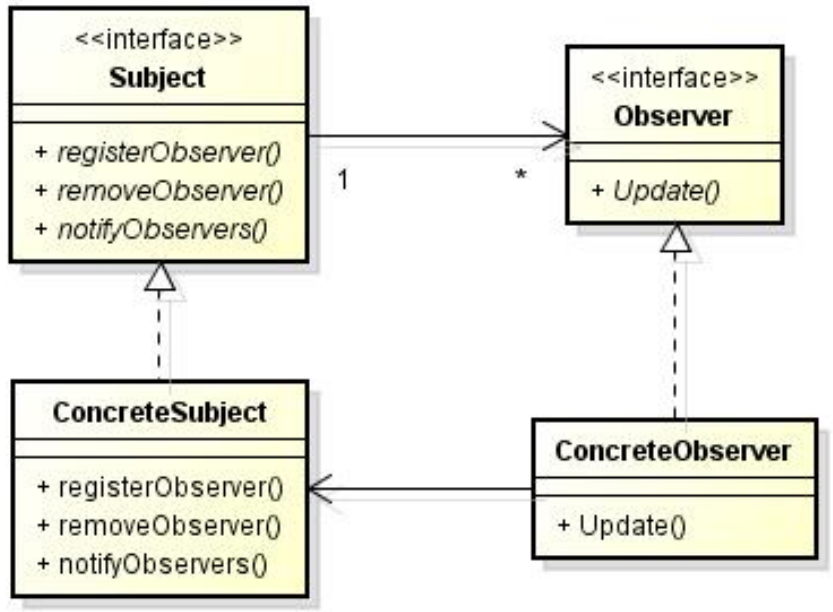
```java
interface Element {
    public void accept(Visitor visitor);
}
class Keyboard implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
class Mouse implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

interface Visitor {
    public void visit(Mouse mouse);
    public void visit(Keyboard keyboard);
}
class ConcreteVisitor implements Visitor {
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");}
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");}
}
```

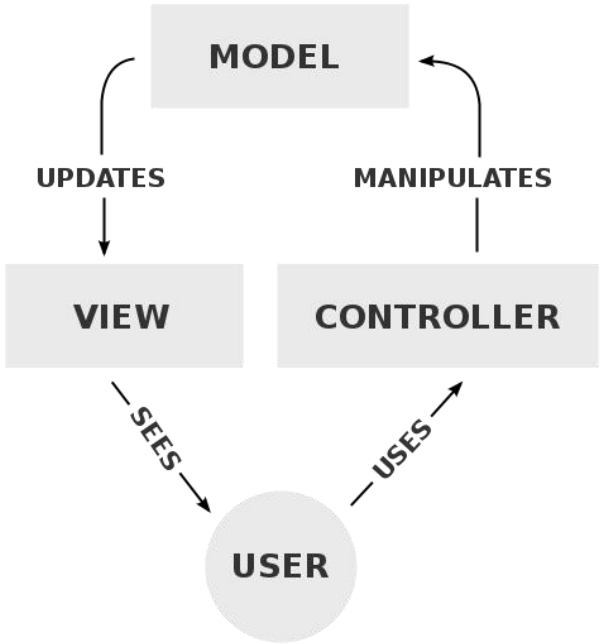
1.先把Element跟Visitor的方法定義好
Element只有accept的方法(Override)

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

```java
public class main {
    public static void main(String[] args) {
        Element element = new  Keyboard();
        element.accept(new ConcreteVisitor());

        element = new Mouse();
        element.accept(new ConcreteVisitor());
    }
}
```

24. Model-View-Control 模式





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

↓透過View的方法註冊進去

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


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

↓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);

    }
}
```

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

↓View裡面的方法，實際上是註冊到Button

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

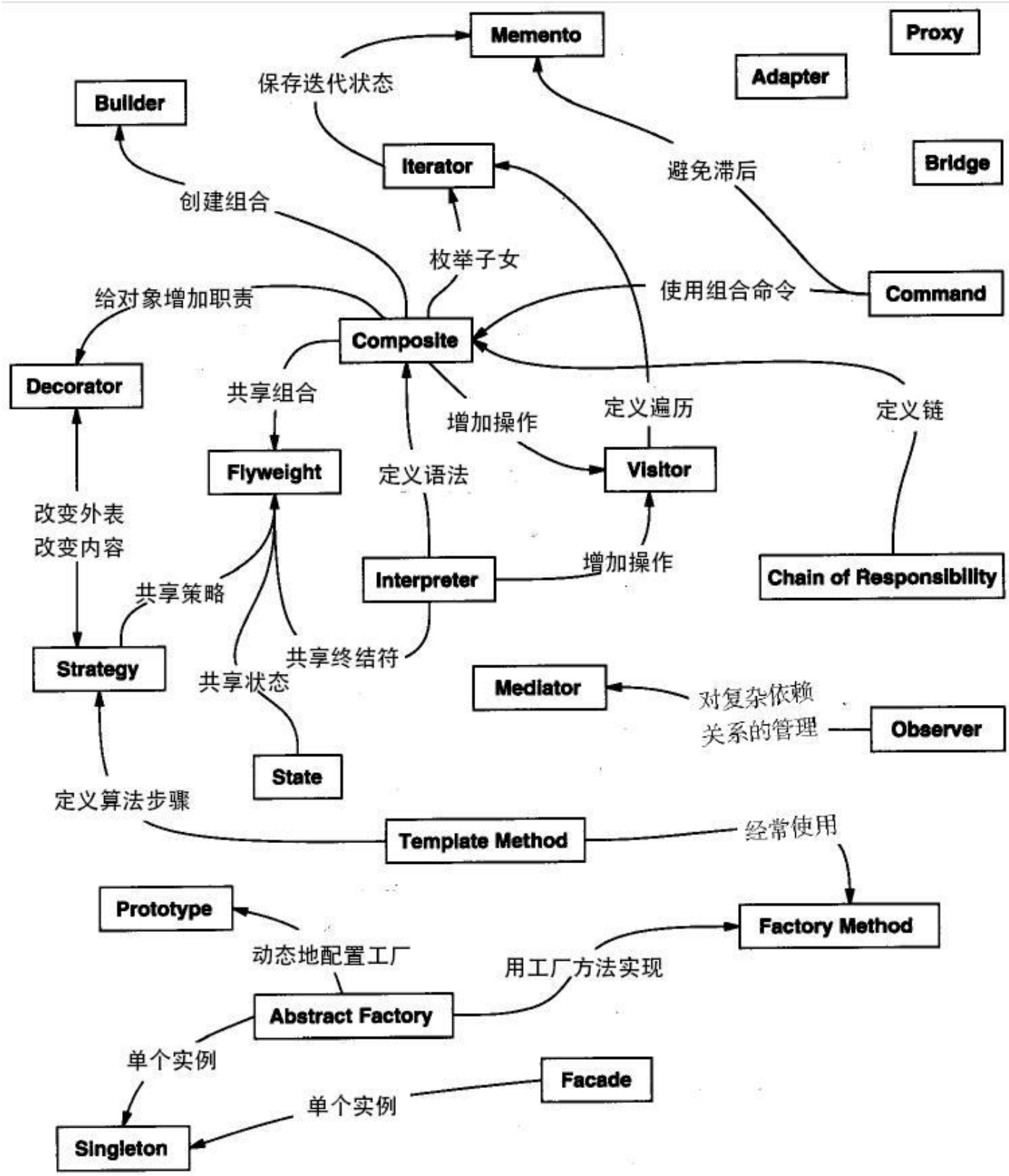

Memento
Proxy
保存迭代状态
Adapter
Builder
Iterator
避免滞后
Bridge
创建组合
枚举子女
给对象增加职责
使用组合命令
Command
Composite
Decorator
共享组合
定义遍历
定义链
增加操作
Flyweight
定义语法
Visitor
改变外表
改变内容
Interpreter
增加操作
Chain of Responsibility
共享策略
共享终结符
Strategy
共享状态
Mediator
对复杂依赖
关系的管理
Observer
State
定义算法步骤
Template Method
经常使用
Prototype
Factory Method
动态地配置工厂
用工厂方法实现
Abstract Factory
单个实例
Facade
单个实例
Singleton

图　设计模式之间的关系

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

Creational Patterns

Structural Patterns

BehavioralPatterns

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/