



題目:設計模式 for PHP

版本:1.3

撰者:Hank_Kuo

日期:2013.05.30

引言

目前公司所接觸的專案與開發項目規模越來越大，怎樣設計出可以滿足多變的修改與擴充需求的系統架構，已經變成一個重要的課題。而Design Pattern則是解決上述需求的一個解決方法。它是對軟體設計中普遍存在（反覆出現）的各種問題，所提出的解決方案。它可以運用在大多數物件導向的程式語言中，它能夠避免會引起麻煩的緊耦合，以增強軟體設計面對並適應變化的能力。

有鑑於目前市面上現有書籍資料較為艱澀難懂，網路資料較為鬆散雜亂。因此在筆者學習的Design Pattern的過程中，整理出本筆記，希望能夠幫助大家能夠不用花這麼多學習時間，就可以比較容易的理解Design Pattern，並且運用在實際的專案上。

由於本筆記的資料皆採自與市面出版書籍與網路資料，由於筆者本身能力有限，筆記裡面也許有說明不清楚或是不正確的地方，歡迎大家提出問題與為本筆記勘誤。信箱: hank7444@gmail.com

郭政翰(Hank Kuo)
2013年5月于台北市



Design Patterns書中23個設計模式

- Creational Patterns(生成模式)：
 - Simple Factory(簡單工廠模式)
 - Factory Method(工廠方法模式)
 - Abstract Factory(抽象工廠模式)
 - Builder(生成器模式)
 - Prototype(原型模式)
 - Singleton(單例模式)
- Structural Patterns(結構模式)：
 - Adapter(轉接器模式)
 - Bridge(橋梁模式)
 - Composite(合成模式)
 - Decorator(裝飾模式)
 - Façade(表象模式)
 - Flyweight(享元模式)
 - Proxy(代理人模式)
- Behavioral Patterns(行為模式)：
 - Chain of Responsibility(責任鍊模式)
 - Command(命令模式)
 - Interpreter(解釋器模式)
 - Iterator(迭代器模式)
 - Mediator(中介者模式)
 - Memento(備忘錄模式)
 - Observer(觀察者模式)
 - State(狀態模式)
 - Strategy(策略模式)
 - Template Method(樣板方法模式)
 - Visitor(訪客模式)

設計模式速查表(1/4)

Creational Patterns(生成模式)

Simple Factory(簡單工廠模式)

Implement a method or class to create objects based on specific parameters (the Context or product type). The Simple Factory chooses and instantiates the required class and returns the result to the caller.

Factory Method(工廠方法模式)

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

Abstract Factory(抽象工廠模式)

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

Builder(生成器模式)

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

Prototype(原型模式)

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

Singleton(單例模式)

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



設計模式速查表(2/4)

Structural Patterns(結構模式)

<u>Adapter</u> (轉接器模式)	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
<u>Bridge</u> (橋梁模式)	Decouple an abstraction from its implementation so that the two can vary independently.
<u>Composite</u> (合成模式)	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
<u>Decorator</u> (裝飾模式)	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
<u>Façade</u> (表象模式)	Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
<u>Flyweight</u> (享元模式)	Use sharing to support large numbers of fine-grained objects efficiently.
<u>Proxy</u> (代理人模式)	Provide a surrogate or placeholder for another object to control access to it.

設計模式速查表(3/4)

Behavioral Patterns(行為模式)

<u>Chain of Responsibility(責任鍊模式)</u>	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.
<u>Command(命令模式)</u>	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
<u>Interpreter(解釋器模式)</u>	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
<u>Iterator(迭代器模式)</u>	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
<u>Mediator(中介者模式)</u>	Define an object that encapsulates how 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.
<u>Memento(備忘錄模式)</u>	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

設計模式速查表(4/4)

Behavioral Patterns(行為模式)

<u>Observer</u> (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.
<u>State</u> (狀態模式)	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
<u>Strategy</u> (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.
<u>Template Method</u> (樣板方法模式)	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
<u>Visitor</u> (訪客模式)	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.

OOP三大元素(1/4)

- 封裝(Encapsulation)



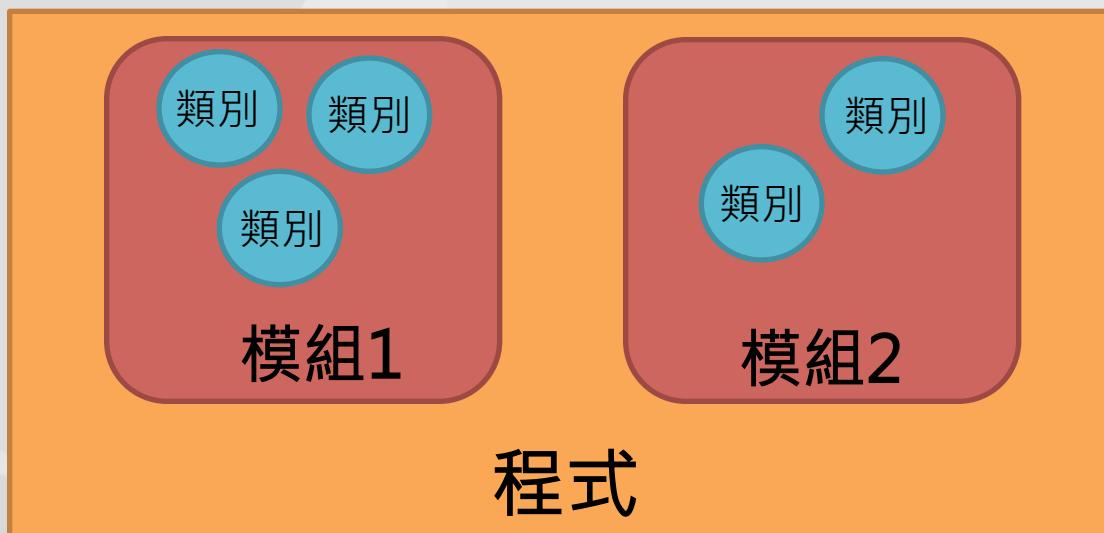
- 繼承(Inheritance)



- 多型(Polymorphism)

OOP三大元素(1/4)：封裝(1/2)

封裝 (Encapsulation) 的目的，是將程式碼切割成許多類別 (Class)，使各類別之間的關連性降到最低，這麼一來比較不會產生「牽一髮而動全身」的狀況，降低類別間相互依賴的程度，也等於是降低複雜度，讓開發與維護更容易。



在OO的世界，一般來說，一個程式有多個模組，一個模組內包含多個類別。

OOP三大元素(2/4)：封裝(2/2)

盡可能降低能見度，才能「降低互相依賴的程度」。別人不需要知道的，就不要讓它知道，這就是所謂的「資訊隱藏」（Information Hiding）。

封裝將相關的資料和使用到這些資料的方法包成類別。最理想的狀況是，讓資料能見度降到最低，外面完全看不見，留下的對外介面（Interface）只剩下Method。換句話說，每個物件的Interface是一些方法的集合，完全沒有資料。

能見度設定得太寬，造成資訊隱藏效果不佳，可能會帶來相當多負面效果（例如複雜度提高、程式容易出錯、非Thread-Safe.....等）；能見度設定得太緊，則造成效率變差、擴充程度變差（有些設計因而做不出來）

OOP三大元素(3/4)：繼承(1/1)

繼承的目的，是要達到「程式碼再用」(Code Reuse) 或「介面再用」。而繼承的手段，就是「擴充」或「修改」。

繼承所導致的程式碼再用，是指子類別能自動沿襲父類別的所有程式碼，好讓你可以不用寫太多程式碼，只需要稍微擴充或修改，就能符合你的需求。

「擴充」指的是定義新的方法 (Method) ；「修改」指的是針對父類別中的某方法重新定義其行為。

OOP三大元素(4/4)：多型(1/1)

繼承所導致的介面再用，是為OO的下一個階段（也就是多型）作準備。介面再用，搭配方法的修改，就形成了多型。

一個物件為何會有不同型別？這是因為繼承而來，物件可以扮演所有祖先類別的角色。例如當某物件為子類別，當此物件被「轉型」成父類別之後，此物件就具有兩種不同的類別：「實際類別」是子類別，「形式類別」是父類別。如果，在這個情況下呼叫此物件的方法m，會執行到父類別定義的方法m？還是子類別定義（修改）的方法m？

答案是實際類別的方法，也就是子類別定義的方法m。因此，所謂的**多型**就是：不管形式類別是什麼，一定會執行到實際類別的方法。

PHP多型範例(1/1)

```
abstract class Shape
{
    private $draw;

    function setDraw($draw) {
        $this->draw = $draw;
    }

    function getDraw() {
        return $this->draw;
    }
}

class Circle extends Shape
{
    function __construct() {
        $this->setDraw('畫圓圈');
    }
}

class Rectangle extends Shape
{
    function __construct() {
        $this->setDraw('畫長方形');
    }
}
```

圖型類別

```
class Drawer
{
    private $shapes = array();

    function addShape(Shape $shape) {
        array_push($this->shapes, $shape);
    }

    function drawAllShapes() {
        foreach($this->shapes as $shape) {
            echo $shape->getDraw() . '<br/>';
        }
    }
}
```

Drawer不知道
是Circle或Rectangle，
只管呼叫getDraw();

圖型閱讀器

```
$circle = new Circle();
$rectangle = new Rectangle();
$drawer = new Drawer();

$drawer->addShape($circle);
$drawer->addShape($rectangle);

$drawer->drawAllShapes();
```

畫圓圈
畫長方形

客戶端

輸出結果

(補充)PHP類別、虛擬類別、介面差異(1/9)

- 抽象類別(Abstract Class):
 - 在領域概念中無法具體化的類別
- 介面(Interface):
 - 一種可以附上在任意類別上的特徵，也是一種規範

(補充)PHP類別、虛擬類別、介面差異(2/9)

- 類別(Class):
 - 可以宣告屬性(attribute)成員
 - 可以擁有常數(const)成員
 - 可以實例化(instance)
 - 不可以有抽象方法(Abstract method)
 - 可以實作方法內容(從虛擬類別、介面繼承或定義的虛擬方法，都必需實作)
 - 子類別只可以繼承一個父類別
- 虛擬類別(Abstract Class):
 - 可以宣告屬性(attribute)成員
 - 可以擁有常數(const)成員
 - 不可以實例化(instance)
 - 至少一個抽象方法(Abstract method) 注:沒強制
 - 可以實作方法內容(不可實作抽象方法)

子類別只可以繼承一個虛擬類別



(補充)PHP類別、虛擬類別、介面差異(3/9)

- 介面(Interface):
 - 不可以宣告屬性(attribute)成員
 - 可以擁有常數(const)成員
 - 不可以實例化(instance)
 - 可以有抽象方法(**Interface所有方法都是抽象的**，只是不用宣告abstract關鍵字)
 - 不可以實作方法內容
 - 類別可以定義多個介面
 - 介面可以衍生自另一個介面

(補充)PHP類別、虛擬類別、介面差異(4/9)

	Class	Abstract Class	Interface
宣告屬性(attribute)	✓	✓	✗
常數(const)	✓	✓	✓
實例化	✓	✗	✗
抽象方法	✗	✓	✓
實作方法內容	✓	✓	✗
類別可否定義或繼承多個?	✗	✗	✓

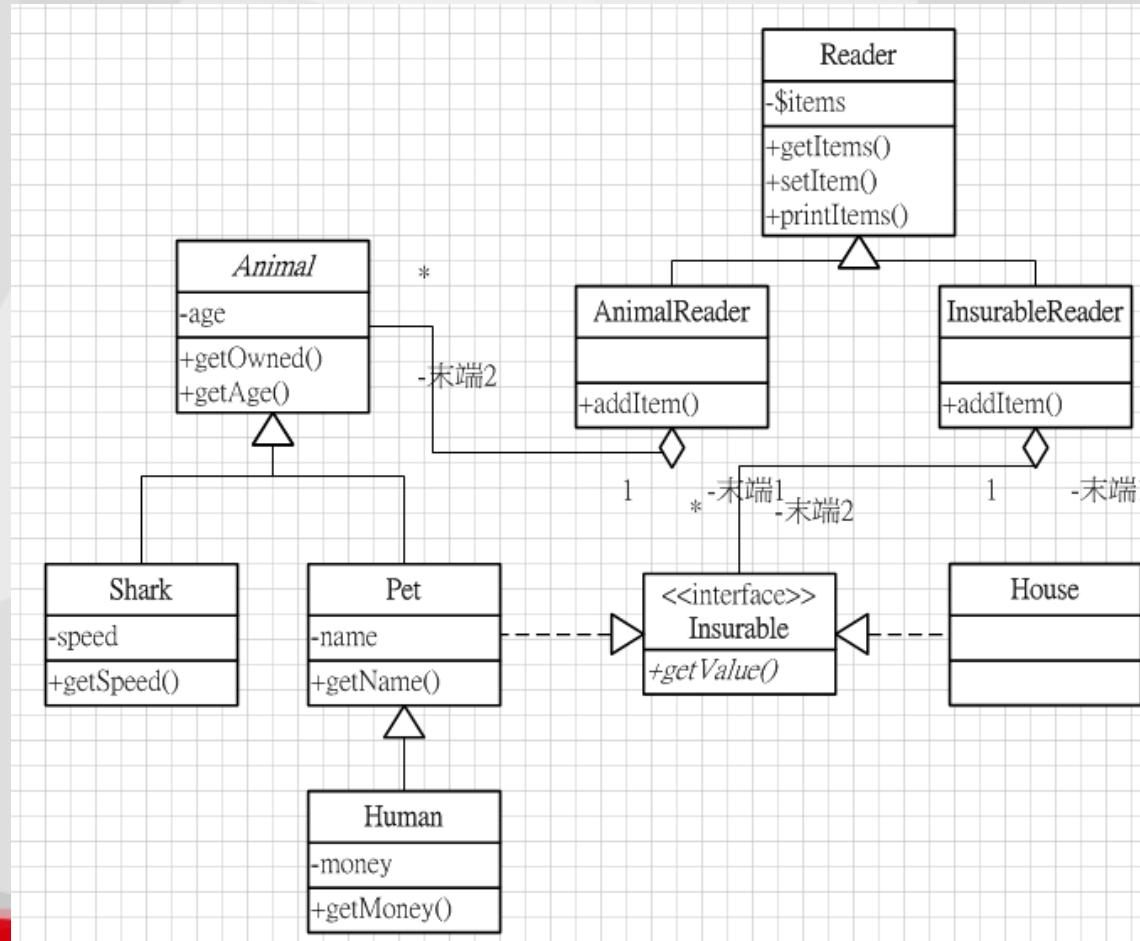
(補充)PHP類別、虛擬類別、介面差異(5/9)

Abstract Class與**Interface**的使用時機：

- 當類別有共同的行為或屬性時，可以考慮使用 Abstract Class
- 當類別有共同的操作介面，但是實作上卻有所差異時，可以考慮使用 Interface 。

(補充)PHP類別、虛擬類別、介面差異(6/9)

範例(UML):



(補充)PHP類別、虛擬類別、介面差異(7/9)

範例(實作1/3):

```
abstract class Animal
{
    private $age;

    public abstract function getOwned();

    protected function __construct($age) {
        $this->age = $age;
    }

    public function getAge(){
        return $this->age;
    }
}

interface Insurable
{
    public function getValue();
}
```

```
// 繼承至Animal
class Shark extends Animal
{
    private $speed; // 游泳速度

    public function __construct($speed,$age) {
        parent::__construct($age);
        $this->speed = $speed;
    }

    public function getSpeed() {
        return $this->speed;
    }

    // 實作自abstract class繼承來的method
    public function getOwned() {
        return ("getOwned Shark String");
    }
}

// 繼承至Animal、定義了Insurable介面
class Pet extends Animal implements Insurable
{
    private $name;

    public function __construct($name,$age) {
        parent::__construct($age);
        $this->name = $name;
    }

    public function getName() {
        return $this->name;
    }

    // 實作自abstract class繼承來的method
    public function getOwned() {
        return ("getOwned Pet String");
    }

    // 實作自Interface定義的method
    public function getValue() {
        return ("getValue Pet String");
    }
}
```

(補充)PHP類別、虛擬類別、介面差異(8/9)

範例(實作2/3):

```
// 繼承至Pet，同時也繼承了Pet實作getValue()的部分，  
// 等於也繼承了Insurable Interface  
class Human extends Pet  
{  
    private $money;  
  
    public function __construct($money,$name,$age) {  
        parent::__construct($name, $age);  
        $this->money = $money;  
    }  
  
    public function getMoney() {  
        return $this->money;  
    }  
  
    // 實作自abstract class繼承來的method  
    public function getOwned() {  
        return ("getOwned Human String");  
    }  
  
    // 定義了Insurable介面  
class House implements Insurable  
{  
    // 實作自Interface定義的method  
    public function getValue() {  
        return ("getValue House String");  
    }  
}
```

```
abstract class Reader  
{  
    private $items = array();  
  
    public function getItems() {  
        return $this->items;  
    }  
  
    public function setItem($item) {  
        array_push($this->items, $item);  
    }  
  
    public abstract function printItems();  
}  
  
  
class InsurableReader extends Reader  
{  
    public function addItem(Insurable $item) {  
        parent::setItem($item);  
    }  
  
    public function printItems() {  
        foreach (parent::getItems() as $item) {  
            echo 'getValue():' . $item->getValue() . '<br/>';  
        }  
    }  
}  
  
class AnimalReader extends Reader  
{  
    public function addItem(Animal $item) {  
        parent::setItem($item);  
    }  
  
    public function printItems() {  
        foreach(parent::getItems() as $item) {  
            echo 'getOwned():' . $item->getOwned() . '<br/>';  
        }  
    }  
}
```

(補充)PHP類別、虛擬類別、介面差異(9/9)

範例(實作3/3):

```
include_once '../class/example/abstractClassAndInterface.php';

$pet = new Pet("pet",6);
$human = new Human(100, 'human', 27);
$shark = new Shark(200, 15);
$house = new House();

$insurableReader = new InsurableReader(); // Insurable讀取器

$insurableReader->addItem($pet);
$insurableReader->addItem($human);
// $insurableReader->addItem($shark);      // Shark沒有定義Insurable這個interface，故發生錯誤
$insurableReader->addItem($house);
$insurableReader->printItems();

echo '<br/>';

$animalReader = new AnimalReader(); // Animal讀起器

$animalReader->addItem($pet);
$animalReader->addItem($human);
$animalReader->addItem($shark);
// $animalReader->addItem($house); // House沒有繼承至Animal這個abstract class，故發生錯誤
$animalReader->printItems();
```

OOP設計原則(1/9)

- 低耦合(Low-Coupling)
- 高內聚力(High Cohesion)

OOP設計原則(2/9)

- 為介面而不是為實作而寫程式
(Program to an interface, not an implementation)
- 開-閉原則 (Open-Closed Principle, OCP)
軟體實體應當對擴充開放，對修改關閉。
(Software entities should be open for extension, but closed for modification)

也就是：軟體系統中包含的各種元件，例如模組（ Modules ）、類（ Classes ）以及功能（ Functions ）等等，應該在不修改現有代碼的基礎上，引入新功能。開閉原則中“開”，是指對於元件功能的擴展是開放的，是允許對其進行功能擴展。開閉原則中“閉”，是指對於原有代碼的修改是封閉的，即不應該修改原有的代碼。

OOP設計原則(3/9)：內聚

- **內聚力(Cohesion)**：模組內指令的相關性。
每個模組內描述的功能只有一個，單一輸入口及單一輸出口，資料定義只是為支援此一功能的設計。內聚力愈高愈佳。
 - 功能內聚力(Function Cohesion)：單獨處理一件工作
 - 順序內聚力(Sequential Cohesion)：模組順序執行，一個模組的輸出會成為下一組的輸入
 - 溝通內聚力(Communication Cohesion)：使用相同的資料
 - 暫時內聚力(Tempral Cohesion)：模組執行無順序關係但須在一定時間內完成一件工作
 - 程序內聚力(Procedural Cohesion)：按照順序執行而不共用資料
 - 邏輯內聚力(Logical Cohesion)：根據上層模組傳來的參數決定執行的功能
 - 偶發內聚力(Coincidental Cohesion)：模組可做好幾件不相干工作，各模組具有功能內聚力

OOP設計原則(4/9):耦合

- 耦合力(**Coupling**)：指模組間的相互依存性。
通常當依存性高時，會增加模組間維護及程式撰寫困難度
 - 資料耦合力(Data Coupling)：
模組間藉由資料傳遞參數
 - 資料結構耦合力(Stamp Coupling)：
模組間的溝通是用資料結構，但在目的模組中，並不一定會使用到所有資料欄位。
 - 控制耦合力(Control Coupling)：
一個模組的運作，是透過其他模組傳來控制訊號。
 - 共同耦合力(Common Coupling)：
模組間使用共同的資料變數(想成 **global** 變數)
 - 內容耦合力(Content Coupling)：
A模組可使用**B**模組的程式碼或改變其變數

OOP設計原則(5/9): 範例(1/5)

假設我們有一個表格，表格內容有兩個類別：
Circle跟Square，Circle的參數類型為半徑，
而Square的參數類型為邊長，如下表：

Type	parameter	value
Circle	radius	5.00
Square	side	2.00
Square	side	8.00

OOP設計原則(6/9): 範例(2/5)

```
class Lister
{
    public $items = array();

    public function addItem($item) {
        array_push($this->items, $item);
    }

    public function out() {
        echo "+-----+<br/>
          | Type   | parameter | value  |<br/>
          +-----+<br/>";

        for ($i = 0, $s = sizeof($this->items); $i < $s; $i++) {

            // 判斷是Circle還是Square
            if ($this->items[$i] instanceof Circle) {
                echo "| Circle | radius   | " . sprintf('%5.2f', $this->items[$i]->radius) . " |<br/>";
            } else {
                echo "| Square | side     | " . sprintf('%5.2f', $this->items[$i]->side) . " |<br/>";
            }
            echo "+-----+<br/>";
        }
    }
}
```

```
class Circle
{
    public $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }
}

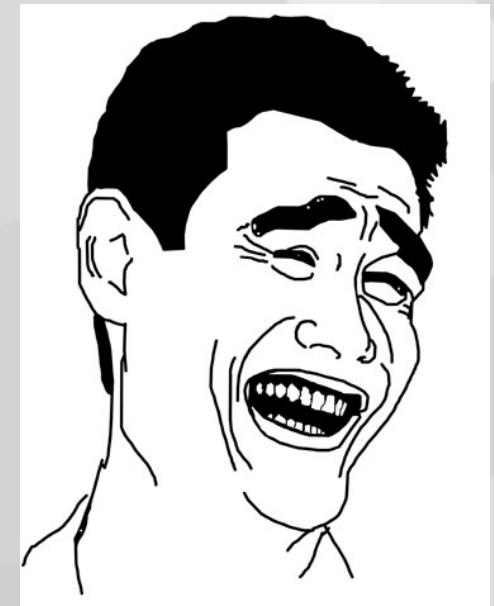
class Square
{
    public $side;

    public function __construct($side) {
        $this->side = $side;
    }
}
```

我們很直覺地寫出這些程式碼，收工！

如果這時候我們要加上星型、菱形.....

好吧，那就硬幹囉~~~~~



OOP設計原則(7/9): 範例(3/5)

```
class Lister
{
    public $items = array();

    public function addItem($item) {
        array_push($this->items, $item);
    }

    public function out() {
        echo "+-----+<br/>
        | Type   | parameter | value  |<br/>
        +-----+<br/>";

        for ($i = 0, $s = sizeof($this->items); $i < $s; $i++) {

            //判斷是Circle還是Square
            if ($this->items[$i] instanceof Circle){
                echo "| Circle | radius   | " . sprintf('%5.2f', $this->items[$i]->radius) . " |<br/>";
            } else if ($this->items[$i] instanceof Square){
                echo "| Square | side     | " . sprintf('%5.2f', $this->items[$i]->side) . " |<br/>";
            } else if ($this->items[$i] instanceof Star){
                echo "| Star   | angle    | " . sprintf('%5.2f', $this->items[$i]->side) . " |<br/>";
            } else {
                echo "| Diamond | diam    | " . sprintf('%5.2f', $this->items[$i]->side) . " |<br/>";
            }
        }
        echo "+-----+<br/>";
    }
}
```



如果這時候我們要再加上其他形狀，或是Lister有其他print格式時.....



Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

OOP設計原則(8/9): 範例(4/5)

問題出在哪？

1. Lister與Circle..等緊綁在一起。



高耦合

2. Lister沒有Circle..等就不能運作。



低聚合

3. Circle..等不能獨立運作。



低聚合

OOP設計原則(9/9): 範例(5/5)

程式碼重構：

```
abstract class Shape
{
    private $value;

    public function __construct($value) {
        $this->value = $value;
    }

    public function getValue() {
        return $this->value;
    }

    public abstract function out();
}

class Circle extends Shape
{
    public function out(){
        echo "| Circle | radius | " . sprintf('%5.2f', $this->getValue()) . " |<br/>";
    }
}

class Square extends Shape
{
    public function out() {
        echo "| Square | side | " . sprintf('%5.2f', $this->getValue()) . " |<br/>";
    }
}
```

```
class Lister
{
    public $items = array();

    public function addItem(Shape $item) {
        array_push($this->items, $item);
    }

    public function out() {
        echo "+---+---+<br/>
              | Type | parameter | value |<br/>
              +---+---+<br/>";

        for ($i = 0, $s = sizeof($this->items); $i < $s; $i++) {
            $this->items[$i]->out();
            echo "+---+---+<br/>";
        }
    }
}
```

Shape列印功能實作部分從
Lister中分離出來了!!
變得更容易維護、修改與擴充

Hiiir

程式碼需要重構的四個征兆(1/5)

1. 程式碼有重複
2. 類別知道的太多
3. 萬能類別
4. 類別中頻繁出現特定條件的判斷(**if, switch**)

程式碼需要重構的四個征兆(2/5)

1. 程式碼有重複

程式碼重複的地方通常意味著緊密耦合。

如果發生改變一個地方的程式碼，同時也要修改另一個地方的程式碼的狀況，那這兩個程式碼可能本 應該放在同一個地方。

程式碼需要重構的四個征兆(3/5)

2. 類別知道的太多

如果一個類別使用了它區域(scope)以外的內容，該類別就會綁定到外部環境中，使他很難保持獨立，導致發生高耦合。例如使用全域(global)變數，將容易破壞類別的獨立性，使用時因慎重考慮。

程式碼需要重構的四個征兆(4/5)

3. 萬能類別

類別是否嘗試一次完成很多工作？如果是的話，就要檢查類別的職責。會發現其中一些功能可以獨立出來，成為另一個類別。

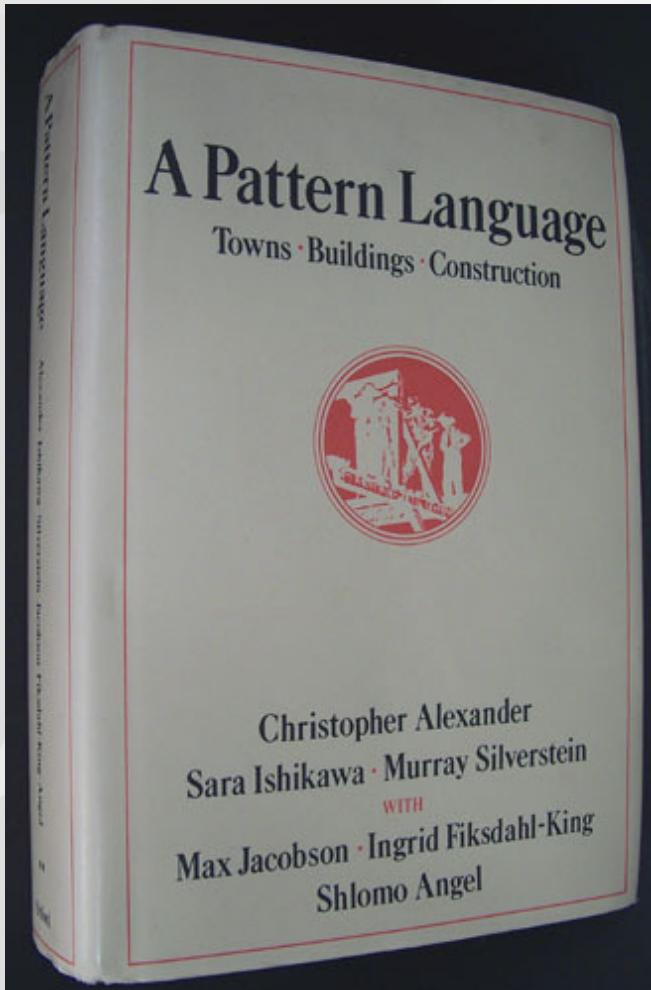
如果類別的責任過多，那在創建子類別的時候可能會產生問題。容易導致產生過多的子類別或是過度依賴條件判斷敘述。

程式碼需要重構的四個征兆(5/5)

4. 類別中頻繁出現特定條件的判斷(**if, switch**)

如果發現一個類別中頻繁的使用特定條件的判斷，特別是當你發現這些條件判斷在多個方法中出現時，就說明這個類別需要拆分成兩個或者更多。將某些功能獨立出來放在獨立類別中，拆分出來的類別應該有共同的基礎抽象類別。

什麼是設計模式?(1/3)



奧地利 建築師 Christopher Alexander

樣式語言 (Pattern Language) 一詞，起源於 Christopher Alexander 在 1970 年代前後對於建築領域所做的理論研究與實作。Alexander 認為模式 (Pattern) 可有效解決在特定建築領域中，一再重複出現的問題。

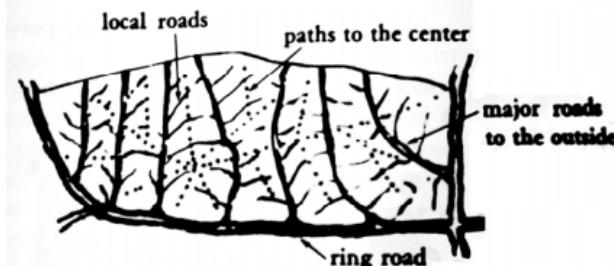
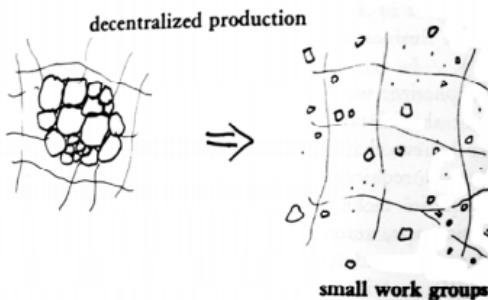
什麼是設計模式?(2/3)

- Preserving the agricultural valleys for farmlands and protect them from and further development of any building.

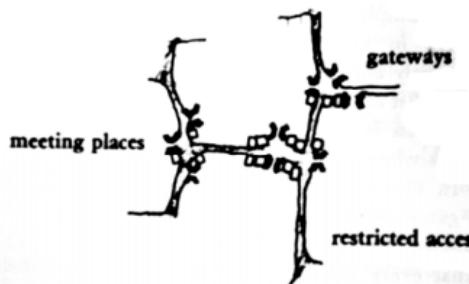
hills for building



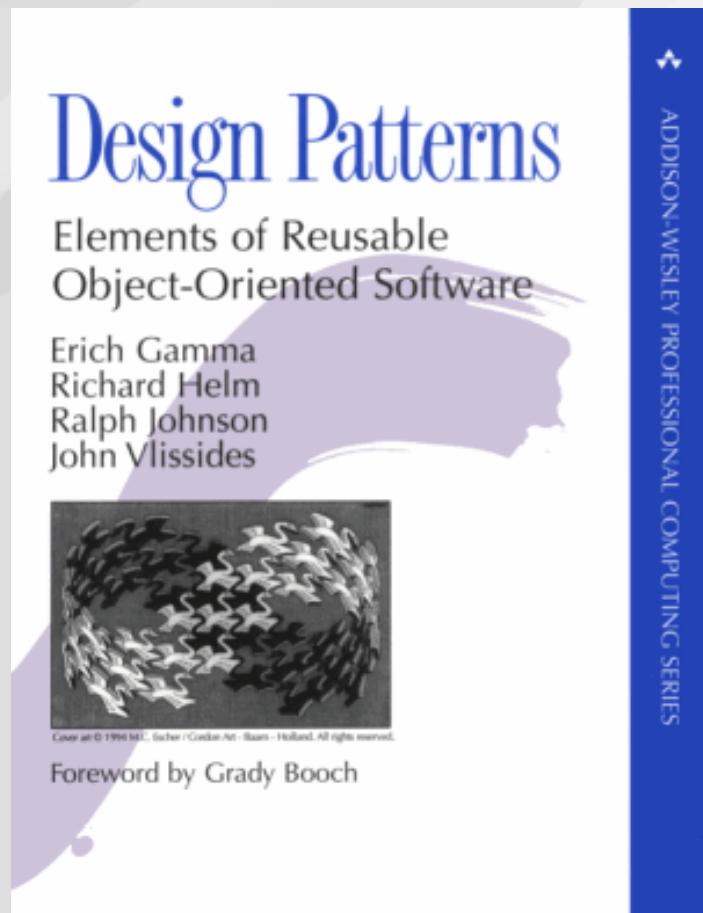
- Distribution of work space and the residential areas and the hierarchy of the roads for local transport.



- Defining a neighborhood and the amount of people in it and formation of a boundary around it .



什麼是設計模式?(3/3)



Design Patterns – Elements of Reusable Object-Oriented Software

Hiiir

Erich Gamma
Ralph Johnson
John Vlissides
Richard Helm



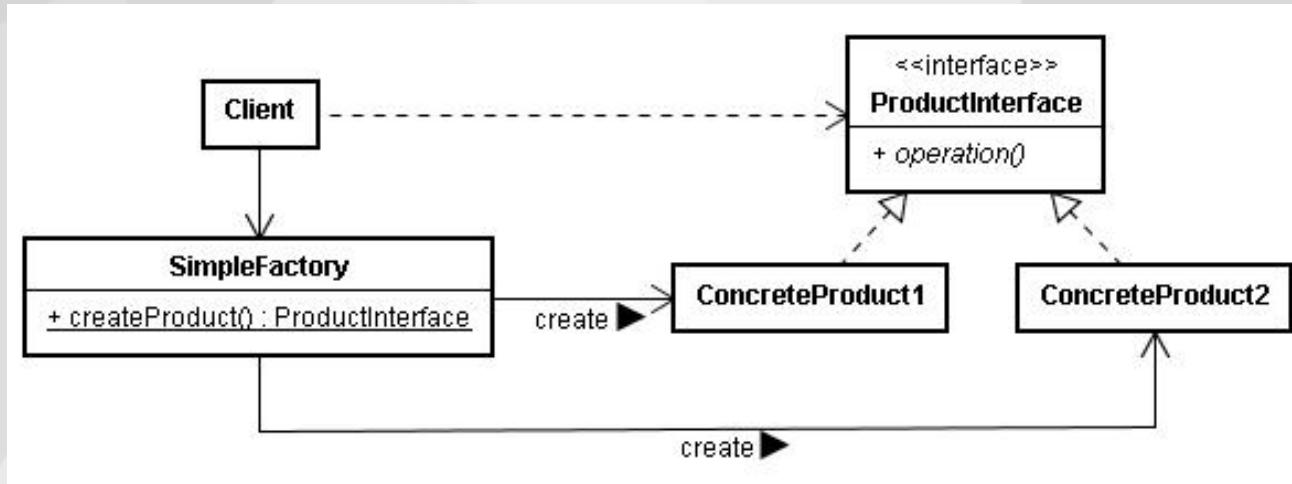
Gang of Four(GoF)

Simple Factory(簡單工廠模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

Simple Factory(1/5):UML



Simple Factory(2/5): 模式分析

優點：

- 分工明確，各司其職。
- 用戶端不再創建物件，而是把創建物件的職責交給了具體的工廠去創建。
- 用戶端只負責調用。

缺點：

- 工廠的靜態方法無法被繼承。
- 代碼維護不易（物件要是很多的話，工廠是一個很龐大的類別）。
- 這種模式對OCP原則(Open-Close Principle)支援的不夠，如果有新的產品加入到系統中就要修改工廠類。

使用時機：

- 用戶端需要創建物件，物件有同樣製作流程，但實作內容不同，並且數量不多的情況下，可以考慮使用簡單工廠模式。

Simple Factory(3/5):範例(1/3)

現在有一個基地需要訓練兩種單位：工人與士兵，一般老百姓要成為工人或士兵，都需要經過三個步驟，分別為使用資源、訓練跟產出，這時候我們要怎麼規劃呢？



Simple Factory(4/5): 範例(2/3)

```
// 抽象單位  
abstract class Unit {  
    public abstract function getMaterial(); // 取得材料  
    public abstract function train(); // 訓練  
    public abstract function create(); // 產生  
  
// 工人  
class Worker extends Unit {  
    public function getMaterial() {  
        echo "使用了50單位的水晶<br/>";  
    }  
  
    public function train() {  
        echo "訓練時間10秒<br/>";  
    }  
  
    public function create() {  
        echo "I am a Worker, I am ready to work!<br/><br/>";  
    }  
}
```

Product

ConcreteProduct

```
// 士兵  
class Solider extends Unit {  
    public function getMaterial() {  
        echo "使用了50單位的水晶、10單位的瓦斯<br/>";  
    }  
  
    public function train() {  
        echo "訓練時間20秒<br/>";  
    }  
  
    public function create() {  
        echo "I am a Solider, Waiting for your order!<br/><br/>";  
    }  
}
```

ConcreteProduct

Simple Factory(5/5): 範例(3/3)

```
// 簡單建築物工廠
class SimpleBuildFactory {
    public static function createUnit($type) {
        switch($type) {
            case "solider":
                return new Solider();
                break;

            case "worker":
                return new Worker();
                break;
        }
    }
}

// 生產中心
class CreateBuildCenter
```

SimpleFactory

Client

```
include_once '../class/pattern/simpleFactory.php';

$building = new CreateBuildCenter(); // 建立生產中心

$solider = $building->outputUnit('solider'); // 建立士兵
$worker = $building->outputUnit('worker'); // 建立工兵
```

使用了50單位的水晶、10單位的瓦斯
訓練時間20秒
I am a Solider, Waiting for your order!

使用了50單位的水晶
訓練時間10秒
I am a Worker, I am ready to work!

Factory Method(工廠方法模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

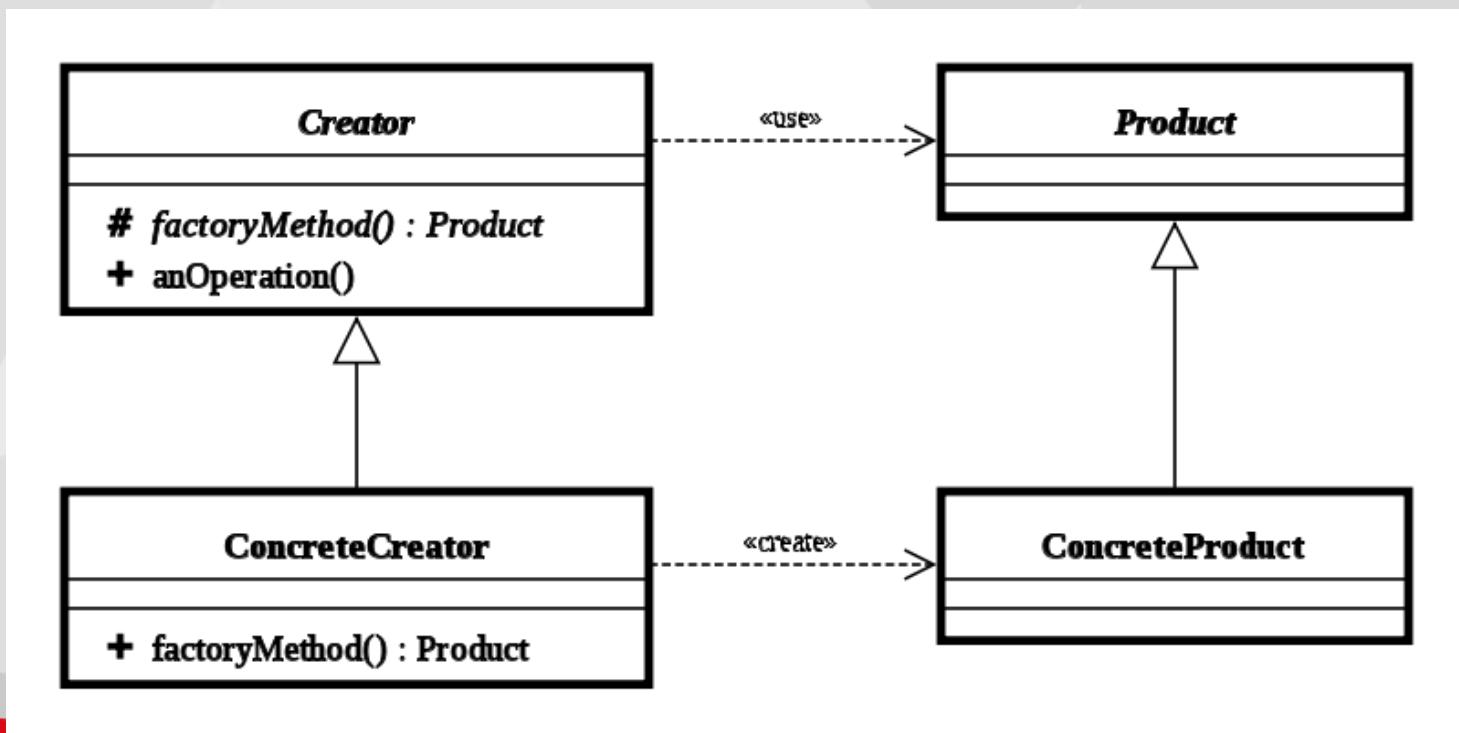
Factory Method(1/6):UML

Product：定義 Factory Method 所造物件的介面。

ConcreteProduct：具體實作出 **Product** 介面。

Creator：宣告 Factory Method，它會傳回 **Product** 型別之物件。

ConcreteCreator：覆寫 Factory Method 以傳回**ConcreteProduct** 的物件個體。



Factory Method(2/6): 模式分析(1/2)

優點：

- 有了 Factory Method 設計模式，就不必再將與應用場合高度相依的類別寫死在主程式裡。主程式只須面對 Product 介面，因此可和任何未知的ConcreteProduct 類別合作無間。
- 不論工廠內部的管理方法、製造流程如何地更換變動，都不會影響到系統外部使用者的操作行為，完善地封裝隱藏物件生成過程的實做細節。

缺點：

- 在添加新產品時，需要編寫新的具體產品類，而且還要提供與之對應的具體工廠類，系統中類別的個數將成對增加，在一定程度上增加了系統的複雜度，有更多的類別需要編譯和運行，會給系統帶來一些額外的開銷。
- 由於考慮到系統的可擴展性，需要引入抽象層，在用戶端代碼中均使用抽象層進行定義，增加了系統的抽象性和理解難度，且在實現時可能需要用到DOM、反射等技術，增加了系統的實現難度。

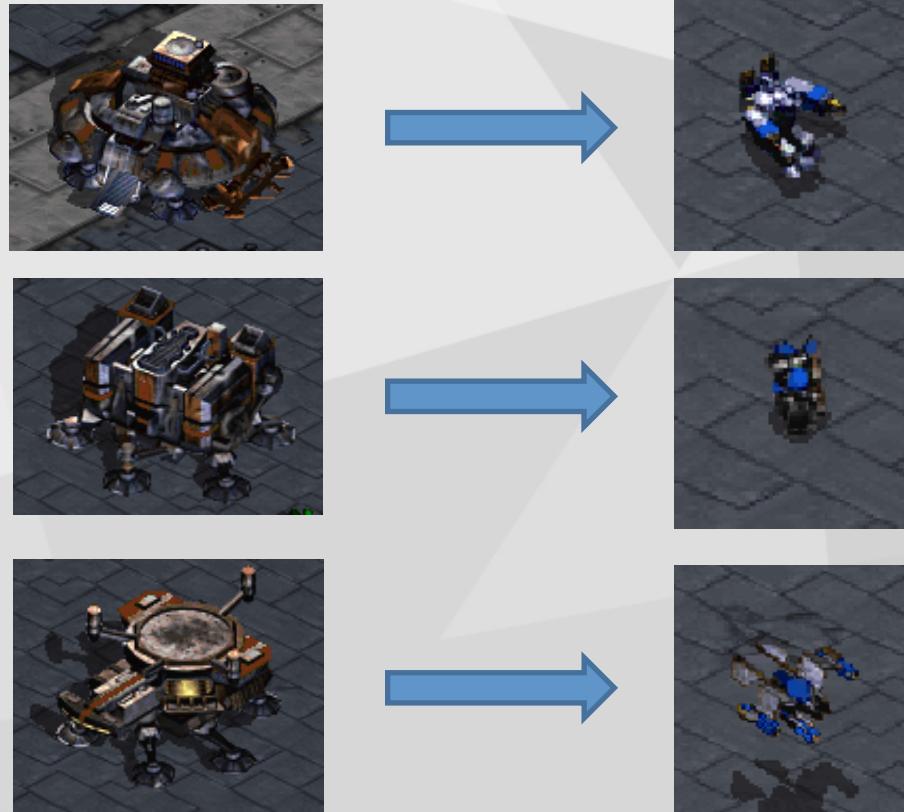
Factory Method(3/6): 模式分析(2/2)

使用時機：

- 當類別無法明指欲生成的物件類別時。
- 當類別希望讓子類別去指定欲生成的物件類型時。
- 當類別將權力下放給一個或多個輔助用途的子類別，你又希望將「交付給哪些子類別」的知識集中在一處時。

Factory Method(4/6): 範例(1/3)

現在除了要生產工人與士兵，還要生產戰鬥機，但是戰鬥機還需要多一個工序：建造，這時候我們要怎麼規劃呢？



Factory Method(5/6): 範例(2/3)

```
// 抽象的建築物工廠
abstract class BuildFactory
{
    abstract function outputUnit();
}

// 指揮中心
class CommandCenter extends BuildFactory
{
    public function outputUnit() {
        return new Worker();
    }
}

// 軍營
class Barrack extends BuildFactory
{
    public function outputUnit() {
        return new Solider();
    }
}

// 空軍基地
class Airport extends BuildFactory
{
    public function outputUnit() {
        $aircraft = new Aircraft();
        $aircraft->build();
        return $aircraft;
    }
}
```

Creator

ConcreteCreator

ConcreteCreator

ConcreteCreator

Product

```
// 抽象單位
abstract class Unit
{
    public abstract function playSlogan(); // 播放單位口號
}

// 工人
class Worker extends Unit
{
    public function playSlogan() {
        echo "SUV準備好了，長官你想蓋啥建築呢?<br/><br/>";
    }
}

// 士兵
class Solider extends Unit
{
    public function playSlogan() {
        echo "快給我戰鬥嘛吧!!<br/><br/>";
    }
}

// 飛機
class Aircraft extends Unit
{
    public function playSlogan() {
        echo "我已經準備好起飛出擊了，長官!<br/><br/>";
    }

    public function build() {
        echo "組裝飛機中...<br/>";
    }
}
```

ConcreteProduct

ConcreteProduct

ConcreteProduct

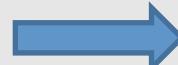
Factory Method(6/6): 範例(3/3)

```
include_once '../class/pattern/factory.php';

$barrack = new Barrack();           // 建立軍營
$solider = $barrack->outputUnit(); // 產生單位
$solider->playSlogan();

$commandCenter = new CommandCenter(); // 建立指揮中心
$worker = $commandCenter->outputUnit(); // 產生單位
$worker->playSlogan();

$airport = new Airport();           // 建立機場
$aircraft = $airport->outputUnit(); // 產生單位
$aircraft->playSlogan();
```



快給我戰鬥藥吧!!

SUV準備好了，長官你想蓋啥建築呢？

組裝飛機中...

我已經準備好起飛出擊了，長官！

Abstract Factory(抽象工廠模式)

Abstract Factory(1/6):UML

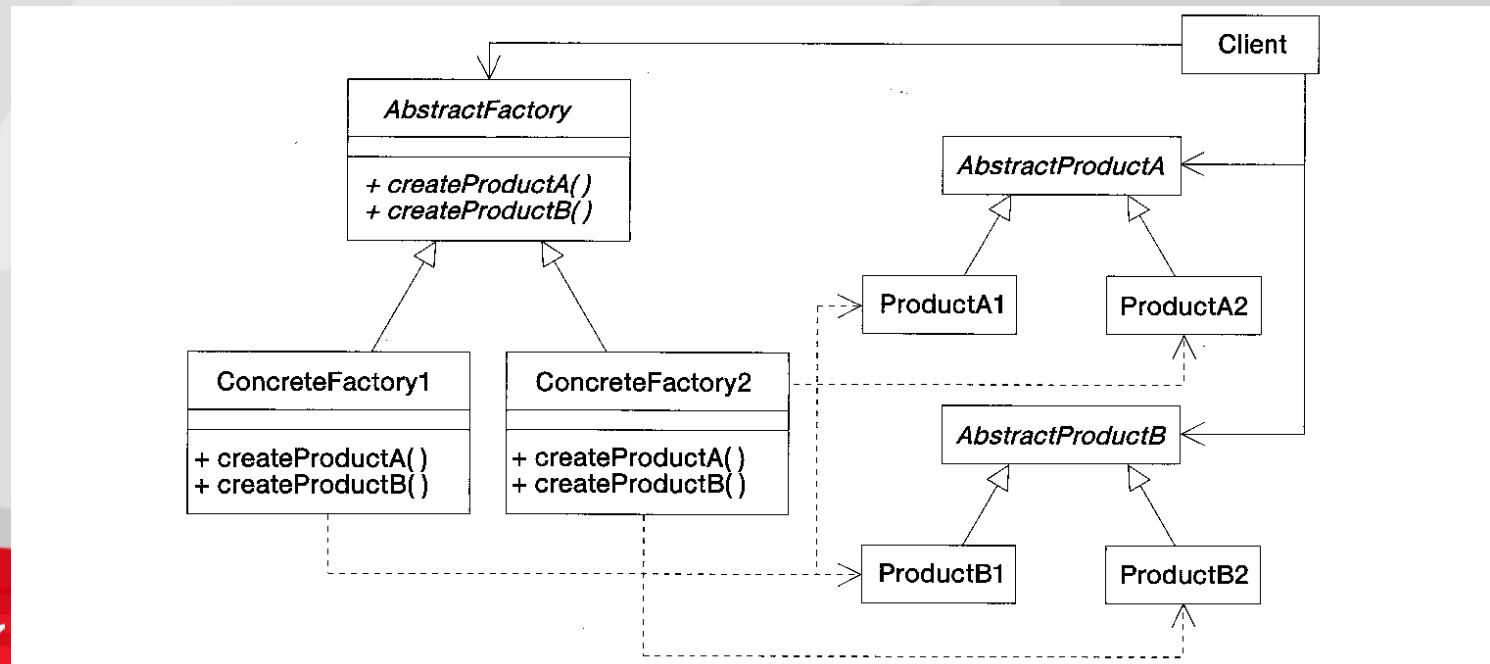
AbstractFactory：此介面宣告出可生成各抽象成品物件的操作。

ConcreteFactory：具體實作出可建構具象成品物件的操作。

AbstractProduct：宣告某成品物件類型之介面。

ConcreteProduct：是 ConcreteFactory 所建構的成品物件；也是 AbstractFactory 介面的具象實作。

Client：只觸及 AbstractFactory 和 AbstractProduct 兩抽象類別所訂之介面。



Abstract Factory(2/6): 模式分析

優點：

- 擴充了工廠方法模式，支援多個產品族的創建
- 做到了“開-閉”原則

缺點：

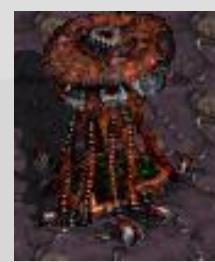
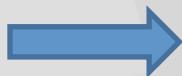
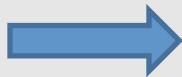
- 暴露具體工廠
- 系統架構更加複雜

使用時機：

- 有創建一批有相同介面物件的需求
- 不想暴露太多類的細節給使用者，或者隱藏物件的創建工作
- 產品**包含多於一個的產品族**，用工廠方法模式無法解決。其中同一個產品族的產品是在一起使用的
- 系統提供一個產品類的庫，用戶端可以不依賴於實現，做到動態切換產品。

Abstract Factory(3/6):範例(1/4)

現在除了有人類單位，現在要多一個蟲族生產線，這時候我們要怎麼規劃呢？



Abstract Factory(4/6): 範例(2/4)

Abstract ProductA

```
// 抽象人類單位
abstract class TerranUnit
{
    public abstract function playSlogan(); // 播放單位口號
}

// 工人
class TerranWorker extends TerranUnit
{
    public function playSlogan() {
        echo "SUV準備好了，長官你想蓋啥建築呢?<br/><br/>";
    }
}

// 士兵
class TerranSolider extends TerranUnit
{
    public function playSlogan() {
        echo "快給我戰鬥藥吧!!<br/><br/>";
    }
}

// 飛機
class TerranAircraft extends TerranUnit
{
    public function playSlogan() {
        echo "我已經準備好起飛出擊了，長官!<br/><br/>";
    }
}
```

ProductA1

ProductA2

ProductA3

Abstract ProductB

```
// 抽象蟲族單位
abstract class ZergUnit
{
    public abstract function shout(); // 嘶吼
}

// 工人
class ZergWorker extends ZergUnit
{
    public function shout() {
        echo "(黏液黏液)，請問主宰您想長出什麼建築物?<br/><br/>";
    }
}

// 士兵
class ZergSolider extends ZergUnit
{
    public function shout() {
        echo "給我更多的人類!!!<br/><br/>";
    }
}

// 飛蟲
class ZergAircraft extends ZergUnit
{
    public function shout() {
        echo "嘎~~~~嘎~~~~嘎~~~~<br/><br/>";
    }
}
```

ProductB1

ProductB2

ProductB3

Abstract Factory(5/6): 範例(3/4)

Abstract Factory

```
// 抽象的建築物工廠
abstract class BuildFactory
{
    public abstract function outputTerranUnit(); // 產出人類單位
    public abstract function outputZergUnit(); // 產生異形單位
}
```

ConcreteFactory

```
// 指揮中心
class CommandCenter extends BuildFactory
{
    public function outputTerranUnit() {
        return new TerranWorker();
    }

    public function outputZergUnit() {
        return new ZergWorker();
    }
}

// 軍營
class Barrack extends BuildFactory
{
    public function outputTerranUnit() {
        return new TerranSolider();
    }

    public function outputZergUnit() {
        return new ZergSolider();
    }
}

// 空軍基地
class Airport extends BuildFactory
{
    public function outputTerranUnit() {
        return new TerranAircraft();
    }

    public function outputZergUnit() {
        return new ZergAircraft();
    }
}
```

Abstract Factory(6/6): 範例(4/4)

```
include_once '.../../class/pattern/abstractFactory.php';

$barrack = new Barrack(); //建立軍營
$terranUnit = $barrack->outputTerranUnit(); //產生人類士兵
$zergUnit = $barrack->outputZergUnit(); //產生異形士兵

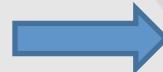
$terranUnit->playSlogan();
$zergUnit->shout();

$commandCenter = new CommandCenter(); //建立指揮中心
$terranUnit = $commandCenter->outputTerranUnit(); //產生人類工兵
$zergUnit = $commandCenter->outputZergUnit(); //產生異形工兵

$terranUnit->playSlogan();
$zergUnit->shout();

$airport = new Airport(); //建立機場
$terranUnit = $airport->outputTerranUnit(); //產生人類飛機
$zergUnit = $airport->outputZergUnit(); //產生異形飛機

$terranUnit->playSlogan();
$zergUnit->shout();
```



快給我戰鬥藥吧!!

給我更多的人類!!!

SUV準備好了，長官你想蓋啥建築呢?

(黏液黏液)，請問主宰您想長出什麼建築物?

我已經準備好起飛出擊了，長官!

嘎~~~~~嘎~~~~~嘎~~~~~

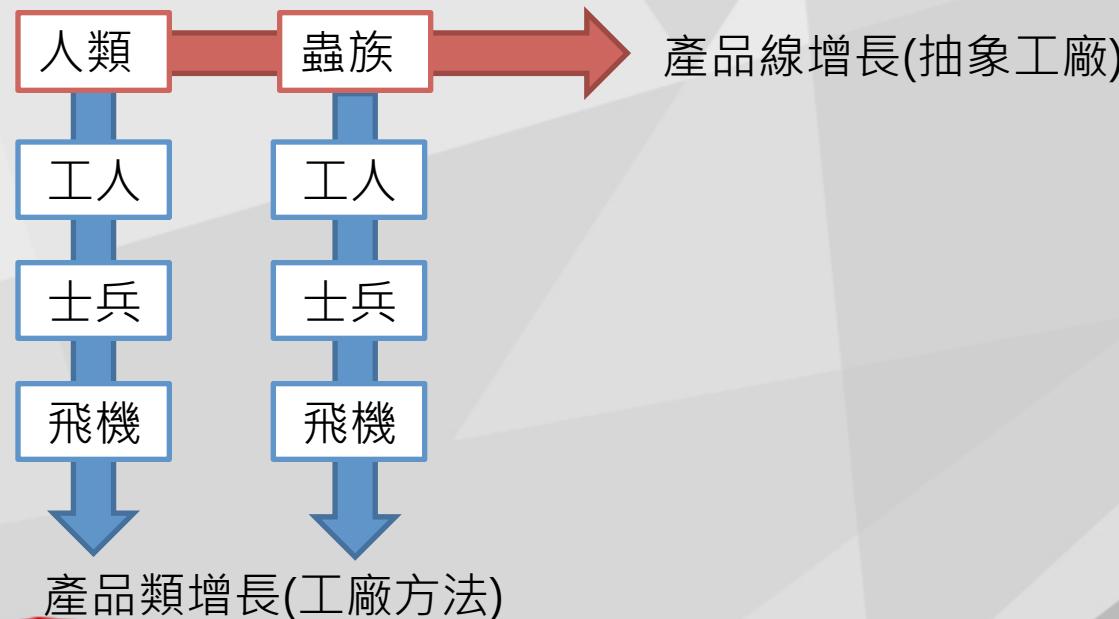
工廠系列模式差異比較

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

工廠系列模式差異比較(1/1)

- 簡單工廠：只有一個具體工廠類來創建一種父類別的多個不同子類別
- 工廠方法：多個派生於一個父類別的具體工廠類，每個具體工廠只生產一種父類別的一個子類別(產品類的增長，垂直增長)
- 抽象工廠：也是只有一個工廠基類，但是每個具體工廠生產多個相關父類別的一個子類別(產品線的增長，水平增長)



工廠系列模式好處到底在哪裡？

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

工廠系列模式好處到底在哪裡(1/1)

- 如果有個class叫Person, 他的construct被修改了，例如：
\$person = new Person(); 改為 \$person = new Person('Hank')，如果在很多地方都有person物件，就要一個個去修改。如果用工廠模式，產生物件就是：

```
$factory = new PersonFactory();
$person = $factory->createObj();
```

這樣只要去改PersonFactory的createObj()就好，不會動到client端。
- 如果今天所有Person物件都要改成Plant物件，我們只要改變一開始工廠的物件，例如：\$factory = new PersonFactory() 改為 \$factory = new PlantFactory()，那factory->createObj()就會生產植物物件出來，最小化的減少client端的變動。

Builder(生成器模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

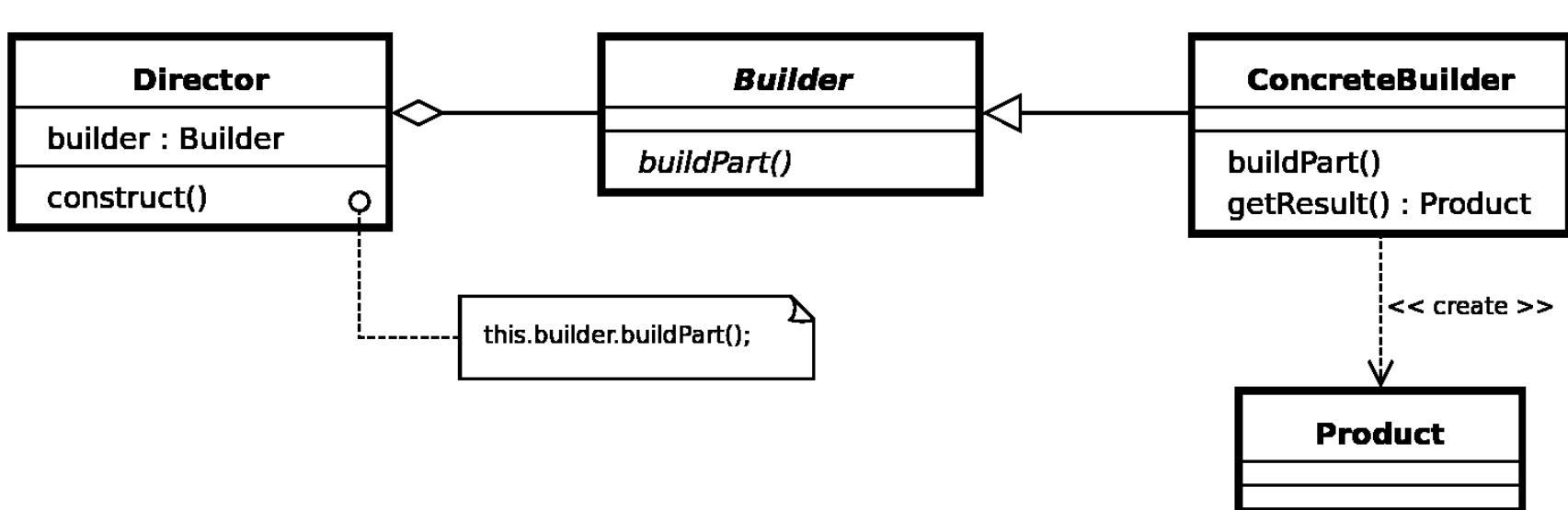
Builder(1/14):UML

Builder：給出一個抽象介面，以規範產品物件的各個組成成分的建造。一般而言，此介面獨立于應用程式的商業邏輯。模式中直接創建產品物件的是ConcreteBuilder。具體建造者類必須實現這個介面所要求的方法：一個是建造方法，另一個是結果返還方法。

Concrete Builder：實作Builder的接口，建造Product物件並回傳。

Director : 使用Builder以創建Product物件。

Product：預計被製作的產品



Builder(2/14): 模式分析(1/2)

優點：

- 客戶端不必知道Product內部組成的細節，將Product本身與Product的創建過程解耦，使得相同的創建過程可以創建不同的Product物件。
- 每一個Concrete Builder都相對獨立，而與其他的Concrete Builder無關，因此可以很方便地替換Concrete Builder或增加新的Concrete Builder，用戶使用不同的Concrete Builder即可得到不同的Product物件。
- 可以更加精細地控制Product的創建過程。將複雜Product的創建步驟分解在不同的方法中，使得創建過程更加清晰，也更方便使用程式來控制創建過程。
- 增加新的Concrete Builder無須修改原有類別的代碼，Director針對Builder程式設計，系統擴展方便，符合“開閉原則”。

Builder(3/14): 模式分析(2/2)

缺點：

- Builder模式所創建的Product一般具有較多的共同點，其組成部分相似，**如果Product之間的差異性很大，則不適合使用Builder模式**，因此其使用範圍受到一定的限制。
- 如果Product的內部變化複雜，可能會導致需要定義很多Concrete Builder類別來實現這種變化，導致系統變得很龐大。

使用時機：

- 需要生成的產品物件有複雜的內部結構，這些產品物件通常**包含多個成員屬性**。
- 需要生成的產品物件的屬性相互依賴，需要指定其生成順序。
- 物件的創建過程獨立於創建該物件的類別。在Builder模式中引入了Director類別，將創建過程封裝在Director類別中，而不在Builder類別中。
- 隔離複雜物件的創建和使用，並使得**相同的創建過程可以創建不同的產品**。
- 經常被用來建立合成結構

Builder_(4/14): 範例1-完整Builder_(1/4)

現在我們要能產生超值套餐，每份套餐裡面都有飲料、點心與主食，要怎麼做呢？

The image shows a promotional menu from McDonald's. On the left, there is a grid of meal deals with their names, prices, and small images. On the right, there is a large circular badge with the McDonald's logo and text.

套餐名稱	價格	原價
雙層牛肉吉事堡餐	\$79	\$109
麥香雞餐	\$79	\$99
麥香魚餐	\$79	\$105
大麥克餐	\$99	\$119
勁辣雞腿堡餐	\$99	\$125
麥克雞塊餐六塊	\$99	\$109
四盎司牛肉堡餐	\$109	\$125
麥脆雞餐二塊	\$119	\$135
板烤雞腿堡餐	\$119	\$129
雙層四盎司牛肉堡餐	\$129	\$145

超值午/晚餐 \$79 起

麥當勞陪你抗漲
超值晚餐 \$79 起
晚餐同享超值午餐優惠價
11:00~14:00 超值午餐長期供應 17:00~20:00 超值晚餐限期優惠

• 超值晚餐餐組

超值晚餐優惠至2012年7月31日
超值午餐、晚餐不適用於歡樂送

Hiiir

Builder(5/14):範例1-完整Builder(2/4)

```
// Product
class Meal
{
    private $food; // 主食
    private $drink; // 飲料
    private $dessert; // 點心

    public function setFood($food) {
        $this->food = $food;
    }

    public function setDrink($drink) {
        $this->drink = $drink;
    }

    public function setDessert($dessert) {
        $this->dessert = $dessert;
    }

    public function showMeal() {
        echo $this->food . ", " . $this->drink . ", " . $this->dessert . "<br>";
    }
}
```

Product

```
// Builder
abstract class MealBuilder
{
    protected $meal = null;

    public function __construct() {
        $this->meal = new Meal();
    }

    public abstract function buildFood();
    public abstract function buildDrink();
    public abstract function buildDessert();
    public function getMeal() {
        return $this->meal;
    }
}
```

Builder

Builder(6/14):範例1-完整Builder(3/4)

```
// ConcreteBuilder
class ChickenKitMealBuilder extends MealBuilder
{
    public function buildFood() {
        $this->meal->setFood("一個雞腿堡");
    }

    public function buildDrink() {
        $this->meal->setDrink("一杯可樂");
    }

    public function buildDessert() {
        $this->meal->setDessert("一包薯條");
    }
}

// ConcreteBuilder
class BeefKitMealBuilder extends MealBuilder
{
    public function buildFood() {
        $this->meal->setFood("一個牛肉堡");
    }

    public function buildDrink() {
        $this->meal->setDrink("一杯紅茶");
    }

    public function buildDessert() {
        $this->meal->setDessert("一個蘋果派");
    }
}
```

ConcreteBuilder

```
// Director
class MealDirector
{
    private $mealBuilder;

    public function setMealBuilder(MealBuilder $mealBuilder) {
        $this->mealBuilder = $mealBuilder;
    }

    public function buildMeal() {
        $this->mealBuilder->buildFood();
        $this->mealBuilder->buildDrink();
        $this->mealBuilder->buildDessert();

        return $this->mealBuilder->getMeal();
    }
}
```

MealDirector

Hiiir

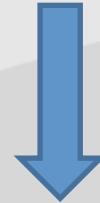
Builder(7/14):範例1-完整Builder(4/4)

```
include_once '....../class/pattern/builder.php';

$chickenKitMealBuilder = new ChickenKitMealBuilder();
$beefKitMealBuilder = new BeefKitMealBuilder();
$mealDirector = new MealDirector();

$mealDirector->setMealBuilder($chickenKitMealBuilder);
$chickenKitMeal = $mealDirector->buildMeal();
$chickenKitMeal->showMeal();

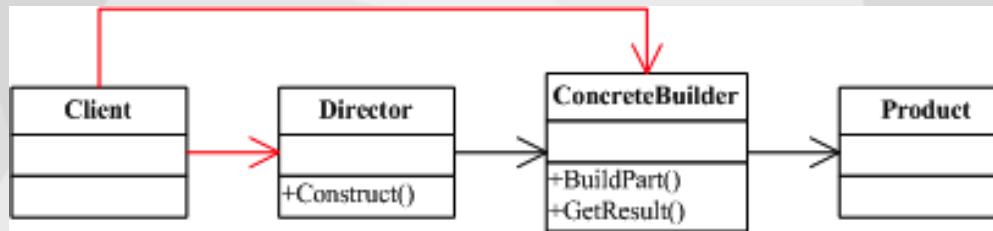
$mealDirector->setMealBuilder($beefKitMealBuilder);
$beefKitMeal = $mealDirector->buildMeal();
$beefKitMeal->showMeal();
```



一個雞腿堡，一杯可樂，一包薯條
一個牛肉堡，一杯紅茶，一個蘋果派

Builder(8/14):範例1-省略Builder(抽象建造者)

如果只有一個ConcreteBuilder的話，那可以省略Builder(抽象建造者)



```
// ConcreteBuilder
class ChickenKitMealBuilder
{
    protected $meal = null;

    public function __construct() {
        $this->meal = new Meal();
    }

    public function buildFood() {
        $this->meal->setFood("一個雞腿堡");
    }

    public function buildDrink() {
        $this->meal->setDrink("一杯可樂");
    }

    public function buildDessert() {
        $this->meal->setDessert("一包薯條");
    }

    public function getMeal() {
        return $this->meal;
    }
}
```

```
// Director
class MealDirector
{
    private $mealBuilder;

    public function setMealBuilder($mealBuilder) {
        $this->mealBuilder = $mealBuilder;
    }

    public function buildMeal() {
        $this->mealBuilder->buildFood();
        $this->mealBuilder->buildDrink();
        $this->mealBuilder->buildDessert();

        return $this->mealBuilder->getMeal();
    }
}
```

Builder(9/14): 範例1-省略Director

如果只有一個具體建造者，除了可以省略Builder，還可以把Director也省略，讓ConcreteBuilder也承擔Director的角色

```
// ConcreteBuilder
class BeefKitMealBuilder
{
    protected $meal = null;

    public function __construct() {
        $this->meal = new Meal();
    }

    public function buildFood() {
        $this->meal->setFood("一個牛肉堡");
    }

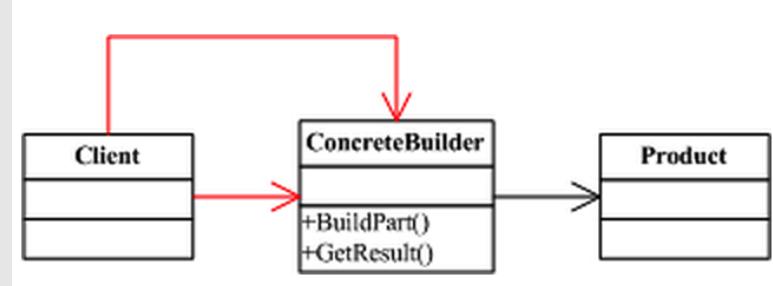
    public function buildDrink() {
        $this->meal->setDrink("一杯紅茶");
    }

    public function buildDessert() {
        $this->meal->setDessert("一個蘋果派");
    }

    public function getMeal() {
        return $this->meal;
    }

    public function buildMeal() {
        $this->buildFood();
        $this->buildDrink();
        $this->buildDessert();

        return $this->getMeal();
    }
}
```



```
include_once '../class/pattern/builder.php';

$beefKitMealBuilder = new BeefKitMealBuilder();
$beefKitMeal = $beefKitMealBuilder->buildMeal();
$beefKitMeal->showMeal();
```



一個牛肉堡, 一杯紅茶, 一個蘋果派

Builder(10/14):範例2-實際應用範例(Java code)

```
public interface SiteMapBuilder {  
  
    public void buildHeader();  
  
    public void buildFooter();  
  
    public void buildPage(URL url);  
  
    public String getSiteMap();  
}
```

Builder

```
public class HtmlSiteMapBuilder implements SiteMapBuilder {  
  
    private String header;  
    private String footer;  
    private List<URL> urls;  
  
    public void buildHeader() {  
        header = "<ul class=\"sitemap\">\n";  
    }  
  
    public void buildFooter() {  
        footer = "</ul>\n";  
    }  
  
    public void buildPage(URL url) {  
        if (urls == null) {  
            urls = new ArrayList<URL>();  
        }  
        urls.add(url);  
    }  
  
    public String getSiteMap() {  
        StringBuffer body = new StringBuffer();  
        for (URL url : urls) {  
            body.append("<url>\n<loc>" + url.toString() + "</loc>\n</url>");  
        }  
        return header + body + footer;  
    }  
}
```

ConcreteBuilder

行動×商務 社群×媒體
mobile e-commerce social media

Builder(11/14):範例2-實際應用範例(Java code)

```
public class GoogleSiteMapBuilder implements SiteMapBuilder {  
  
    private String header;  
    private String footer;  
    private List<URL> urls;  
  
    public void buildHeader() {  
        header = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"  
            + "<urlset xmlns=\"http://www.sitemaps.org/schemas/sitemap/0.9\">\n";  
    }  
  
    public void buildFooter() {  
        footer = "</urlset>\n</xml>\n";  
    }  
  
    public void buildPage(URL url) {  
        if (urls == null) {  
            urls = new ArrayList<URL>();  
        }  
        urls.add(url);  
    }  
  
    public String getSiteMap() {  
        StringBuffer body = new StringBuffer();  
        for (URL url : urls) {  
            body.append("<url>\n<loc>" + url.toString() + "</loc>\n</url>");  
        }  
        return header + body + footer;  
    }  
}
```

ConcreteBuilder

Hiiir

社群×媒體
merce social media

Builder(12/14):範例2-實際應用範例(Java code)

```
public class SiteMapMaker {  
  
    String siteMap;  
  
    public SiteMapMaker(SiteMapBuilder siteMapBuilder) {  
        try {  
            siteMapBuilder.buildHeader();  
            siteMapBuilder.buildFooter();  
            for (int i = 0; i < Data.webpages.length; i++) {  
                siteMapBuilder.buildPage(new URL(Data.webpages[i].getUrl()));  
            }  
        } catch (MalformedURLException e) {  
            e.printStackTrace();  
        }  
        siteMap = siteMapBuilder.getSiteMap();  
    }  
  
    public void saveSiteMapToDisk(File sitemapfile) {  
        FileWriter fileWriter;  
        try {  
            fileWriter = new FileWriter(sitemapfile);  
            fileWriter.write(siteMap.toString());  
            fileWriter.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Director

<https://github.com/dparoulek/java-koans/tree/master/src/main/java/com/upgradingdave/koans/builder>



Builder(13/14): Builder vs. Abstract Factory(1/2)

Abstract Factory: Emphasizes a family of product objects (either simple or complex)

Builder: Focuses on constructing a complex object step by step

Abstract Factory: Focus on *what* is made

Builder: Focus on *how* it is made

Abstract Factory: Focus on defining many different types of *factories* to build many *products*, and it is not a one builder for just one product

Builder: Focus on building a one complex but one single *product*

Abstract Factory: Defers the choice of what concrete type of object to make until run time

Builder: Hide the logic/operation of how to compile that complex object

Abstract Factory: *Every* method call creates and returns different objects

Builder: Only the *last* method call returns the object, while other calls partially build the object

Builder(14/14):Builder vs. Abstract Factory(2/2)

- 與Abstract Factory相比，Builder返回一個組裝好的完整產品，而Abstract Factory返回一系列相關的產品，這些產品位於不同的產品等級結構，構成了一個產品族。
- 在Abstract Factory中，用戶端實例化工廠類，然後調用工廠方法獲取所需產品物件，而在Builder中，用戶端可以不直接調用Builder的相關方法，而是通過Director來指導如何生成物件，包括物件的組裝過程和建造步驟，**它側重於一步一步構造一個複雜物件，返回一個完整的物件**。
- 如果將Abstract Factory看成汽車配件生產工廠，生產一個產品族的產品，那麼Builder就是一個汽車組裝工廠，通過對部件的組裝可以返回一輛完整的汽車。

Prototype(原型模式)

Hiiir

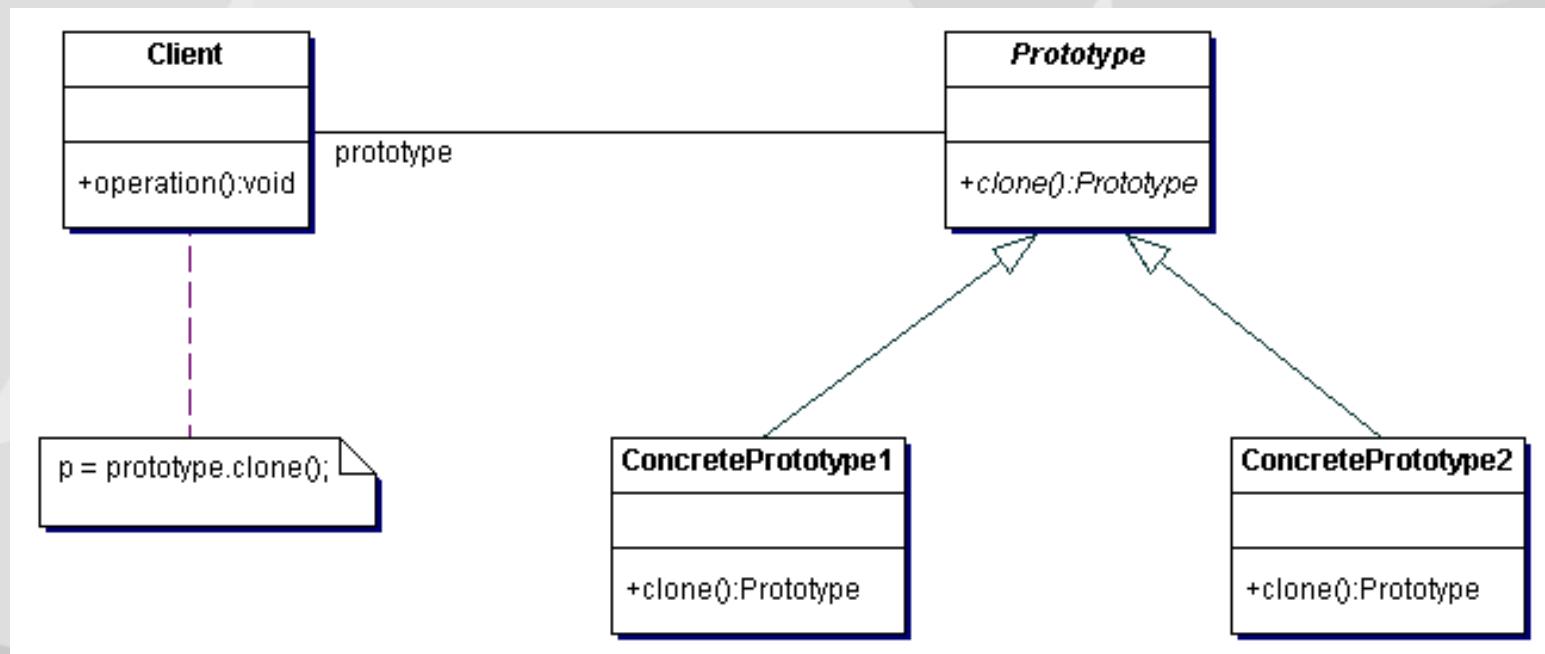
行動×商務 社群×媒體
mobile e-commerce social media

Prototype(1/7):UML

Client：發出創建物件的請求。

Prototype：抽象類別，負責定義所有ConcretePrototype所需的介面(接口)。

Concrete Prototype：被複製的物件，需實作Prototype所定義的介面(接口)。



Prototype(2/7): 模式分析(1/2)

優點：

- 當創建新的物件實例較為複雜時，使用Prototype可以簡化物件的創建過程。在某些環境下，複製已有物件可以比重新建立物件更有效率。
- 可以動態增加或減少產品類。
- 原型模式提供了簡化的創建結構。
- 可以使用深複製(deep clone)的方式保存物件的狀態。

缺點：

- 需要為每一個類別配備一個clone方法，而且這個clone方法需要對類別的功能進行通盤考慮，這對全新的類別來說不是很難，但對已有的類別進行改造時，不一定是件容易的事，必須修改其原始程式碼，違背了“開閉原則”。
- 在實現深複製時需要編寫較為複雜的代碼。

Prototype(3/7): 模式分析(2/2)

使用時機：

- 創建新物件的過程很昂貴或很複雜時，新的物件可以通過**Prototype**模式對已有物件進行複製來獲得，如果是相似物件，則可以對其屬性稍作修改。
- 如果系統要保存物件的狀態，而物件的狀態變化很小，或者物件本身占記憶體不大的時候，也可以使用**Prototype**模式配合**Memento**模式來應用。相反，如果物件的狀態變化很大，或者物件佔用的記憶體很大，那麼採用**State**模式會比**Prototype**模式更好。
- 需要避免使用分層次的工廠類別來創建分層次的物件，並且類別的實例物件只有一個或很少的幾個組合狀態，通過複製原型物件得到新實例可能比使用構造函數創建一個新實例更加方便。

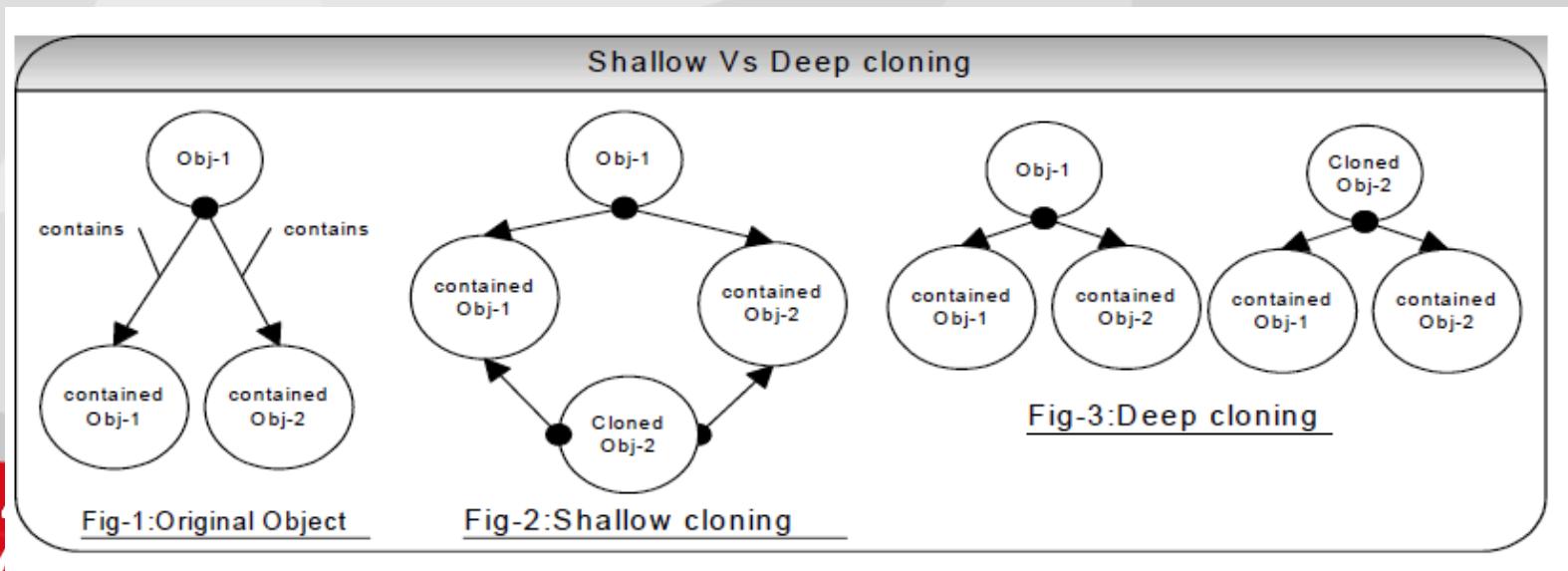
Prototype(4/7): 模式說明(1/1)

淺複製(shallow clone) :

被複製物件的所有變數都含有與原來的物件相同的值，而**所有的對其他物件的引用仍然指向原來的物件**。換言之，淺複製僅僅複製所考慮的對象，而不複製它所引用的對象。

深複製(deep clone) :

被複製物件的所有變數都含有與原來的物件相同的值，除去那些引用其他物件的變數。那些**引用其他物件的變數將指向被複製過的新物件**，而不再是原有的那些被引用的物件。換言之，深複製把要複製的物件所引用的物件都複製了一遍。



Prototype(5/7): 範例(1/3)

```
// Prototype
abstract class BookPrototype
{
    public $subObject1;
    public $subObject2;
    public $title;
    public $topic;

    public function __construct($obj1, $obj2, $title) {
        $this->subObject1 = $obj1;
        $this->subObject2 = $obj2;
        $this->title = $title;
    }

    // Deep Clone
    public function __clone() {
        // Force a copy of $this->object, otherwise it will point to same object.
        $this->subObject1 = clone $this->subObject1;
    }

    public function setTitle($title) {
        $this->title = $title;
    }

    public function show() {
        echo $this->topic . " " . $this->title . "<br>" .
            "object1 Value: " . $this->subObject1->showValue() .
            "object2 Value: " . $this->subObject2->showValue();
    }
}
```

Prototype

```
// ConcretePrototype
class NovelPrototype extends BookPrototype
{
    public function __construct($obj1, $obj2, $title) {
        parent::__construct($obj1, $obj2, $title);
        $this->topic = "Novel";
    }
}

// ConcretePrototype
class ReferenceBookPrototype extends BookPrototype
{
    public function __construct($obj1, $obj2, $title) {
        parent::__construct($obj1, $obj2, $title);
        $this->topic = "ReferenceBook";
    }
}

// 模擬Prototype裡面的子物件
class SubObject
{
    static $instances = 0;
    public $instance;
    public $value;

    public function __construct($value) {
        $this->instance = ++self::$instances;
        $this->value = $value;
    }

    public function __clone() {
        $this->instance = ++self::$instances;
    }

    public function showValue() {
        return $this->value . "<br>";
    }
}
```

ConcretePrototype

Prototype(6/7): 範例(2/3)

```
include_once '../../../../../class/pattern/prototype.php';

$obj1 = new SubObject("obj1");
$obj2 = new SubObject("obj2");

// 產生一個NovelPrototype物件
$novel1 = new NovelPrototype($obj1, $obj2, 'Novel1');
$novel1->show();
print_r($novel1);
echo "<br><br>";

// 複製$novel;
$novel2 = clone $novel1;
$novel2->show();
print_r($novel2);

// 修改novel2的title
$novel2->setTitle('modified Novel2 Title');

echo "<br><br>";
$novel1->show();

echo "<br><br>";
$novel2->show();

echo "<br><br>";
$novel3 = $novel2;
$novel3->show();
print_r($novel3);
```

Novel Novel1
object1 Value: obj1
object2 Value: obj2
NovelPrototype Object ([subObject1] => SubObject Object ([instance] => 1 [value] => obj1)
[subObject2] => SubObject Object ([instance] => 2 [value] => obj2) [title] => Novel1 [topic] =>
Novel)

Novel Novel1
object1 Value: obj1
object2 Value: obj2
NovelPrototype Object ([subObject1] => SubObject Object ([instance] => 3 [value] => obj1)
[subObject2] => SubObject Object ([instance] => 2 [value] => obj2) [title] => Novel1 [topic] =>
Novel)

Novel Novel1
object1 Value: obj1
object2 Value: obj2

Novel modified Novel2 Title
object1 Value: obj1
object2 Value: obj2

Novel modified Novel2 Title
object1 Value: obj1
object2 Value: obj2
NovelPrototype Object ([subObject1] => SubObject Object ([instance] => 3 [value] => obj1)
[subObject2] => SubObject Object ([instance] => 2 [value] => obj2) [title] => modified Novel2
Title [topic] => Novel)

建立了新實體

引用原來物件

Prototype(7/7): 範例(3/3)

PHP物件複製(Clone)規則：

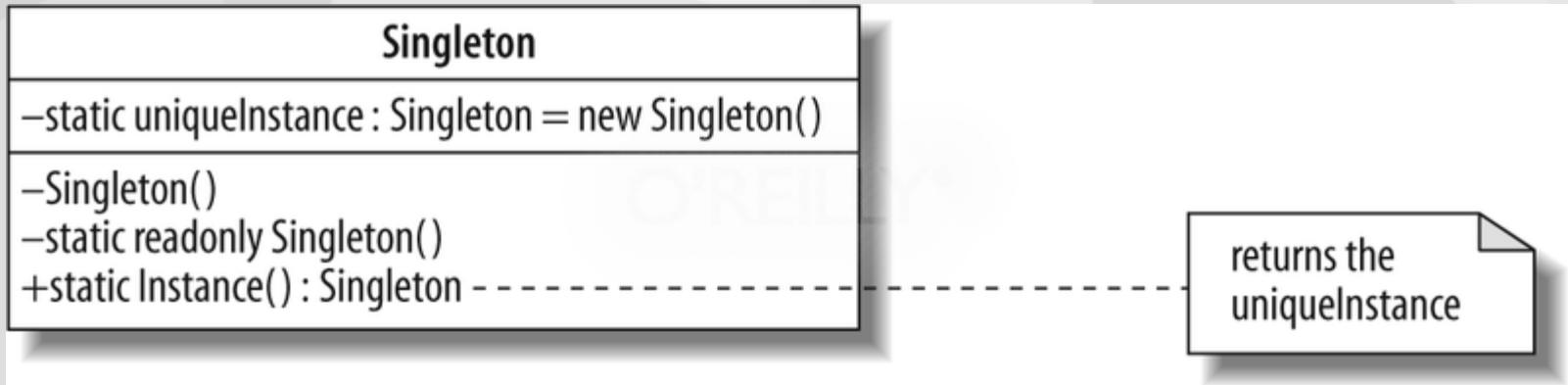
- 有在`_clone()`底下指定複製的子物件，會建立新的實體，反之會引用原本的子物件
- 數值類的變數會直接複製一份，修改複製過後變數的數值不會影響被複製物件的數值
- 使用`assign(=)`，只會引用原本的子物件，即使`_clone()`裡有指定

Singleton(單例模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

Singleton(1/3):UML



Singleton(2/3):模式分析

優點：

- 一種對全域變數的改進
- 一個類別只有一個實例

缺點：

- 在系統任何地方都可以使用，可能會造成很難調整的依賴關係，例如如果改變一個單例物件，那所有使用該單例物件的類別可能都會受到影響。

使用時機：

- 希望確保一個類別只能有一個實體時使用

Singleton(3/3): 範例(1/1)

```
class Preferences
{
    private $props = array();
    private static $instance;

    private function __construct() {
        // 初始化要做的事情
    }

    public static function getInstance() {
        if ( empty( self::$instance ) ) {
            self::$instance = new Preferences();
        }
        return self::$instance;
    }

    public function setProperty( $key, $val ) {
        $this->props[$key] = $val;
    }

    public function getProperty( $key ) {
        return $this->props[$key];
    }
}
```

```
include_once '../../../../../class/pattern.singleton.php';

$pref = Preferences::getInstance();

$pref->setProperty("name", "matt");

unset($pref);

$pref2 = Preferences::getInstance();
echo $pref2->getProperty("name");
```



matt

其實..單例模式在PHP沒想像中好用...

單例模式在PHP沒想像中好用

- (a) PHP operates like most other CGI implementations. When the webserver receives a request for a specific URL that happens to be a .cgi/.php script, then the interpreter is invoked with the referenced script. The PHP interpreter runs it, and sends the output back to the webserver/client. After it's finished, that CGI interpreter terminates and takes all its runtime data with it. That explains why variables do not persist or are shared between two distinct PHP scripts or invocations. It works the same for the ordinary mod_php handler. Only there it's simple process forking (duplication) and termination.
- (b) The \$_SESSION array is handled by PHP. It can discern the per-user storage using the unique cookie key. And since each CGI or mod_php process does have a separate variable pool, there is really no issue. PHP just needs a bit of file locking to prevent two scripts from overwriting the session store when the same user requested two scripts at once.
- (c) "Global" variables also only exist per-process. They are gone like everything else when the PHP script has finished.

單例模式在PHP沒想像中好用

- 簡單來說，在同一個request中，單例模式的物件跟Global物件是有效的，但是request結束後所有資料就會消失。
- 所以如果有變數要在不同的request中使用，建議還是用\$_SESSION來儲存。

request1

```
include_once '.../class/pattern.singleton.php';

$pref = Preferences::getInstance();
$pref->setProperty( "name", "matt" );
unset($pref);

$pref2 = Preferences::getInstance();
echo $pref2->getProperty("name");
```

在同一request中，
我們可以取到matt

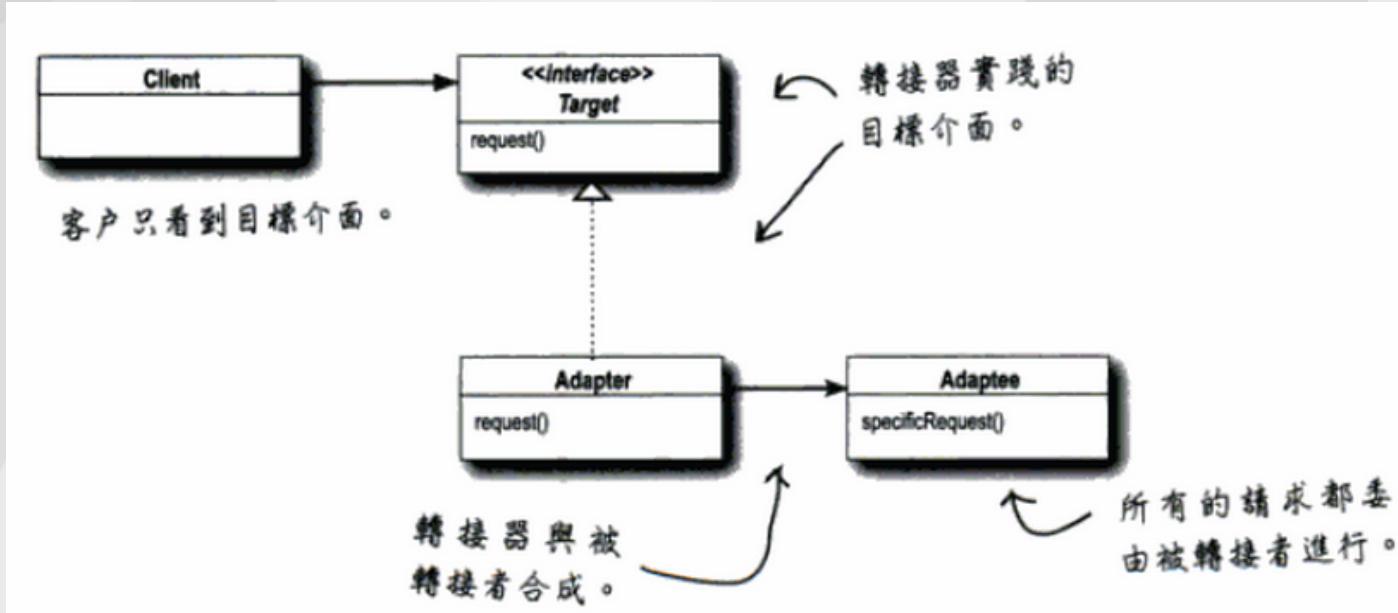
request2

如果我們用瀏覽器打開另一隻php，同樣建立單例物件，會發現matt消失了。因為當request結束後，物件就會自動消失，所以在request2，等於又重新建了一個單例物件。



Adapter(轉接器模式)

Adapter(1/11):UML(1/2)



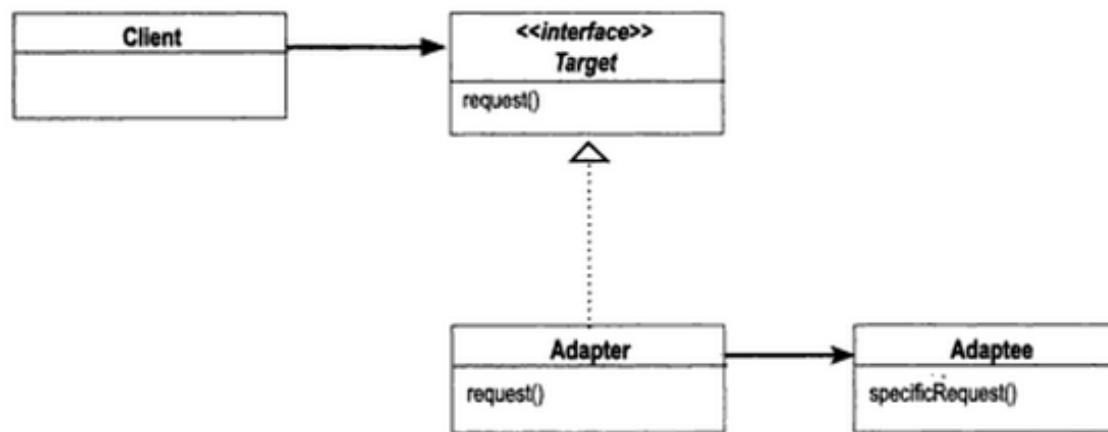
Adapter(2/11):UML(2/2)

類別轉接器



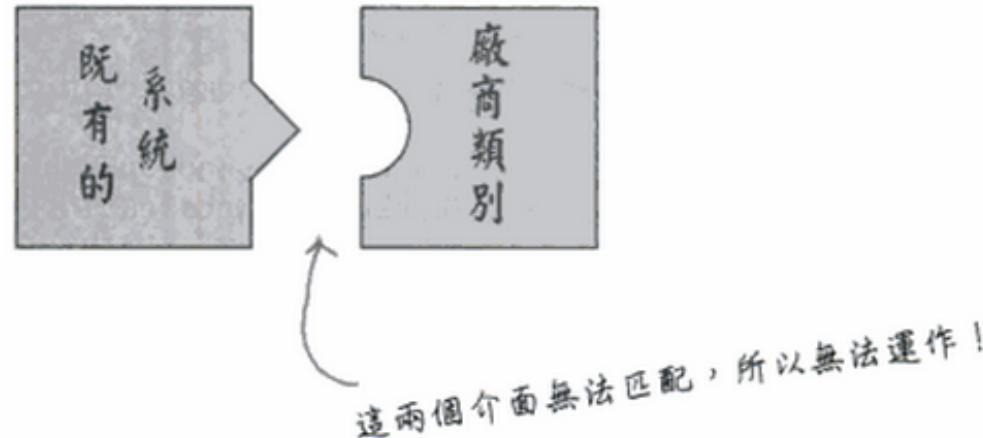
PHP不支援多重繼承，
所以類別轉接器不可用

物件轉接器



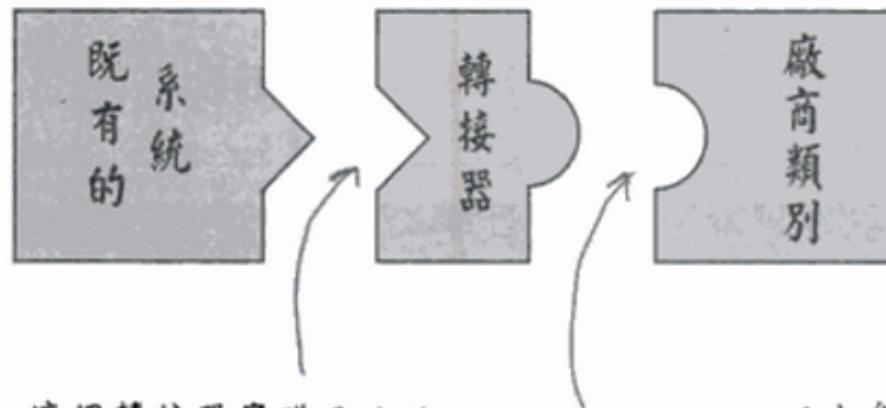
Adapter(3/11): 模式說明(1/4)

假設你擁有一個軟體系統，希望它能和一個新的廠商類別庫搭配使用，但是這個新廠商所設計出來的介面，不同於舊廠商的介面：



Adapter(4/11): 模式說明(2/4)

你不想改變既有的程式碼，解決這個問題（而你也不能改變廠商的程式碼）。所以該怎麼做？這個嘛，你可以寫一個類別，將新廠商介面轉接成你所期盼的介面。

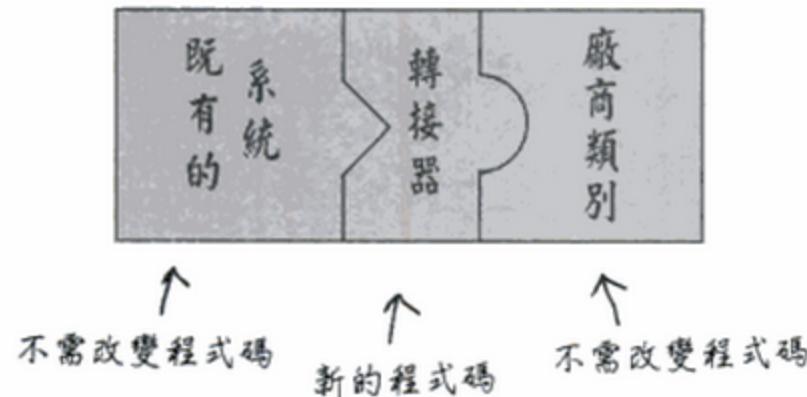


這個轉接器實踐了你的
系統所期盼的介面。

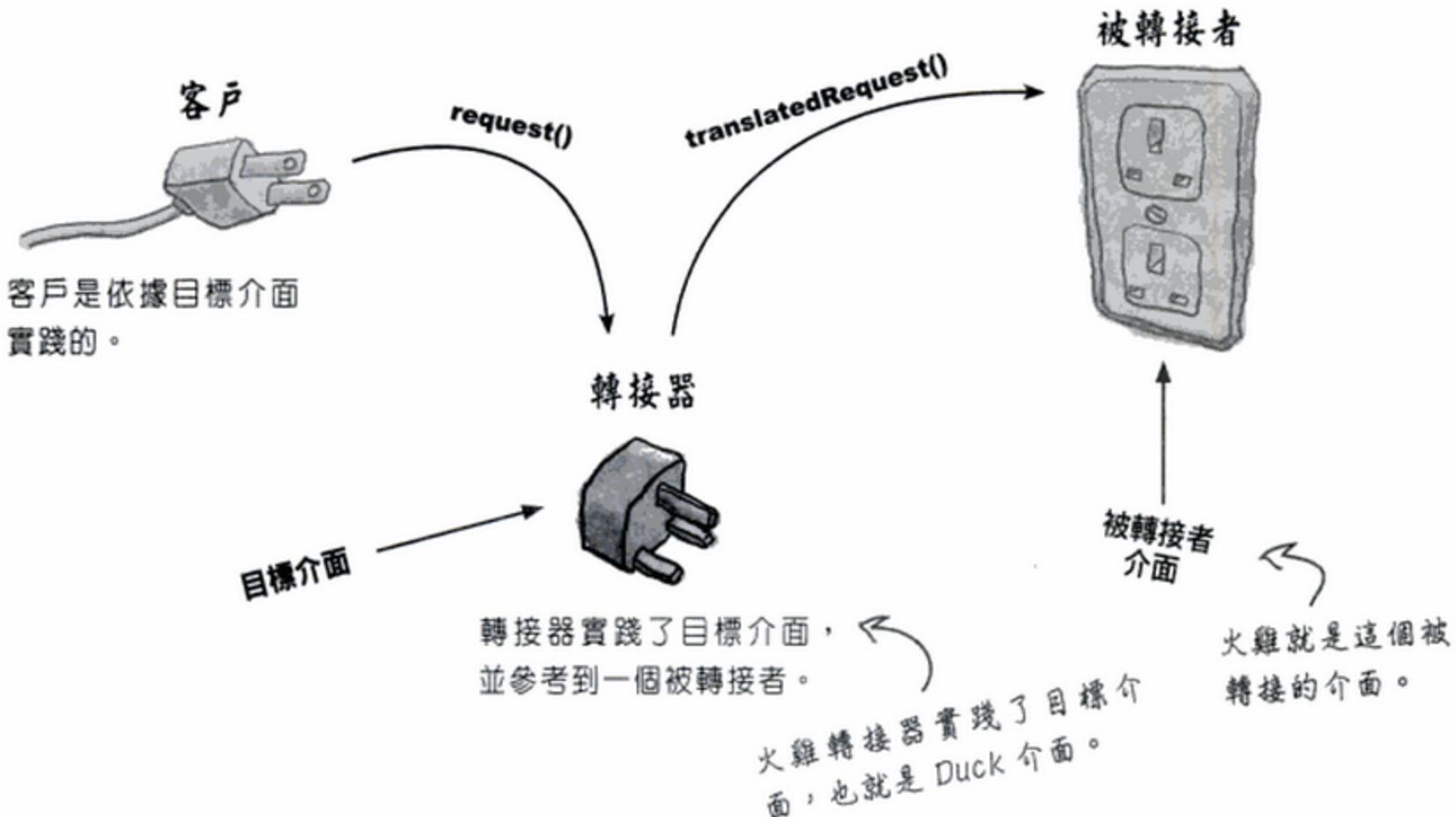
而且這個轉接器也能
和廠商的介面溝通。

Adapter(5/11): 模式說明(3/4)

這個轉接器運作起來就如同是一個中間人，它將客戶所發出的請求轉換成廠商類別所看得懂的請求。



Adapter(6/11): 模式說明(4/4)



Adapter(7/11): 模式分析

優點：

- 將目標類別和被轉接類別解耦，通過引入一個轉接器類別來重用現有的被轉接類別，而無須修改原有代碼。
- 增加了類別的透明性和複用性，將具體的實現封裝在轉接器中，對於用戶端類來說是透明的，而且提高了轉接器的複用性。
- 靈活性和擴展性都非常好，通過使用設定檔，可以很方便地更換轉接器，也可以在不修改原有代碼的基礎上增加新的轉接器類別，完全符合“開閉原則”。

缺點：

- 對於Java、C#等不支援多重繼承的語言，一次最多只能使用一個Adapter類別，而且目標抽象類別只能為抽象類別，不能為具體類，其使用有一定的局限性，不能將一個Adapter類別和它的子類都適配到目標介面。

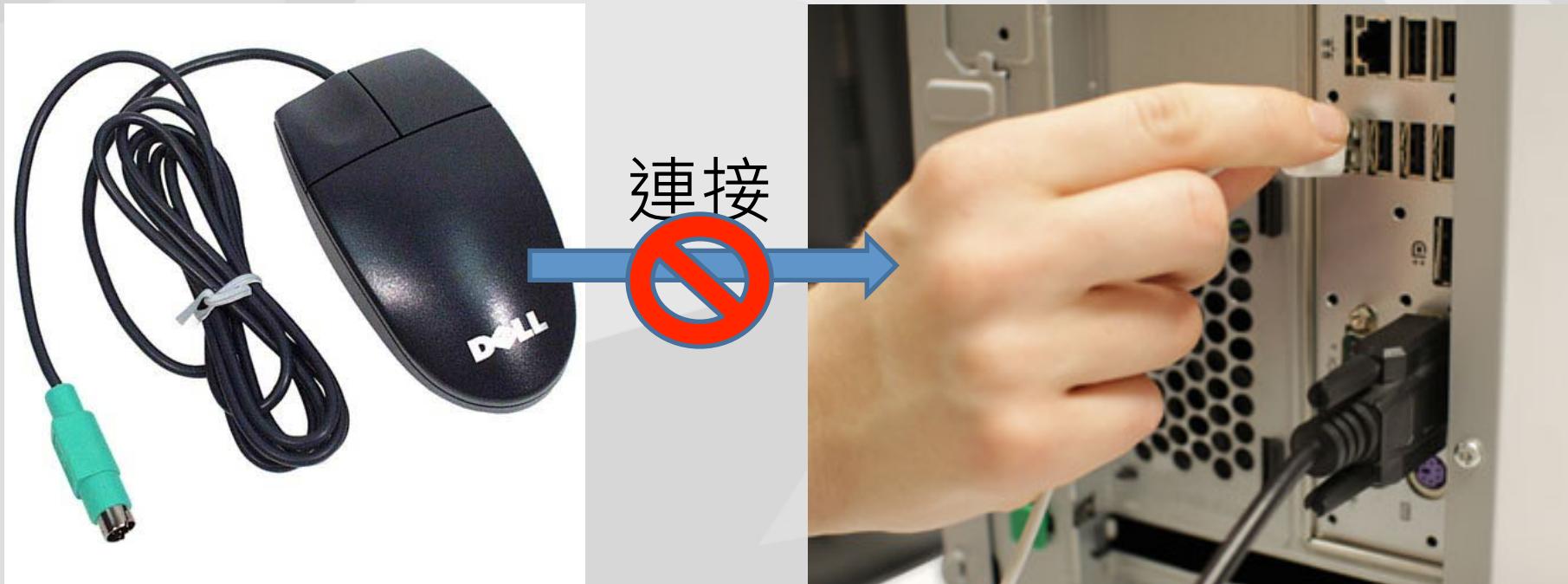
使用時機：

- 需要將一個類別的介面，轉換成另一個介面以供客戶使用。
- 系統需要使用現有的類別，而這些類別的介面不符合系統的需要



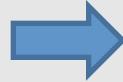
Adapter(8/11):範例(1/4)

假設我們現在有一隻ps/2介面的滑鼠，要給只有usb介面的電腦使用，該怎麼做呢？



Adapter(9/11):範例(2/4)

用個Adapter就好啦～～～！



Adapter(10/11): 範例(3/4)

```
// USB滑鼠介面
interface UsbMouse
{
    public function usb_click();
    public function usb_move();
    public function usb_connect();
}

// 簡單USB滑鼠
class SimpleUsbMouse implements UsbMouse
{
    public function usb_click() {
        echo "USB滑鼠點一下<br/>";
    }

    public function usb_move() {
        echo "USB滑鼠移動<br/>";
    }

    public function usb_connect() {
        echo "USB滑鼠連接<br/>";
    }
}
```

```
// PS/2滑鼠介面
interface Ps2Mouse
{
    public function ps2_click();
    public function ps2_move();
    public function ps2_connect();
}

// 簡單PS/2滑鼠
class SimplePs2Mouse implements Ps2Mouse
{
    public function ps2_click() {
        echo "PS/2滑鼠點一下<br/>";
    }

    public function ps2_move() {
        echo "PS/2滑鼠移動<br/>";
    }

    public function ps2_connect() {
        echo "PS/2滑鼠連接<br/>";
    }
}
```

Adaptee

```
// PS/2 -> USB轉接器
class Ps2UsbAdapter implements UsbMouse
{
    private $ps2Mouse;

    public function __construct(Ps2Mouse $ps2Mouse) {
        $this->ps2Mouse = $ps2Mouse;
    }

    public function usb_click() {
        $this->ps2Mouse->ps2_click();
    }

    public function usb_move() {
        $this->ps2Mouse->ps2_move();
    }

    public function usb_connect() {
        $this->ps2Mouse->ps2_connect();
    }
}
```

Adapter

Adapter(11/11): 範例(4/4)

```
include_once '../../../../../class/pattern/adapter.php';

$ps2Mouse = new SimplePs2Mouse(); // 建立一個ps/2介面的滑鼠

$ps2UsbAdapter = new Ps2UsbAdapter($ps2Mouse); // 將ps/2介面滑鼠接到轉接器中

// 現在ps2滑鼠可用usb介面操作
$ps2UsbAdapter->usb_connect();
$ps2UsbAdapter->usb_click();
$ps2UsbAdapter->usb_move();
```



PS/2滑鼠連接
PS/2滑鼠點一下
PS/2滑鼠移動

Bridge(橋梁模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

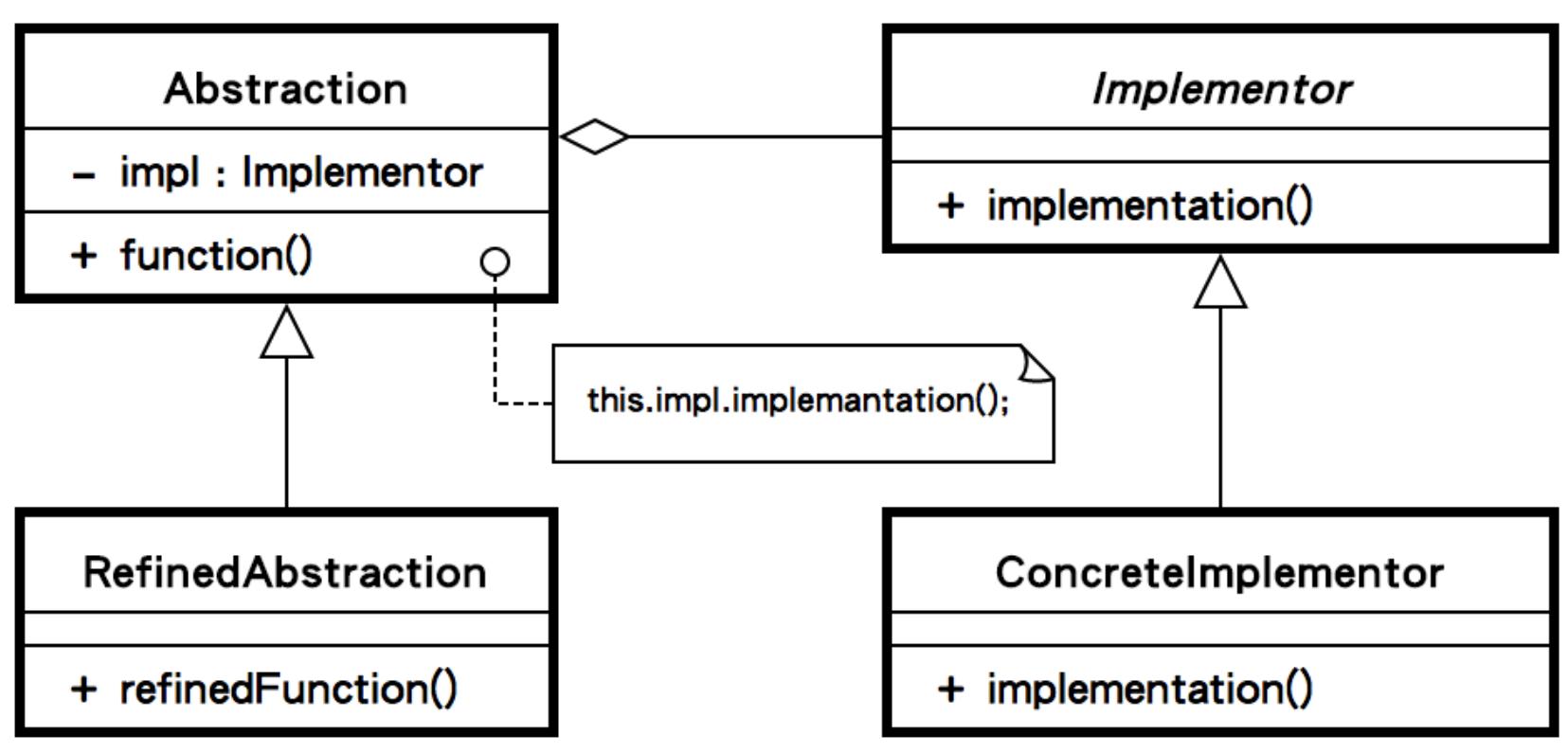
Bridge(1/11): UML(1/1)

Abstraction : 抽象化角色，負責保存一個對實現化對象的引用。

Refined Abstraction : 擴展抽象化角色，改變和修正父類別對抽象化的定義。

Implementor : 代表給出實現化角色的接口，但不提供具體的實現。

Concrete Implementor : 約定實現化角色接口的具體實現。



Bridge(2/11): 模式分析(1/2)

優點：

- Bridge分離了抽象部分和實現部分，極大的提高了系統的靈活性。讓抽象部分和實現部分獨立開來。分別定義接口，這有助于對系統進行分層，從而產生更好的結構化的系統。對於系統的高層部分，只需要知道抽象部分和實現部分的接口就可以了。
- 由於Bridge把抽象部分和實現部分分離開了，而且分別定義接口，這就使得抽象部分和實現部分可以分別獨立的擴展，而不會互相影響，從而大大地提高了系統的可擴展性。
- 由於Bridge把抽象部分和實現部分分離開了，所以在實現Bridge的時候，就可以動態的選擇和使用具體的實現。
- 可以明顯減少子類別的數量。

缺點：

- Bridge的引入會增加系統的理解與設計難度，由於聚合關聯關係建立在抽象層，要求開發者針對抽象進行設計與程式設計。
- Bridge要求正確識別出系統中兩個獨立變化的維度，因此其使用範圍具有一定的局限性。



Bridge(3/11): 模式分析(2/2)

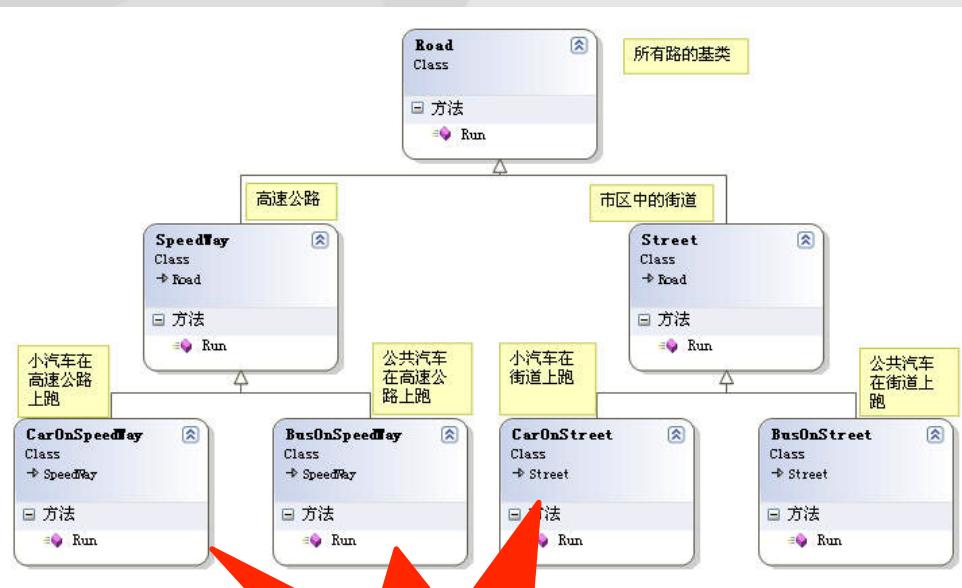
使用時機：

- 如果一個系統需要在構件的抽象化角色和具體化角色之間增加更多的靈活性，避免在兩個層次之間建立靜態的繼承聯繫，通過Bridge可以使它們在抽象層建立一個關聯關係。
- 抽象化角色和實現化角色可以以繼承的方式獨立擴展而互不影響，在程式運行時可以動態將一個抽象化子類的物件和一個實現化子類的物件進行組合，即系統需要對抽象化角色和實現化角色進行動態耦合。
- 一個類存在兩個獨立變化的維度，且這兩個維度都需要進行擴展。
- 雖然在系統中使用繼承是沒有問題的，但是由於抽象化角色和具體化角色需要獨立變化，設計要求需要獨立管理這兩者。
- 對於那些不希望使用繼承或因為多層次繼承導致系統類別的個數急劇增加的系統，Bridge模式尤為適用。
- 相當適合使用在需要跨越多個平台的圖形和視窗系統上

Bridge(4/11): 模式說明(1/1)

簡單來說，Bridge就是要利用組合物件代替繼承物件，已將各物件之間的關係解耦以提高系統的擴展性。

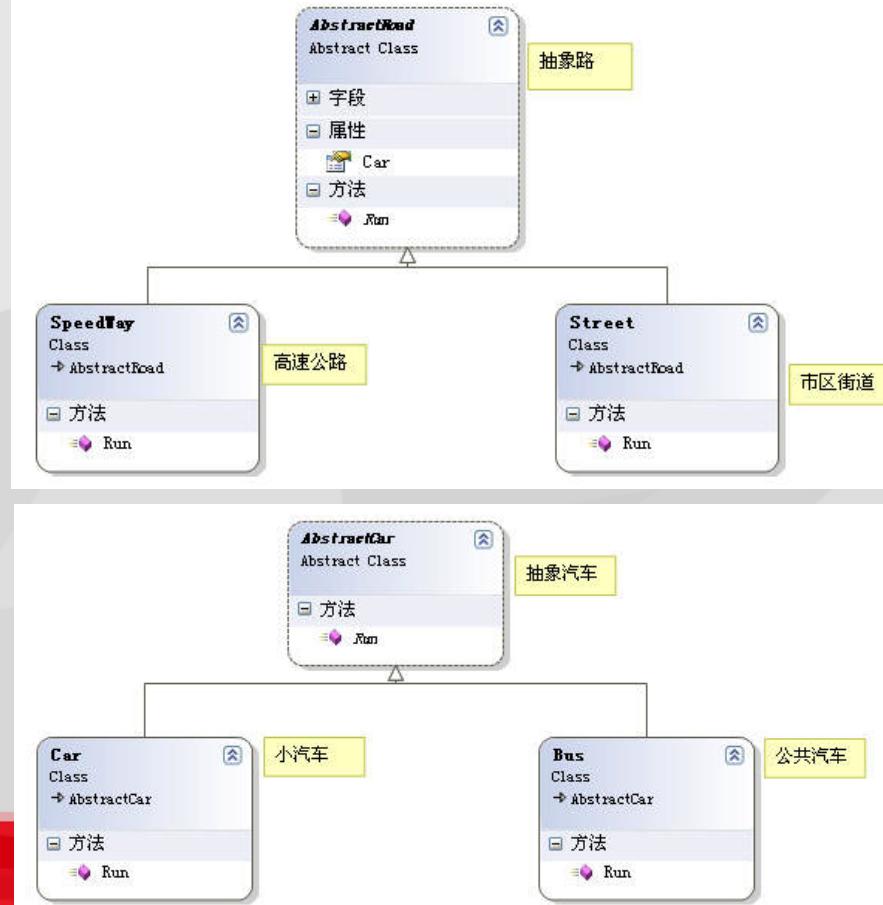
繼承



不符合
OCP原則!

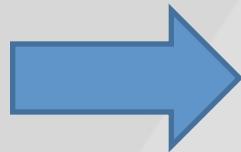
Hiiir

組合



Bridge(5/11):範例(1/1)

假設我們要做一個偽·憤怒鳥遊戲，它有多個版本並且需要在Android跟iOS平台上運作。



iOS



Hiiir

mobile e-commerce

群×媒體
social media

Bridge(6/11): 反面範例(1/4)

```
interface AngryBird
{
    public function play();
}

class AndroidAngryBirdNormal implements AngryBird
{
    public function play() {
        echo "您開始玩Android平台的AngryBird普通版本<br>";
    }
}

class AndroidAngryBirdSpace implements AngryBird
{
    public function play() {
        echo "您開始玩Android平台的AngryBird宇宙版本<br>";
    }
}

class AndroidAngryBirdRio implements AngryBird
{
    public function play() {
        echo "您開始玩Android平台的AngryBirdRio版本<br>";
    }
}
```

```
class IOSAngryBirdNormal implements AngryBird
{
    public function play() {
        echo "您開始玩iOS平台的AngryBird普通版本<br>";
    }
}

class IOSAngryBirdSpace implements AngryBird
{
    public function play() {
        echo "您開始玩iOS平台的AngryBird宇宙版本<br>";
    }
}

class IOSAngryBirdRio implements AngryBird
{
    public function play() {
        echo "您開始玩iOS平台的AngryBirdRio版本<br>";
    }
}
```

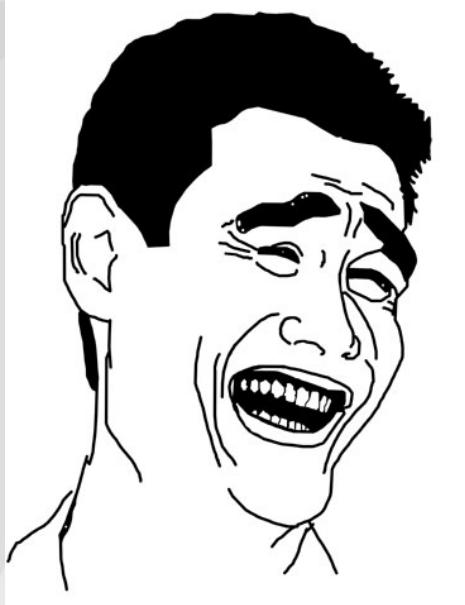
Bridge (7/11): 反面範例(2/4)

```
<?php  
  
    include_once '../class/pattern/bridge.php';  
  
    $androidAngryBirdNormal = new AndroidAngryBirdNormal();  
    echo $androidAngryBirdNormal->play();  
  
    echo '<br/>';  
  
    $iosAngryBirdSpace = new iosAngryBirdSpace();  
    echo $iosAngryBirdSpace->play();  
?>
```



您開始玩Android平台的AngryBird普通版本
您開始玩iOS平台的AngryBird宇宙版本

Bridge (8/11): 反面範例(3/4)



看來沒啥問題嘛～
運作得相當好啊！

Bridge (9/11): 反面範例(4/4)

需求變更來了：

1. 增加支援平台web, 單機版
2. 增加聖誕節、萬聖節特別版本的遊戲



實現部分類別從原本的
 $2 \times 3 = 6$ 個 變為
 $4 \times 5 = 20$ 個....會讓你很舒服

Bridge (10/11): 範例(1/2)

```
// implementor
interface Platform
{
    public function control();
}

// concrete implementor
class AndroidPlatform implements Platform
{
    public function control() {
        return "Android";
    }
}

// concrete implementor
class IOSPlatform implements Platform
{
    public function control() {
        return "IOS";
    }
}
```

```
// abstraction
abstract class AngryBird
{
    protected $platform = null;

    public function __construct(Platform $platform) {
        $this->platform = $platform;
    }

    public abstract function play();
}

// refined abstraction
class AngryBirdNormal extends AngryBird
{
    public function play() {
        echo "您開始玩AngryBird普通版本，平台為:" . $this->platform->control() . "<br>";
    }
}

// refined abstraction
class AngryBirdSpace extends AngryBird
{
    public function play() {
        echo "您開始玩AngryBird宇宙版本，平台為:" . $this->platform->control() . "<br>";
    }
}

// refined abstraction
class AngryBirdRio extends AngryBird
{
    public function play() {
        echo "您開始玩AngryBirdRio版本，平台為:" . $this->platform->control() . "<br>";
    }
}
```

利用Bridge將平台與遊戲解耦



Bridge (11/11): 範例(2/2)

```
include_once '../../../../../class/pattern/bridge.php';

$androidPlatform = new AndroidPlatform();
$iosPlatform = new IOSPlatform();

$angryBirdNormalAndroid = new AngryBirdNormal($androidPlatform);
$angryBirdNormalAndroid->play();

$angryBirdSpaceIOS = new AngryBirdSpace($iosPlatform);
$angryBirdSpaceIOS->play();
```



您開始玩AngryBird普通版本, 平台為:Android
您開始玩AngryBird宇宙版本, 平台為:IOS

需求變更來了:

1. 增加支援平台web, 單機版
2. 增加聖誕節、萬聖節特別版本的遊戲

實現類別從
原本的 $2 + 3 = 5$ 個
變為 $4 + 5 = 9$ 個

Composite(組合模式)

Hiiir

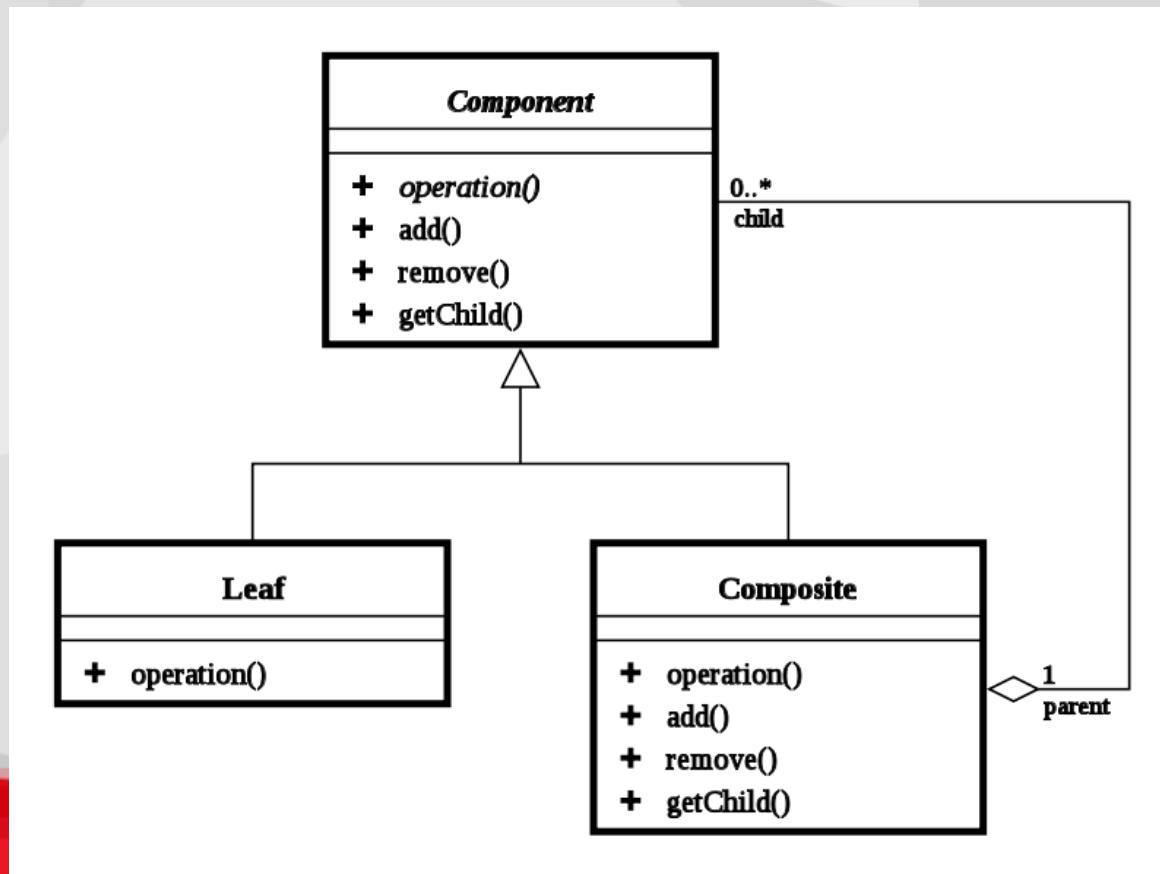
行動×商務 社群×媒體
mobile e-commerce social media

Composite(1/7): UML(1/1)

Component: 一個抽象角色，它給參加組合的物件規定一個介面。給出共有的介面及其預設行為。

Leaf: 代表參加組合的樹葉物件。一個樹葉沒有下級的子物件，定義出參加組合的原始物件的行為。D=

Composite: 代表參加組合的有子物件的物件，並給出樹枝構件對象的行為。



Composite(2/7): 模式分析(1/1)

優點：

- 可以讓我們用樹狀的方式建立物件結構，裡面包含合成和個別的物件
- 可以讓我們用相同的操作處裡合成以及個別的物件，也就是我們可以忽略合成與個別物件之間的差異

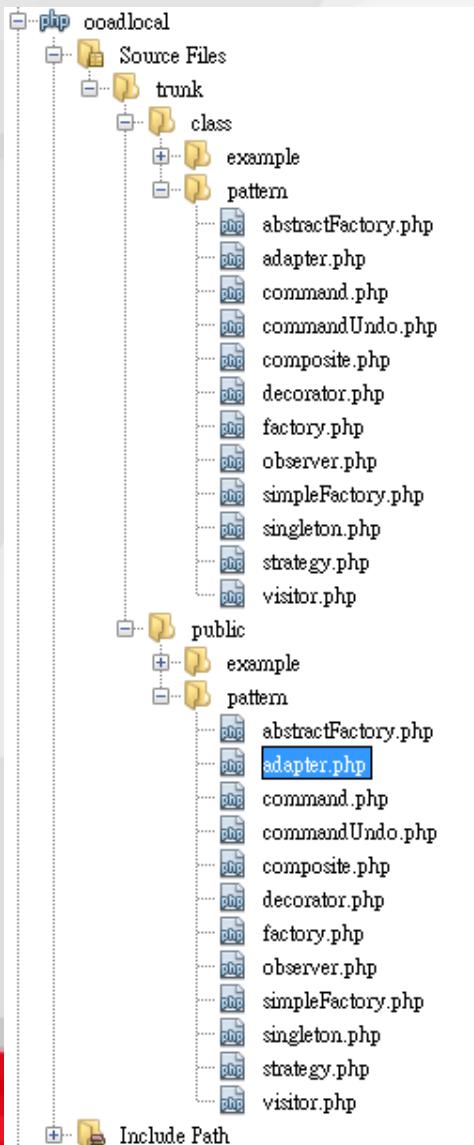
缺點：

- 使用組合模式後，控制樹枝構件的類型不太容易
- 用繼承的方法來增加新的行為很困難
- 當特殊對象或規則越來越多，組合模式就會需要越來越多的類別檢查
- 當子物件的方法會造成大量的系統開銷，用組合操作可能會造成系統崩潰

使用時機：

- 當有數個物件的聚合，它們之間有部分-整體的關係，並且想用一致的方式操作每個元素時

Composite(3/7): 模式說明(1/1)



簡單來說，當你的資料結構為樹狀結構，而且資料間有一致的操作方法，那可以考慮用組合模式。

Composite(4/7): 範例(1/4)

假設我們要做一個偽·世紀帝國的即時戰略遊戲，玩家可以在一個大量區塊組成的地圖上移動戰鬥單位。獨立的單位可以被組合起來一起移動、戰鬥和防守。

我們有以下幾個單位：

攻擊力:4



攻擊力:15



攻擊力:44



弓箭手

攻擊力:15, 不可載運騎兵



www.shutterstock.com · 35511682

騎兵

雷射加農砲

裝甲運兵車

Composite(5/7): 範例(2/4)

```
abstract class Unit
{
    public function getComposite() {
        return null;
    }

    abstract function bombardStrength(); // 輸出戰鬥單位的攻擊力
}

abstract class CompositeUnit extends Unit
{
    private $units = array();

    public function getComposite() {
        return $this;
    }

    protected function units() {
        return $this->units;
    }

    public function addUnit(Unit $unit) { // 將一戰鬥單位加入到軍隊群組中
        if (in_Array($unit, $this->units, true)) {
            return;
        }
        $this->units[] = $unit;
    }

    public function removeUnit(Unit $unit) { // 將一戰鬥單位從軍隊群組中移除
        //$this->units = array_udiff($this->units, array($unit), function($a, $b){ return ($a == $b)?0:1; }); // PHP 5.3寫法
        $this->units = array_udiff($this->units, array($unit), create_function('$a, $b', 'return ($a == $b)?0:1;'));
    }
}
```



Composite(6/7): 範例(3/4)

```
// 軍隊
class Army extends CompositeUnit
{
    public function bombardStrength() { // 計算總攻擊力
        $ret = 0;
        foreach ($this->units() as $unit) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }

// 裝甲運兵車
class TroopCarrier extends CompositeUnit
{
    public function addUnit(Unit $unit) { // 將單位加入
        if ($unit instanceof Cavalry) {
            throw new Exception("Can't get a horse on the vehicle");
        }
        parent::addUnit($unit);
    }

    public function bombardStrength() {
        return 15;
    }
}
```

```
// 弓箭手
class Archer extends Unit
{
    public function bombardStrength() {
        return 4;
    }
}

// 雷射加農砲
class LaserCannonUnit extends Unit
{
    public function bombardStrength() {
        return 44;
    }
}

// 騎兵
class Cavalry extends Unit
{
    public function bombardStrength() {
        return 15;
    }
}
```

Composite(7/7): 範例(4/4)

```
include_once ' ../../class/pattern/composite.php';

$archer = new Archer();
echo "archer attacking with strength: {$archer->bombardStrength()}" . "<br/>";

$main_army = new Army();

$main_army->addUnit($archer);
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCannonUnit());

echo "main_army attacking with strength: {$main_army->bombardStrength()}" . "<br/>";

$sub_army = new Army();

$sub_army->addUnit(new Archer());
$sub_army->addUnit(new Cavalry());
$sub_army->addUnit(new Cavalry());
echo "sub_army attacking with strength: {$sub_army->bombardStrength()}" . "<br/>";

$main_army->addUnit($sub_army);

echo "main_army attacking with strength: {$main_army->bombardStrength()}" . "<br/>";

$troop = new TroopCarrier();

// $troop->addUnit(new Cavalry());
$troop->addUnit(new Archer());
$troop->addUnit(new Archer());
// $troop->addUnit(new Cavalry());

echo "troop attacking with strength: {$troop->bombardStrength()}" . "<br/>";

$main_army->addUnit($troop);

echo "attacking with strength: {$main_army->bombardStrength()}" . "<br/>";
```



archer attacking with strength: 4
main_army attacking with strength: 52
sub_army attacking with strength: 34
main_army attacking with strength: 86
troop attacking with strength: 15
attacking with strength: 101

Decorator(裝飾模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

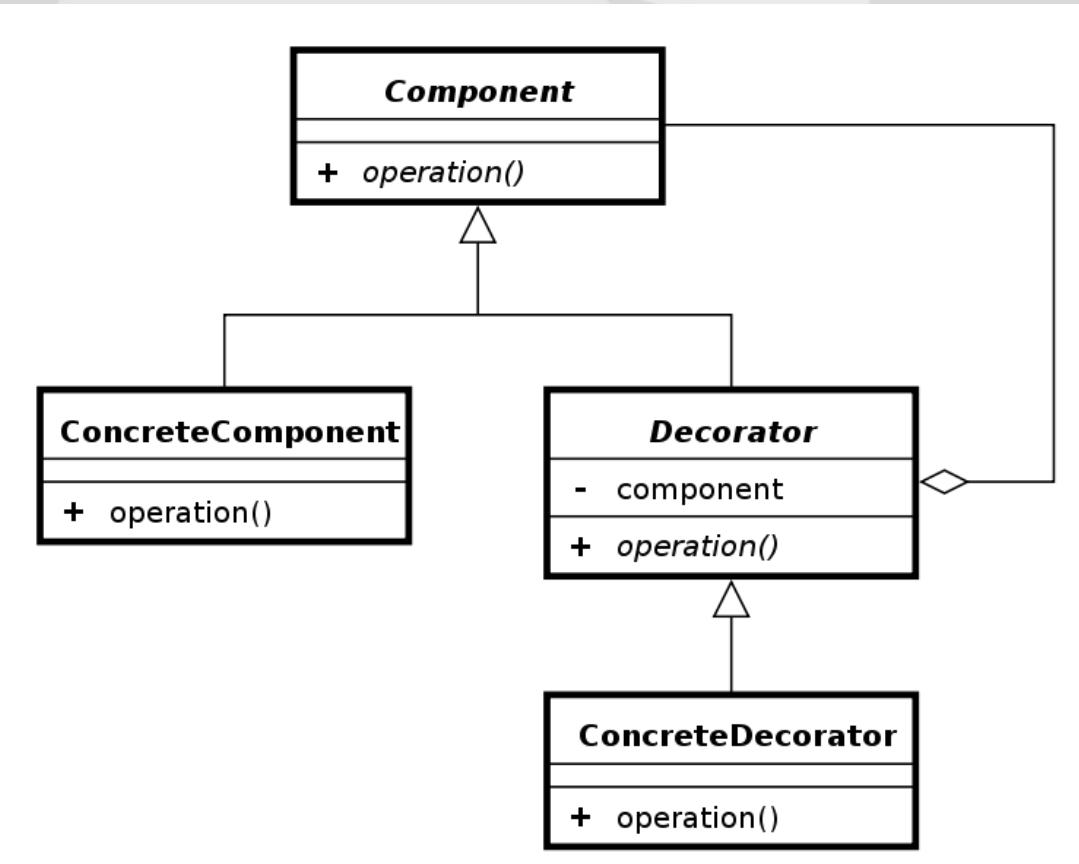
Decorator(1/6): UML(1/1)

Component：給出一個抽象介面，以規範準備接收附加功能的物件

ConcreteComponent：定義一個將要接收附加功能物件的行為

Decorator：持有一個Component物件，並定義一個與抽象構件介面一致的介面。

ConcreteDecorator：負責給物件加上附加功能



Decorator(2/6)：模式分析(1/1)

優點：

- 提供比繼承更多的靈活性
- 使用不同的裝飾組合可以創造出不同行為的組合

缺點：

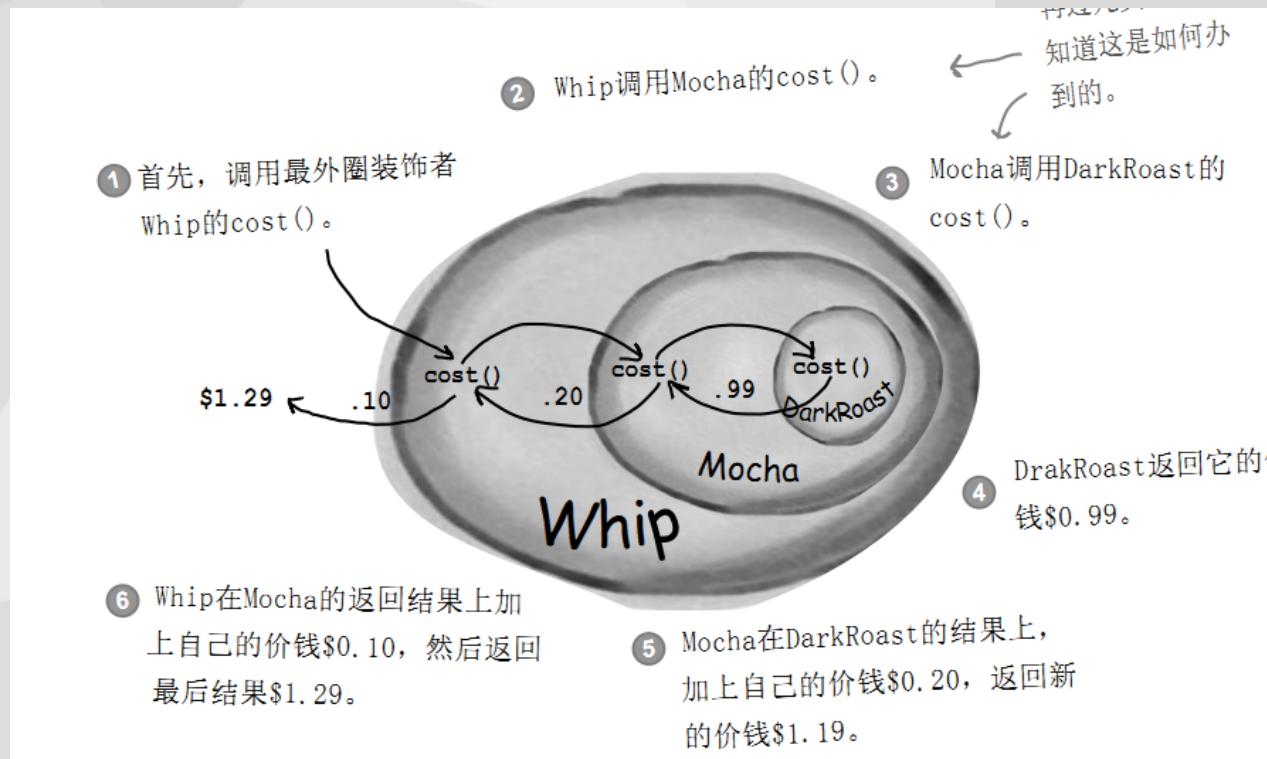
- 靈活性帶來比較大的出錯性
- 裝飾鏈不能過長，否則會影響效率

使用時機：

- 在不影響其他物件的情況下，動態透明的增加責任或功能到某一物件
- 這些功能可以隨時添加或取消
- 系統需要新增新功能，但不想因為過多的子類別繼承造成系統快速膨脹

Decorator(3/6): 模式說明(1/1)

程式運作原理



Decorator(4/6): 範例(1/3)

假設我們要做一個偽·模擬城市的遊戲，地圖上每一單元的土地都有其財富係數，然後我們需要製造一些土地上有自然資元和人類製造的污染的情形，該怎麼做呢？



土地上有鑽石
財富係數+2



土地被汙染了
財富係數-4

Decorator(5/6): 範例(2/3)

```
abstract class Tile
{
    public abstract function getWealthFactor();
}

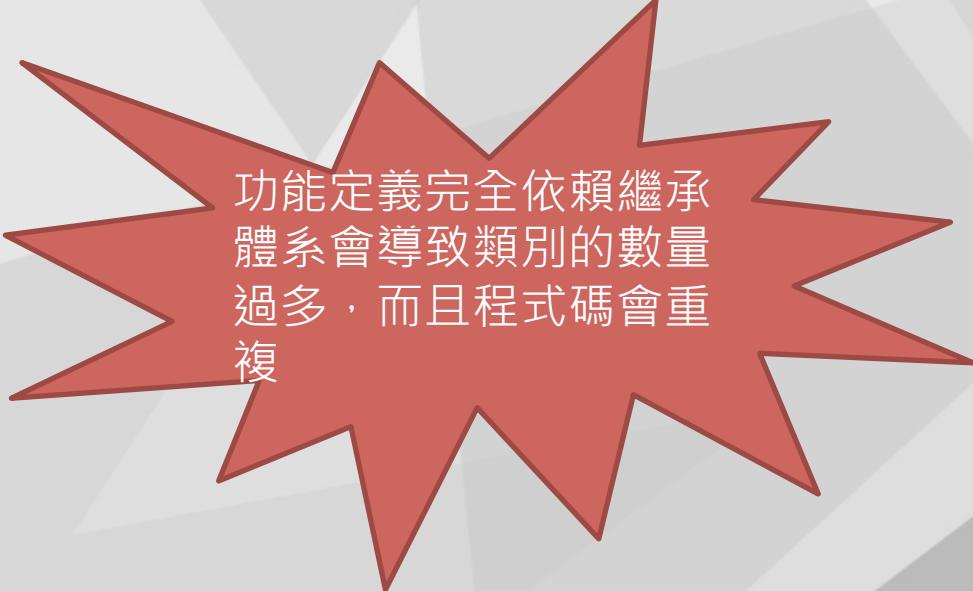
class Plains extends Tile
{
    private $wealthfactor = 2;

    public function getWealthFactor() {
        return $wealthfactor;
    }
}

// 有鑽石
class DiamondPlains extends Plains
{
    public function getWealthFactor() {
        parent::getWealthFactor() + 2;
    }
}

// 被汙染
class PollutedPlains extends Plains {
    public function getWealthFactor() {
        parent::getWealthFactor() - 4;
    }
}
```

用繼承的方式實作...
如果要表達平原又有鑽石又有汙染的情況?!!



功能定義完全依賴繼承
體系會導致類別的數量
過多，而且程式碼會重
複

Decorator(6/6):範例(3/3)

```
abstract class Tile Component
{
    public abstract function getWealthFactor();
}

// 平原
class Plains extends Tile
{
    private $wealthfactor = 2;

    public function getWealthFactor() {
        return $this->wealthfactor;
    }
}

Decorator

abstract class TileDecorator extends Tile
{
    protected $tile;

    public function __construct(Tile $tile){
        $this->tile = $tile;
    }
}

// 鑽石裝飾
class DiamondDecorator extends TileDecorator
{
    public function getWealthFactor() {
        return $this->tile->getWealthFactor() + 2;
    }
}

// 汚染裝飾
class PollutionDecorator extends TileDecorator
{
    public function getWealthFactor() {
        return $this->tile->getWealthFactor() - 4;
    }
}
```



```
include_once '../../class/pattern/decorator.php';

$tile = new Plains();

echo $tile->getWealthFactor() . "<br>";

$tile = new DiamondDecorator(new Plains()); // 平原+鑽石資源
echo $tile->getWealthFactor() . "<br>";

$tile = new PollutionDecorator(new DiamondDecorator(new Plains())); // 平原+鑽石資源+汙染
echo $tile->getWealthFactor() . "<br>";
```



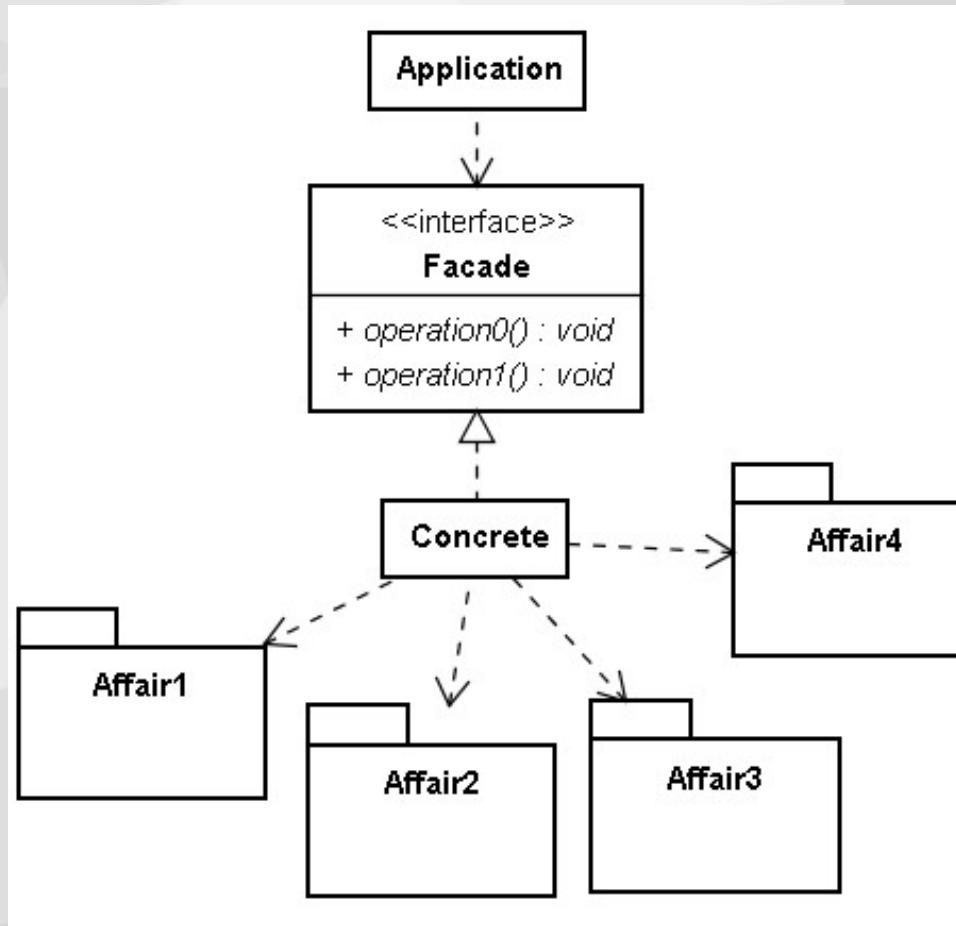
```
2  
4  
0
```

Façade(表象模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

Façade(1/15): UML(1/1)



Façade(2/15)：模式分析(1/1)

優點：

- 減少系統的相互依賴。調用者通過**Façade**提供的介面訪問子系統，這樣只要**Façade**提供的介面不發生變化，子系統怎麼改變都不會影響客戶端代碼
- 提高了安全性。**Façade**可以限制外部能訪問到的子系統功能，只需要為需要的功能提供一個介面即可，沒有提供介面則就不能訪問

缺點：

- 增加新的子系統可能需要修改**Façade**或客戶端的代碼，違背了“開閉原則”
- 過多的或者是不太合理的Facade也容易讓人迷惑，到底是調用Facade好呢，還是直接調用模組好

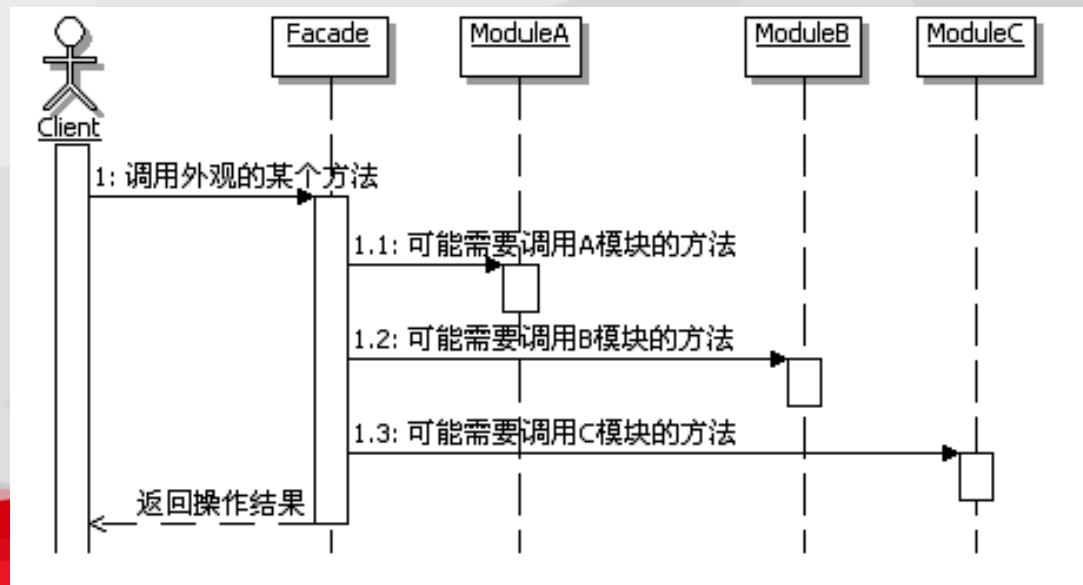
使用時機：

- 當只需使用某複雜系統的部份功能
- 希望封裝或隱藏系統
- 希望為系統增加新的功能
- 使用此模式所需的代價小於學會使用或未來維護系統的成本

Façade(3/15): 模式說明(1/1)

如果不使用**Façade**模式，用戶端通常需要和子系統內部的多個模組交互，也就是說用戶端會有很多的朋友，用戶端和這些模組之間都有依賴關係，任意一個模組的變動都可能會引起用戶端的變動。使用**Façade**過後，用戶端只需要和**Façade**交互，也就是說用戶端只有**Façade**這一個朋友，用戶端就不需要去關心子系統內部模組的變動情況了，用戶端只是和這個**Façade**有依賴關係。

這樣一來，用戶端不但簡單，而且這個系統會更有彈性。當系統內部多個模組發生變化的時候，這個變化可以被這個**Façade**吸收和消化，並不需要影響到用戶端，換句話說就是：可以在不影響用戶端的情況下，實現系統內部的維護和擴展。



Façade(4/15): 範例1(1/2)

假設我們有四個模組，負責進行加減乘除並輸出結果，我們會怎麼做呢？

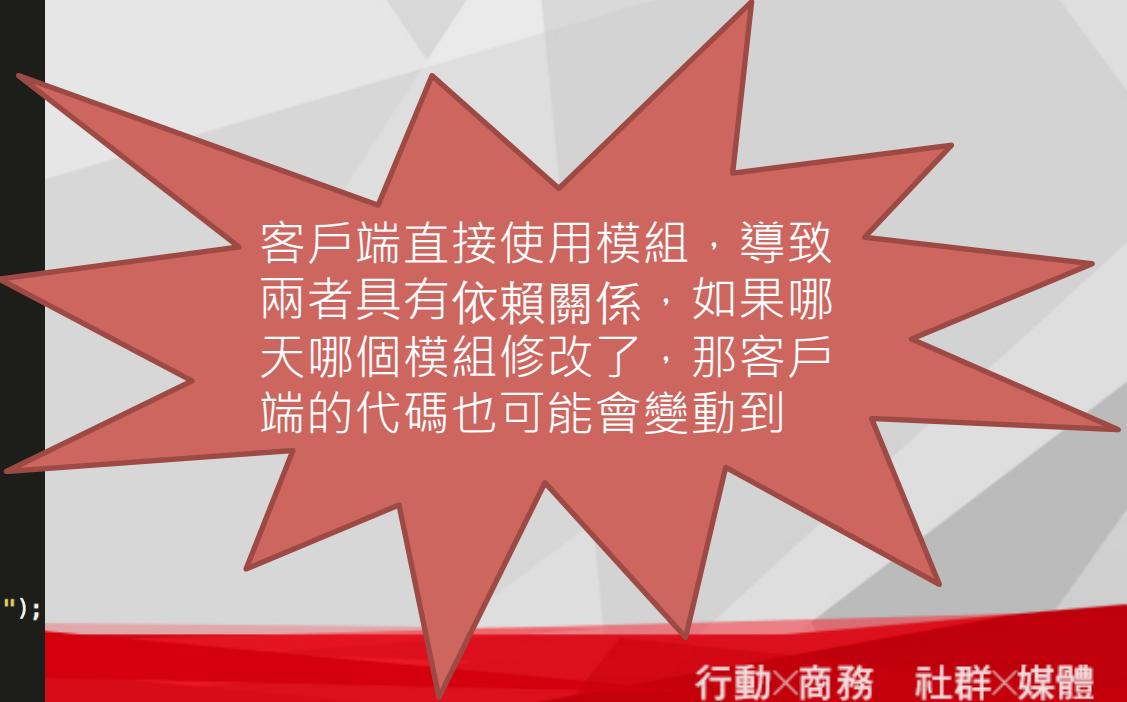
```
// 加法模組
class Adder
{
    public function add($a, $b) {
        return $a + $b;
    }
}

// 減法模組
class Subtractor
{
    public function subtract($a, $b) {
        return $a - $b;
    }
}

// 乘法模組
class Multiplier
{
    public function multiply($a, $b) {
        return $a * $b;
    }
}

// 除法模組
class Divider
{
    public function divide($a, $b) {
        if ($b == 0) {
            throw new Exception("Division by zero.");
        }
        return $a / $b;
    }
}
```

```
$adder = new Adder();
echo "254 + 113 = " . $adder->add(254, 113) . "<br/>";
$divider = new Divider();
echo "256 / 8 = " . $divider->divide(256, 8) . "<br/>";
```



客戶端直接使用模組，導致兩者具有依賴關係，如果哪天哪個模組修改了，那客戶端的代碼也可能會變動到

Façade(5/15): 範例1(2/2)

```
// Façade
class CalculatorFacade
{
    private $_addr;
    private $_subtractor;
    private $_multiplier;
    private $_divider;

    public function __construct() {
        $this->_adder = new Adder();
        $this->_subtractor = new Subtractor();
        $this->_multiplier = new Multiplier();
        $this->_divider = new Divider();
    }

    public function calculate($expression) {
        list($a, $operator, $b) = explode(" ", $expression);

        // eliminating switch constructs is not in the intent of this pattern
        switch ($operator) {
            case '+':
                return $this->_adder->add($a, $b);
            break;
            case '-':
                return $this->_subtractor->subtract($a, $b);
            break;
            case '*':
                return $this->_multiplier->multiply($a, $b);
            break;
            case '/':
                return $this->_divider->divide($a, $b);
            break;
        }
    }
}
```

```
$calculatorFacade = new CalculatorFacade();
echo "254 + 113 = " . $calculatorFacade->calculate("254 + 113") . "<br/>";
echo "256 / 8 = " . $calculatorFacade->calculate("256 / 8") . "<br/>";
```

用戶端只需要和Façade交互，不需要去關心子系統內部模組的變動

Façade(6/15): 範例2(1/10)

再來一個極端一點的例子，我們現在有個家庭劇院系統，以下為它們的關係



擴大機



Façade(7/15):範例2(2/10)

以下為我們需要的動作：

看電影：

- 1.打開戲院燈光組
- 2.將亮度調低10
- 3.將螢幕降下來
- 4.開啟投影機
- 5.將投影機設為寬螢幕模式
- 6.擴大機打開
- 7.連接擴大機與DVD擴大機
- 8.擴大機設為環繞音效
- 9.擴大機音量設為10
- 10.DVD撥放器打開
- 11.DVD撥放器開始撥放

Façade(8/15): 範例2(3/10)

```
// 收音機與聲音控制器
class Tuner
{
    private $amplifier;
    private $description;
    private $frequency;

    public function __construct(Amplifier $amplifier, $description) {
        $this->description = $description;
    }

    public function on() {
        echo($this->description . " on");
        echo("<br />");
    }

    public function off() {
        echo($this->description . " off");
        echo("<br />"); }
    }

    public function setFrequency($frequency) {
        echo($this->description . " setting frequency to " . $frequency);
        echo("<br />"); $this->frequency = $frequency;
    }

    public function setAm() {
        echo($this->description . " setting AM mode");
        echo("<br />"); }

    public function setFm() {
        echo($this->description . " setting FM mode");
        echo("<br />"); }
}
```

```
// 戲院燈光組
class TheaterLights
{
    private $description;

    public function __construct($description) {
        $this->description = $description;
    }

    public function on() {
        echo($this->description . " on");
        echo("<br />"); }
    }

    public function off() {
        echo($this->description . " off");
        echo("<br />"); }

    public function dim($level) {
        echo($this->description . " dimming to " . $level . "%");
        echo("<br />"); }
}
```

Façade(9/15):範例2(4/10)

```
class CDPlayer
{
    private $amplifier;
    private $description;
    private $currentTrack;
    private $title;

    public function __construct(Amplifier $amplifier, $description) {
        $this->amplifier = $amplifier;
        $this->description = $description;
    }

    public function on() {
        echo($this->description . " on");
        echo("<br />");
    }

    public function off() {
        echo($this->description . " off");
        echo("<br />");
    }

    public function eject() {
        $this->title = null;
        echo($this->description . " eject");
        echo("<br />");
    }
}
```

```
public function eject() {
    $this->title = null;
    echo($this->description . " eject");
    echo("<br />");
}

public function play($titleOrTrack) {
    if (is_string($titleOrTrack)) {
        $this->title = $titleOrTrack;
        $this->currentTrack = 0;
        echo($this->description . " playing " . $this->title);
        echo("<br />");
    } else {
        if ($titleOrTrack == null) {
            echo($this->description . " can't play track " . $this->currentTrack . ", no cd inserted");
            echo("<br />");
        } else {
            $this->currentTrack = $titleOrTrack;
            echo($this->description . " playing track " . $this->currentTrack);
            echo("<br />");
        }
    }
}

public function stop() {
    $this->currentTrack = 0;
    echo($this->description . " stopped");
    echo("<br />");
}

public function pause(){
    echo($this->description . " paused " . $this->title);
    echo("<br />");
}

public function __toString() {
    return $this->description;
}
}
```

Façade(10/15): 範例2(5/10)

```
// DVD播放機
class DVDPlayer {

    private $amplifier;
    private $description;
    private $currentTrack;
    private $movie;

    public function __construct($description, Amplifier $amplifier) {
        $this->amplifier = $amplifier;
        $this->description = $description;
    }

    public function on() {
        echo($this->description . " on");
        echo("<br />");
    }

    public function off() {
        echo($this->description . " off");
        echo("<br />");
    }

    public function eject() {
        $this->movie = null;
        echo($this->description . " eject");
        echo("<br />");
    }
}
```

```
public function play($movieOrTrack) {
    if (is_string($movieOrTrack)) {
        $this->movie = $movieOrTrack;
        $this->currentTrack = 0;
        echo($this->description . " playing " . $this->movie);
    } else {
        if ($this->movie == null) {
            echo($this->description . " can't play track " . $movieOrTrack . " no dvd inserted");
        } else {
            $this->currentTrack = $movieOrTrack;
            echo($this->description . " playing track " . $this->currentTrack . " of " . $this->movie);
        }
    }
}

public function stop() {
    $this->currentTrack = 0;
    echo($this->description . " stopped");
    echo("<br />");
}

public function pause() {
    echo($this->description . " paused " . $this->movie);
    echo("<br />");
}

public function setTwoChannelAudio() {
    echo($this->description . " set two channel audio");
    echo("<br />");
}

public function setSurroundAudio() {
    echo($this->description . " set surround audio");
    echo("<br />");
}

public function __toString() {
    return $this->description;
}
}
```

Façade(11/15): 範例2(6/10)

```
// 螢幕
class Screen
{
    private $description;

    public function __construct($description) {
        $this->description = $description;
    }

    public function up() {
        echo($this->description . " going up");
        echo("<br />");
    }

    public function down() {
        echo($this->description . " going down");
        echo("<br />");
    }
}
```

```
// 投影機
class Projector
{
    private $description;
    private $dvdPlayer;

    public function __construct($description, DVDPlayer $dvdPlayer) {
        $this->description = $description;
        $this->dvdPlayer = $dvdPlayer;
    }

    public function on() {
        echo($this->description . " on");
        echo("<br />");
    }

    public function off() {
        echo($this->description . " off");
        echo("<br />");
    }

    public function wideScreenMode() {
        echo($this->description . " in widescreen mode (16x9 aspect ratio)");
        echo("<br />");
    }

    public function tvMode() {
        echo($this->description . " in tv mode (4x3 aspect ratio)");
        echo("<br />");
    }
}
```

Façade(12/15): 範例2(7/10)

```
// 音響擴大機
class Amplifier {
    private $tuner;
    private $dvd;
    private $cd;
    private $description;

    public function __construct($description) {
        $this->description = $description;
    }

    public function on() {
        echo($this->description . " on");
        echo("<br />");
    }

    public function off() {
        echo($this->description . " off");
        echo("<br />");
    }

    public function setCD(CDPlayer $cd) {
        echo($this->description . " setting CD player to " . $cd);
        echo("<br />");
        $this->cd = $cd;
    }

    public function setDVD(DVDPlayer $dvd) {
        echo($this->description . " setting DVD player to " . $dvd);
        echo("<br />");
        $this->dvd = $dvd;
    }

    public function setStereoSound() {
        echo($this->description . " stereo mode on");
        echo("<br />");
    }

    public function setSurroundSound() {
        echo($this->description . " surround sound on (5 speakers, 1 subwoofer)");
        echo("<br />");
    }
}
```

```
public function setTuner(Tuner $tuner) {
    echo($this->description . " setting tuner to " . $this->dvd);
    echo("<br />");
    $this->tuner = $tuner;
}

public function setVolume($volume) {
    echo($this->description . " setting volume to " . $volume);
    echo("<br />");
}
```



Façade(13/15): 範例2(8/10)

如果客戶端直接使用這些模組，情況會是...

```
$amplifier = new Amplifier("Top-O-Line Amplifier");
$tuner = new Tuner($amplifier, "Top-O-Line AM/FM Tuner");
$dvd = new DVDPlayer("Top-O-Line DVD Player", $amplifier);
$cd = new CDPlayer($amplifier, "Top-O-Line CD Player");
$projector = new Projector("Top-O-Line Projector", $dvd);
$lights = new TheaterLights("Theater Ceiling Lights");
$screen = new Screen("Theater Screen");
$movie = 'TopGun';

echo "Get ready to watch a movie...<br/>";
$lights->on();
$lights->dim(10);
$screen->down();
$projector->on();
$projector->wideScreenMode();
$amplifier->on();
$amplifier->setDVD($dvd);
$amplifier->setSurroundSound();
$amplifier->setVolume(5);
$dvd->on();
$dvd->play($movie);
```



Get ready to watch a movie...
Theater Ceiling Lights on
Theater Ceiling Lights dimming to 10%
Theater Screen going down
Top-O-Line Projector on
Top-O-Line Projector in widescreen mode (16x9 aspect ratio)
Top-O-Line Amplifier on
Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player
Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)
Top-O-Line Amplifier setting volume to 5
Top-O-Line DVD Player on
Top-O-Line DVD Player playing TopGun

除了用戶端與模組具有依賴關係的問題外，用戶要正確使用個模組與了解之間的複雜關係也比較困難。

Façade(14/15): 範例2(9/10)

```
class HomeTheaterFacade {
    private $amplifier;
    private $tuner;
    private $dvd;
    private $cd;
    private $projector;
    private $lights;
    private $screen;

    public function __construct(Amplifier $amplifier, Tuner $tuner, DVDPlayer $dvd, CDPlayer $cd, Projector $projector, Screen $screen, TheaterLights $lights) {
        $this->amplifier = $amplifier;
        $this->tuner = $tuner;
        $this->dvd = $dvd;
        $this->cd = $cd;
        $this->projector = $projector;
        $this->screen = $screen;
        $this->lights = $lights;
    }

    public function watchMovie($movie) {
        echo "Get ready to watch a movie...<br/>";
        $this->lights->on();
        $this->lights->dim(10);
        $this->screen->down();
        $this->projector->on();
        $this->projector->wideScreenMode();
        $this->amplifier->on();
        $this->amplifier->setDVD($this->dvd);
        $this->amplifier->setSurroundSound();
        $this->amplifier->setVolume(5);
        $this->dvd->on();
        $this->dvd->play($movie);
    }
}
```

讓我們建立一個Façade....

Façade(15/15): 範例(10/10)

現在用戶端就不需要去關心子系統，只要使用Façade就可以簡單的達到目的

```
$amplifier = new Amplifier("Top-O-Line Amplifier");
$tuner = new Tuner($amplifier, "Top-O-Line AM/FM Tuner");
$dvd = new DVDPlayer("Top-O-Line DVD Player", $amplifier);
$cd = new CDPlayer($amplifier, "Top-O-Line CD Player");
$projector = new Projector("Top-O-Line Projector", $dvd);
$lights = new TheaterLights("Theater Ceiling Lights");
$screen = new Screen("Theater Screen");
$movie = 'TopGun';

$homeTheater = new HomeTheaterFacade($amplifier, $tuner, $dvd, $cd, $projector, $screen, $lights);
$homeTheater->watchMovie($movie);
```

Get ready to watch a movie...

Theater Ceiling Lights on

Theater Ceiling Lights dimming to 10%

Theater Screen going down

Top-O-Line Projector on

Top-O-Line Projector in widescreen mode (16x9 aspect ratio)

Top-O-Line Amplifier on

Top-O-Line Amplifier setting DVD player to Top-O-Line DVD Player

Top-O-Line Amplifier surround sound on (5 speakers, 1 subwoofer)

Top-O-Line Amplifier setting volume to 5

Top-O-Line DVD Player on

Top-O-Line DVD Player playing TopGun



Flyweight(享元模式)

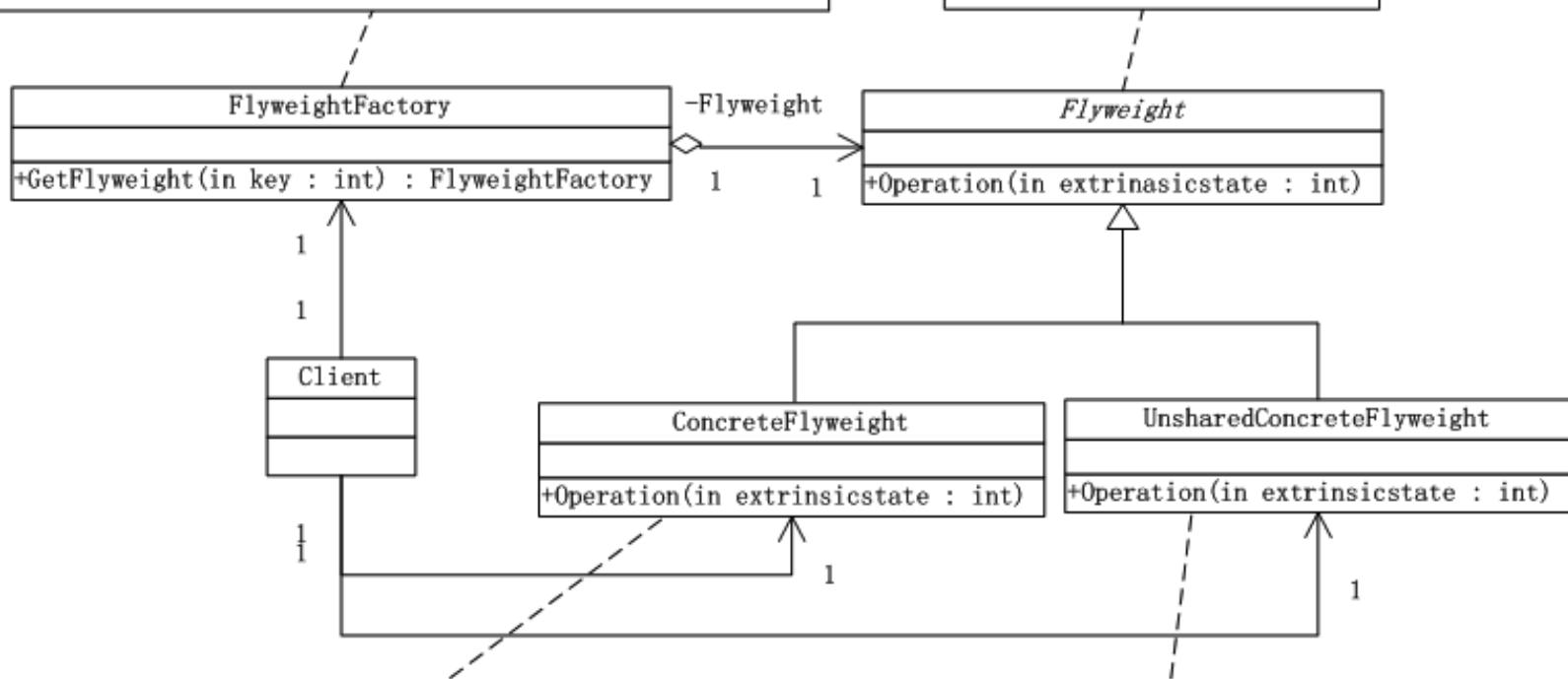
Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

Flyweight(1/17): UML(1/1)

一个享元工厂，用来创建并管理Flyweight对象。它主要是用来确保合理的共享Flyweight，当用户请求一个Flyweight时，FlyweightFactory对象提供一个已建立的实例或者创建一个（如果不存在的话）。

所有具体享元类的超类或接口，通过这个接口，Flyweight可以接受并作用于外部状态。



继承Flyweight超类或者实现Flyweight接口，并为内部状态增加存储空间。

複合享元(UnsharableFlyweight)角色：複合享元角色所代表的物件是不可以共用的，但是一個複合享元物件可以分解成為多個本身是單純享元物件的組合。

Flyweight(2/17):複合享元模式說明

- 複合享元模式物件是由一些單純享元使用合成模式加以複合而成
- 複合享元角色所代表的物件是不可以共用的，但是一個複合享元物件可以分解成多個本身是單純享元物件的組合
- 每個單純享元都具有相同的外部狀態，而這些單純享元的內部狀態是不一樣的時候，可以使用複合享元模式
- 建議盡量使用單純享元模式，因為複合享元模式邏輯複雜很多又要用到額外的記憶體來保存物件與內部狀態。

Flyweight(3/17): 模式分析(1/2)

優點：

- Flyweight大幅度的降低了記憶體中物件的數量

缺點：

- Flyweight會使系統更加複雜，因為為了使物件可以共用，需要將一些狀態外部化，使得程式的邏輯複雜化
- Flyweight將共用物件的狀態外部化，讀取外部狀態會使得程式讀取時間變長，等於是用時間換取空間

Flyweight(4/17): 模式分析(2/2)

使用時機：

- 程式裡面存在著大量的細粒度物件
- 這些細粒度物件導致大量的記憶體開銷
- 這些細粒度物件的大部分狀態可以外部化
- 這些物件可以按照內部狀態分成很多組，當把外部狀態從物件中剔除時，每個組都可以僅用一個物件代替。(例如英文字母A到Z, A~Z符號為內部狀態，外部狀態如大小、字形、位置....等)
- 系統不依賴於這些物件的身份，換言之，這些物件可以是不可分辨的。
- 最後，使用Flyweight需要維護一個記錄系統已有的所有共用單元的表，而這需要耗費資源。因此，應當在有足夠多的Flyweight實例可供共用時才值得使用Flyweight模式。

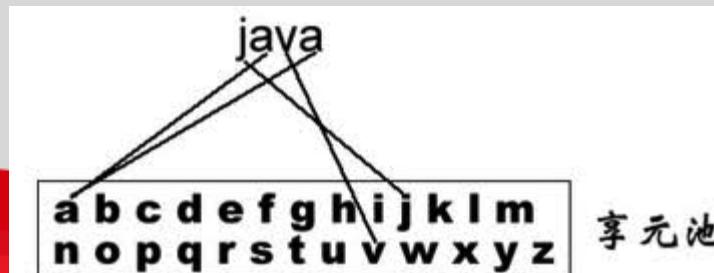


Flyweight(5/17)：模式說明(1/1)

內部狀態(Intrinsic State)：存儲在享元物件內部並且不會隨環境改變而改變。因此內部狀態可以共用。(例如在繪製一個英文字串時，重覆的字元部份我們就可以將之設定為內部狀態，像是 "ABC is BAC"，其中A、B、C的字元資訊部份不必直接儲存於字元物件中，它是屬於可以共享的部份。)

外部狀態(Extrinsic State)：是隨環境改變而改變的、不可以共用的狀態。外部狀態與內部狀態是相互獨立的，享元物件的外部狀態必須由用戶端保存，並在享元物件被創建之後，在需要使用的時候再傳入到享元物件內部。(例如繪製字元時的字型資訊、位置資訊等等，我們繪製一個字元時，先從flyweight pool中找出共享的享元，然後從外部狀態中查找對應的繪製資訊-字型、大小、位置等。)

在享元模式中通常會出現工廠模式，需要創建一個享元工廠來負責維護一個享元池(Flyweight Pool)用於存儲具有相同內部狀態的享元物件。



Flyweight(6/17):範例1:單純享元模式(1/2)

```
// Flyweight
abstract class Flyweight
{
    abstract public function operation($extrinsicState);
}

// ConcreteFlyweight
class ConcreteFlyweight extends Flyweight
{
    private $intrinsicState = null;
    public function __construct($intrinsicState) {
        $this->intrinsicState = $intrinsicState;
    }

    public function operation($extrinsicState) {
        echo 'ConcreteFlyweight operation, Intrinsic State = ' . $this->intrinsicState
        . ' Extrinsic State = ' . $extrinsicState . '<br />';
    }
}

class FlyweightFactory
{
    private $flyweights;

    public function __construct() {
        $this->flyweights = array();
    }

    public function getFlyweight($state) {
        if (isset($this->flyweights[$state])) {
            return $this->flyweights[$state];
        } else {
            return $this->flyweights[$state] = new ConcreteFlyweight($state);
        }
    }
}
```

Flyweight(7/17):範例1:單純享元模式(2/2)

```
- include_once '../class/pattern/flyweight.php';

$flyweightFactory = new FlyweightFactory();
$flyweight = $flyweightFactory->getFlyweight("state A");
$flyweight->operation("other state A");

$flyweight = $flyweightFactory->getFlyweight("state B");
$flyweight->operation("other state B");
```



ConcreteFlyweight operation, Intrinsic State = state A Extrinsic State = other state A
ConcreteFlyweight operation, Intrinsic State = state B Extrinsic State = other state B

Flyweight(8/17):範例2:複合享元模式(1/2)

```
// ConcreteFlyweight
class ConcreteFlyweight extends Flyweight
{
    private $intrinsicState = null;
    public function __construct($intrinsicState) {
        $this->intrinsicState = $intrinsicState;
    }

    public function operation($extrinsicState) {
        echo 'ConcreteFlyweight operation, Intrinsic State = ' . $this->intrinsicState
        . ' Extrinsic State = ' . $extrinsicState . '<br />';
    }
}
```

```
// UnsharedConcreteFlyweight
class UnsharedConcreteFlyweight extends Flyweight
{
    private $flyweights;
    public function __construct() {
        $this->flyweights = array();
    }

    public function operation($state) {
        foreach ($this->flyweights as $flyweight) {
            $flyweight->operation($state);
        }
    }

    public function add($state, Flyweight $flyweight) {
        $this->flyweights[$state] = $flyweight;
    }
}
```

Flyweight(9/17):範例2:複合享元模式(1/2)

```
class FlyweightFactory
{
    private $flyweights;
    public function __construct() {
        $this->flyweights = array();
    }

    public function getFlyweight($state) {
        if (is_array($state)) { // Unshared
            $uFlyweight = new UnsharedConcreteFlyweight();

            foreach ($state as $row) {
                $uFlyweight->add($row, $this->getFlyweight($row));
            }
        }

        return $uFlyweight;
    } else if (is_string($state)) {

        if (isset($this->_flyweights[$state])) {
            return $this->_flyweights[$state];
        } else {
            return $this->_flyweights[$state] = new ConcreteFlyweight($state);
        }
    } else {
        return null;
    }
}
```

Flyweight(10/17):範例2:複合享元模式(1/2)

```
// unshared  
$uflyweight = $flyweightFactory->getFlyweight(array('state A', 'state B'));  
$uflyweight->operation('other state A');
```



ConcreteFlyweight operation, Intrinsic State = state A Extrinsic State = other state A
ConcreteFlyweight operation, Intrinsic State = state B Extrinsic State = other state A

Flyweight(11/17):範例3:文字編輯器(1/5)

現在我們要設計一個文字編輯器，可以打26個大寫字母，並且有大小、顏色、位置X、位置Y四個屬性，如果我們現在直接用物件來產生每一個字母...

```
// 文字編輯器反模式
class Character
{
    private $char = null;
    private $size = 12;
    private $color = "black";
    private $posX = 0;
    private $posY = 0;

    public function __construct($char, $size, $color, $posX, $posY) {
        $this->char = $char;
        $this->size = $size;
        $this->color = $color;
        $this->posX = $posX;
        $this->posY = $posY;
    }

    public function showChar() {
        echo $this->char . " size:" . $this->size . " color:" . $this->color
            . " (" . $this->posX . "," . $this->posY . ") <br>";
    }
}
```

Flyweight(12/17):範例3:文字編輯器(2/5)

```
include_once '../../class/pattern/flyweight.php';

$charArr = array('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z');
$colorArr = array("black", "red", "orange", "yellow", "pink", "brown", "blue", "green");
$sizeArr = array(12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22);

$flyweightFactory = new FlyweightFactory();

// 取得記憶體使用量
$MaxMem = memory_get_peak_usage();
$MEMBefore =memory_get_usage();

echo "產生物件前記憶體目前用掉{$MEMBefore} , 最多用掉{$MaxMem} ";

echo "<br>";
}

for ($i = 0; $i < 100000; $i++) {

    $char = $charArr[rand(0, 25)];
    $size = $sizeArr[rand(0, 10)];
    $color = $colorArr[rand(0, 7)];
    $posX = rand(0, 1000);
    $posY = rand(0, 1000);

    $character = new Character($char, $size, $color, $posX, $posY);
    $chars[] = $character;
}

// 取得記憶體使用量
$MaxMem = memory_get_peak_usage();
$MEM = memory_get_usage() - $MEMBefore;

echo "產生物件後記憶體目前用掉{$MEM} , 最多用掉{$MaxMem} ";
```

Flyweight(13/17):範例3:文字編輯器(3/5)

```
// Flyweight
interface Flyweight
{
    // 設置內外狀態關係連結的方法
    public function showChar($size, $color, $posX, $posY);
}

// ConcreteFlyweight
class Character implements Flyweight
{
    private $char = null; // intrinsicState
    public function __construct($char) {
        $this->char = $char;
    }

    // Operation
    public function showChar($size, $color, $posX, $posY) {
        echo $this->char . " size:" . $size . " color:" . $color
            . " (" . $posX . "," . $posY . ") <br>";
    }
}

class FlyweightFactory
{
    private $flyweights;

    public function __construct() {
        $this->flyweights = array();
    }

    public function getFlyweight($char) {
        if (isset($this->flyweights[$char])) {
            return $this->flyweights[$char];
        } else {
            return $this->flyweights[$char] = new Character($char);
        }
    }
}
```

改用Flyweight....

Flyweight(14/17):範例3:文字編輯器(4/5)

```
include_once '../class/pattern/flyweight.php';

$charArr = array('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z');
$colorArr = array("black", "red", "orange", "yellow", "pink", "brown", "blue", "green");
$sizeArr = array(12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22);

$flyweightFactory = new FlyweightFactory();

// 取得記憶體使用量
$MaxMem = memory_get_peak_usage();
$MEMBefore =memory_get_usage();

echo "產生物件前記憶體目前用掉{$MEMBefore} , 最多用掉{$MaxMem} ";
echo "<br>";

$charsExtrinsicStatus = array();

for ($i = 0; $i < 100000; $i++) {

    $char = $charArr[rand(0, 25)];
    $size = $sizeArr[rand(0, 10)];
    $color = $colorArr[rand(0, 7)];
    $ posX = rand(0, 1000);
    $ posY = rand(0, 1000);

    $character = $flyweightFactory->getFlyweight($char);
    // $character->showChar($size, $color, $posX, $posY);

    $charsExtrinsicStatus[] = array($char, $size, $color, $posX, $posY);
}

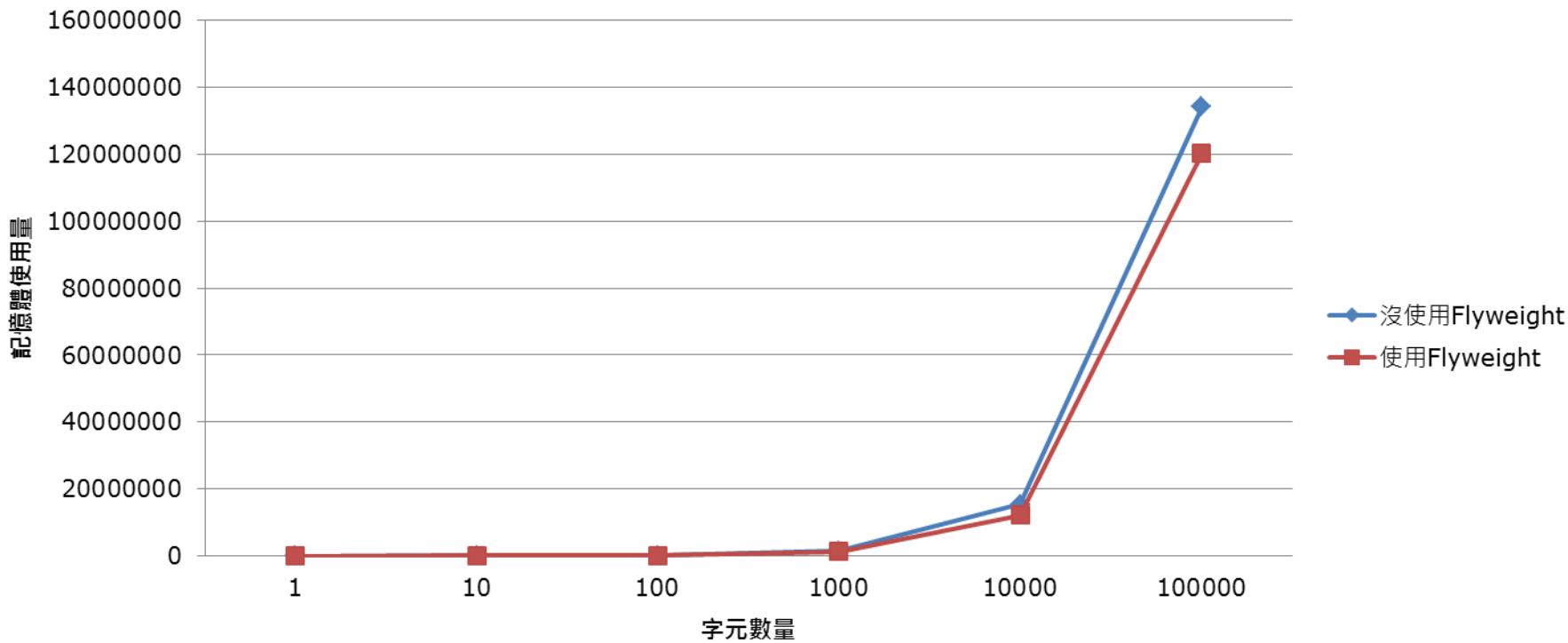
// 取得記憶體使用量
$MaxMem = memory_get_peak_usage();
$MEM = memory_get_usage() - $MEMBefore;

echo "產生物件後記憶體目前用掉{$MEM} , 最多用掉{$MaxMem} ";
```



Flyweight(15/17):範例3:文字編輯器(5/5)

記憶體使用比較圖

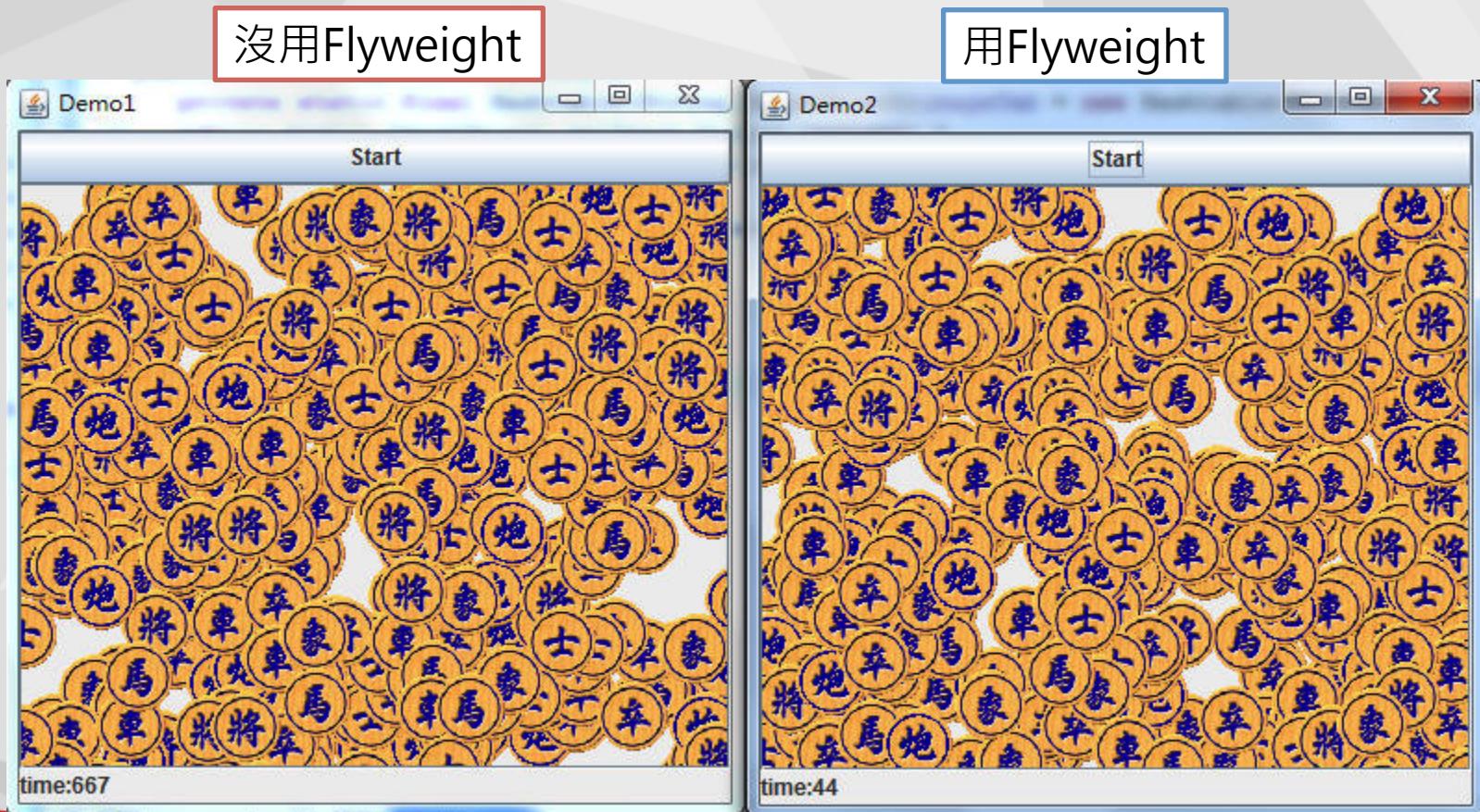


Flyweight(16/17):範例測試感想

- 內部狀態越多，外部狀態越少，使用Flyweight節省記憶體的效果會更好
- 由於Flyweight的部分狀態抽取到外部管理，所以怎麼管理、儲存與索引這些外部狀態，會變成另外一件很重要的事情。
- PHP用Flyweight的效果不明顯，因為每個執行敘處理完後，記憶體都會被釋放...

Flyweight(17/17):Java象棋範例

<http://www.javahome.idv.tw/technology/FlyweightDemo2.asp?pageA=FlyweightDemo2>



Hiiir

Proxy(代理人模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

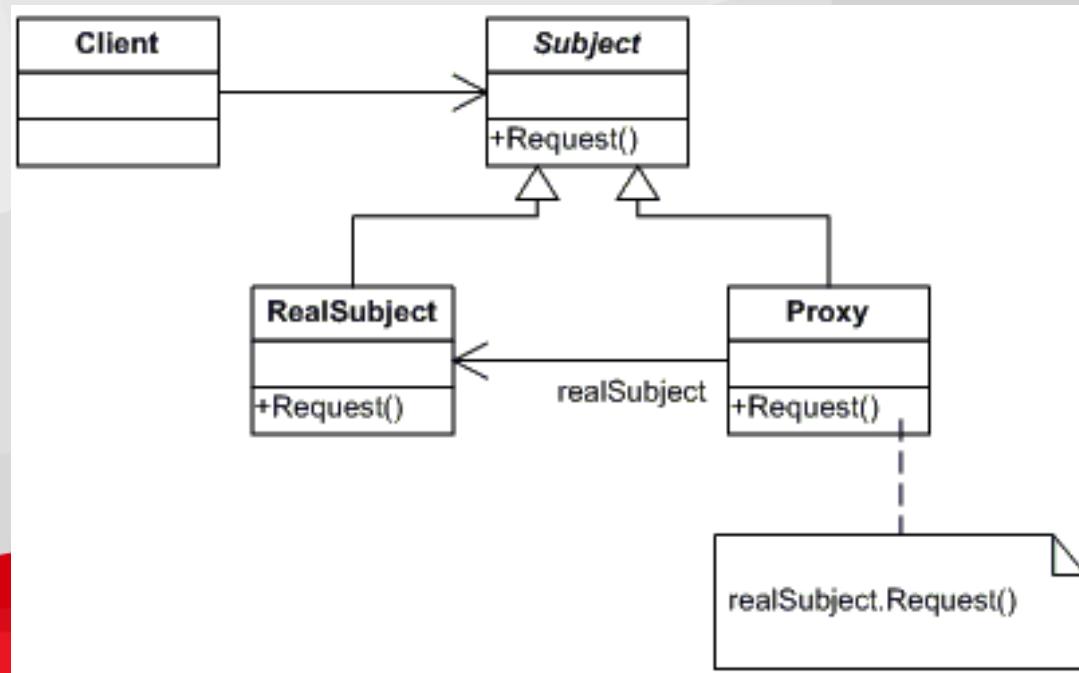
Proxy(1/6) : UML(1/1)

Subject : 聲明了 RealSubject 和 Proxy 的共同介面，這樣一來在任何使用 RealSubject 的地方都可以使用 Proxy

Proxy : Proxy 內部含有對 RealSubject 的引用，從而可以在任何時候操作 RealSubject 物件；Proxy 提供一個與 RealSubject 相同的介面，以便可以在任何時候都可以替代 RealSubject；控制 RealSubject 的應用，負責在需要的時候創建 RealSubject 物件（和刪除 RealSubject 物件）；Proxy 通常在將用戶端調用傳遞給 RealSubject 前或之後，都要執行某個操作，而不是單純的將調用傳遞給 RealSubject。

RealSubject : 定義了代理角色所代表的真實物件。

Client : 客戶端



Proxy(2/6): 模式分析(1/2)

優點：

- Proxy能夠協調調用者和被調用者，在一定程度上降低了系統的耦合度。
- 遠端代理使得用戶端可以訪問在遠端機器上的物件，遠端機器可能具有更好的計算性能與處理速度，可以快速回應並處理用戶端請求。
- 虛擬代理通過使用一個小物件來代表一個大物件，可以減少系統資源的消耗，對系統進行優化並提高運行速度。
- 保護代理可以控制對真實物件的使用權限

缺點：

- 由於在用戶端和真實主題之間增加了代理物件，因此有些類型的代理模式可能會造成請求的處理速度變慢。
- 實現代理模式需要額外的工作，有些代理模式的實現非常複雜。

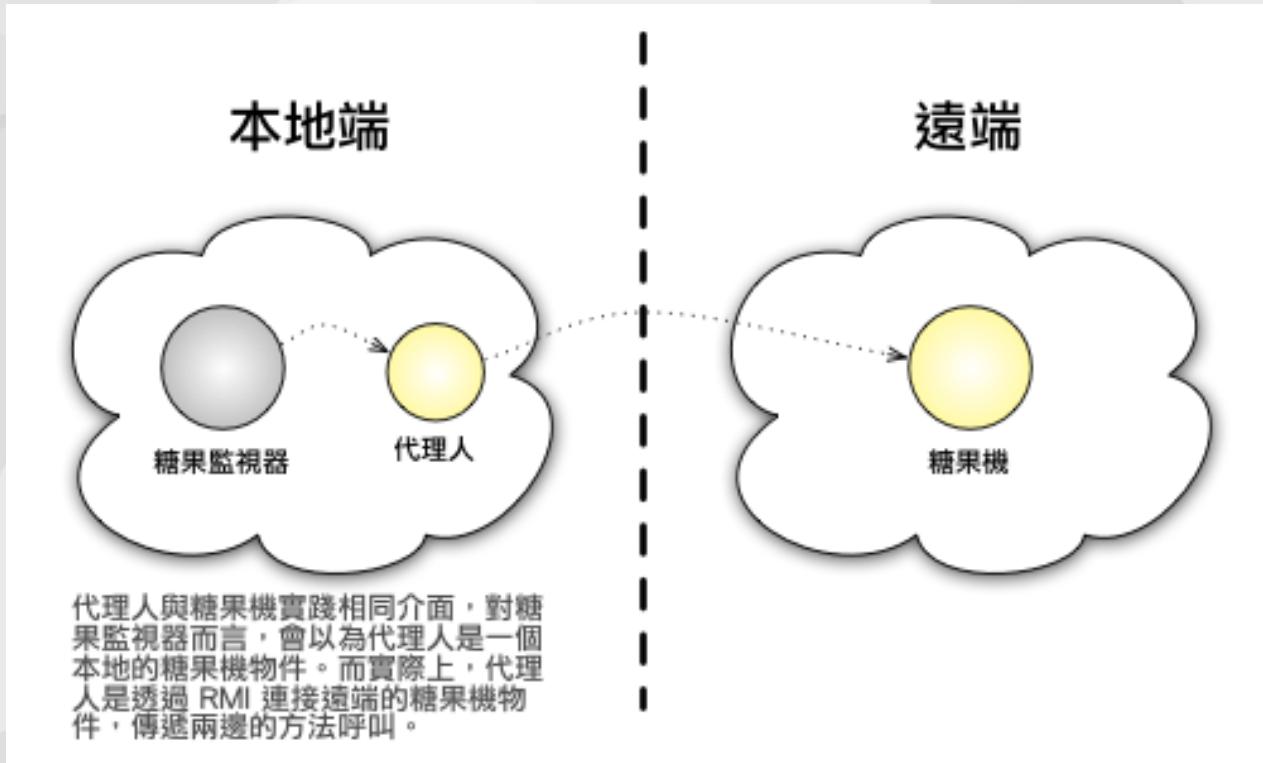
Proxy(3/6): 模式分析(2/2)

使用時機：

- **遠程 (Remote) 代理**：為一個位於不同的位址空間的物件提供一個局域代表物件。這個不同的位址空間可以是在本機器中，也可是在另一台機器中。遠端代理又叫做大使 (Ambassador) 。
- **虛擬 (Virtual) 代理**：根據需要創建一個資源消耗較大的物件，使得此物件只在需要時才會被真正創建。
- **Copy-on-Write代理**：虛擬代理的一種。把複製 (克隆) 拖延到只有在用戶端需要時，才真正採取行動。
- **保護 (Protect or Access) 代理**：控制對一個物件的訪問，如果需要，可以給不同的用戶提供不同級別的使用權限。
- **Cache代理**：為某一個目標操作的結果提供臨時的存儲空間，以便多個用戶端可以共用這些結果。
- **防火牆 (Firewall) 代理**：保護目標，不讓惡意使用者接近。
- **同步化 (Synchronization) 代理**：使幾個使用者能夠同時使用一個物件而沒有衝突。
- **智能引用 (Smart Reference) 代理**：當一個物件被引用時，提供一些額外的操作，比如將對此物件調用的次數記錄下來等。



Proxy(4/6): 模式說明(1/1)



Client物件運作起來就像是呼叫遠端，但其實是在呼叫本地端的代理人物件，再由代理人去跟遠端溝通

Proxy(5/6):範例:訊息發布系統 (1/2)

```
// Subject
interface AbstractPermission
{
    public function modifyUserInfo();
    public function viewNote();
    public function publishNote();
    public function modifyNote();
    public function setLevel($level);
}
```

```
// RealSubject
class RealPermission implements AbstractPermission
{
    private $level = 0;

    public function modifyUserInfo() {
        echo "修改用戶信息! <br>";
    }

    public function viewNote() {
        echo "查看內容! <br>";
    }

    public function publishNote() {
        echo "發佈新訊息! <br>";
    }

    public function modifyNote() {
        echo "修改發佈訊息內容! <br>";
    }

    public function setLevel($level) {
        $this->level = $level;
    }

    public function getLevel() {
        return $this->level;
    }
}
```

Hiiir

Proxy(6/6):範例:訊息發布系統 (2/2)

```
// Proxy
class PermissionProxy implements AbstractPermission
{
    private $realPermission = null;

    public function __construct() {
        $this->realPermission = new RealPermission();
    }

    public function modifyUserInfo() {

        if ($this->realPermission->getLevel() == 0) {
            echo "對不起，你沒有該權限！<br>";
        } else {
            echo "開始修改用戶權限！<br>";
        }
    }

    public function viewNote() {
        $this->realPermission->viewNote();
    }

    public function publishNote() {
        if ($this->realPermission->getLevel() == 0) {
            echo "對不起，你沒有權限發佈新訊息！<br>";
        } else {
            $this->realPermission->publishNote();
        }
    }

    public function modifyNote() {
        if ($this->realPermission->getLevel() == 0) {
            echo "對不起，你沒有權限修改訊息！<br>";
        } else {
            $this->realPermission->modifyNote();
        }
    }

    public function setLevel($level) {
        $this->realPermission->setLevel($level);
    }
}
```

```
include_once '../../class/pattern/proxy.php';

$permissionProxy = new PermissionProxy();

$permissionProxy->modifyUserInfo();
$permissionProxy->viewNote();
$permissionProxy->publishNote();
$permissionProxy->modifyNote();

echo "<br>";
$permissionProxy->setLevel(1);
$permissionProxy->modifyUserInfo();
$permissionProxy->viewNote();
$permissionProxy->publishNote();
$permissionProxy->modifyNote();
```



對不起，你沒有該權限！
查看內容！
對不起，你沒有權限發佈新訊息！
對不起，你沒有權限修改訊息！

開始修改用戶權限！
查看內容！
發佈新訊息！
修改發佈訊息內容！

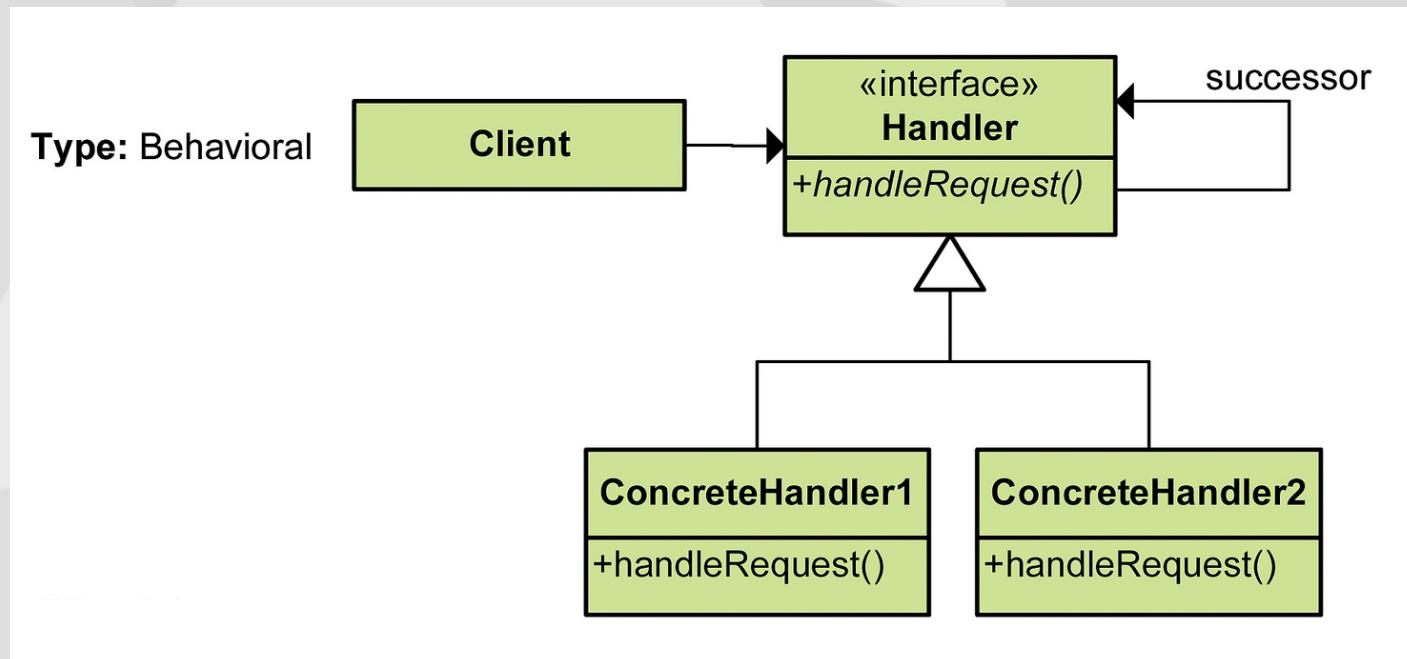
Chain of Responsibility(責任鍊模式)

CoR(1/13):UML(1/1)

Handler：定義出一個處理請求的接口。如果需要，接口可以定義出一個方法以設定和返回對下家的引用。抽象方法handleRequest()規範了子類處理請求的操作。

ConcreteHandler：具體處理者接到請求後，可以選擇將請求處理掉，或者將請求傳給下家。由於具體處理者持有對下家的引用，因此，如果需要，具體處理者可以訪問下家。

Client：客戶端



CoR(2/13): 模式分析(1/1)

優點：

- 責任的分擔。每個類只需要處理自己該處理的工作（不該處理的傳遞給下一個物件完成），明確各類的責任範圍，符合類的最小封裝原則
- 可以根據需要自由組合工作流程。如工作流程發生變化，可以通過重新分配物件鏈便可適應新的工作流程
- 類別與類別之間可以以鬆耦合的形式加以組織

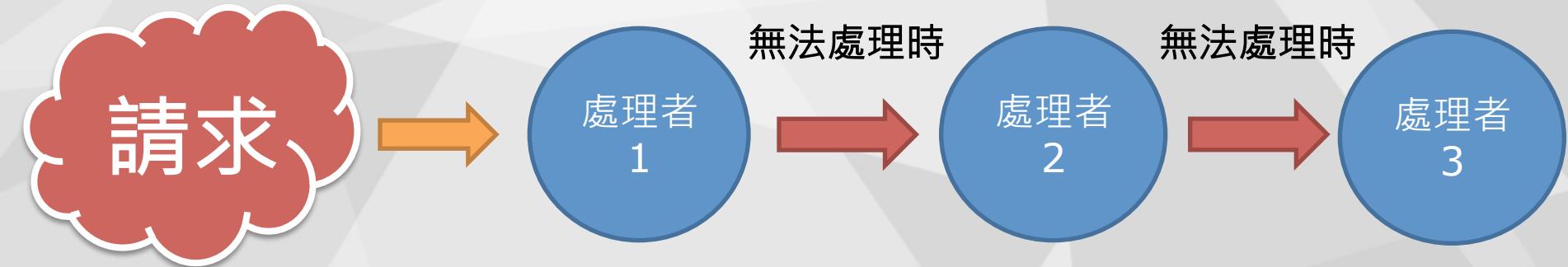
缺點：

- 因為處理時以鏈的形式在物件間傳遞消息，所有的請求都要從鍊的開頭開始執行，有可能會影響處理的速度
- 擴展性差，因為在CoR中，一定要有一個統一的介面Handler

使用時機：

- 為解除請求的發送者和接收者之間耦合，而使多個對象都有機會處理這個請求。將這些對象連成一條鏈，並沿著這條鏈傳遞該請求，直到有一個對象處理它。

CoR(3/13): 模式說明(1/3)



純責任鍊：

一個純的責任鏈模式要求一個具體的處理者物件只能在兩個行為中選擇一個：一個是承擔責任，二是把責任推給下家。不允許出現某一個具體處理者物件在承擔了一部分責任後又把責任向下傳的情況。

不純責任鍊：

一個具體處理者物件在承擔了一部分責任後又把責任向下傳遞。一個請求可以最終不被任何接收端物件所接收。

CoR(4/13): 模式說明(2/3)

```
function funcSwitch($i, $request) {  
    switch($i) {  
        case 1:  
            $hander1.response($request);  
            break;  
  
        case 2:  
            $hander2.response($request);  
            break;  
  
        case 3:  
            $hander3.response($request);  
            break;  
  
        case 4:  
            $hander4.response($request);  
            break;  
  
        case 5:  
            $hander5.response($request);  
            break;  
    }  
}
```

```
function funcIfElse($i, $request) {  
    if($i==1)  
        $hander1.response($request);  
    else if($i==2)  
        $hander2.response($request);  
    else if($i==3)  
        $hander3.response($request);  
    else if($i==4)  
        $hander4.response($request);  
    else if($i==5)  
        $hander5.response($request);  
}
```

if..else 與 switch case也是一種責任鍊的具體實現

那為什麼不直接用if..else跟
switch case就好?

CoR(5/13): 模式說明(3/3)

為什麼用if..else跟switch case不好？

代碼臃腫：

實際應用中的判定條件通常不是這麼簡單地判斷是否為1或者是否為2，也許需要複雜的計算，也許需要查詢資料庫等等，這就會有很多額外的代碼，如果判斷條件再比較多，那麼這個if...else...語句基本上就沒法看了。

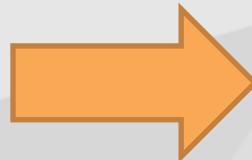
耦合度高：

如果我們想繼續添加處理請求的類別，那麼就要繼續添加else if判定條件；另外，這個條件判定的順序也是寫死的，如果想改變順序，那麼也只能修改這個條件陳述式。

CoR(6/13):純責任鍊範例:Log紀錄器 (1/4)

現在我們要設計一個可以記錄不同Log紀錄的系統，並且能夠彈性的增加或減少能記錄log的類型。

Log資料
(不知類型)



CoR(7/13): 純責任鍊範例: Log紀錄器 (2/4)

```
// Handler
abstract class Logger
{
    protected $handler = null;

    public function setSuccessor(Logger $handler) { // 設定責任要傳遞的下家
        $this->handler = $handler;
    }

    protected function handleRequest($request) {
        if ($this->handler != null) {
            return $this->handler->handleRequest($request); // 如果要傳遞的下家存在，就將request往下傳，如果下家不存在則拋出處理完的request
        } else {
            return "無法處理此類型的資料!";
        }
    }
}
```

CoR(8/13):純責任鍊範例:Log紀錄器 (3/4)

```
// ConcreteHandler 資料庫logger
class dbLogger extends Logger
{
    public function handleRequest($request) {
        if (strpos($request, "db") !== false) {
            return "將db log寫入";
        } else {
            return parent::handleRequest($request);
        }
    }
}

// ConcreteHandler Session Logger
class sessionLogger extends Logger
{
    public function handleRequest($request) {
        if (strpos($request, "session") !== false) {
            return "將session log寫入";
        } else {
            return parent::handleRequest($request);
        }
    }
}

// ConcreteHandler Cache Logger
class cacheLogger extends Logger
{
    public function handleRequest($request) {
        if (strpos($request, "cache") !== false) {
            return "cache log寫入";
        } else {
            return parent::handleRequest($request);
        }
    }
}
```

CoR(9/13): 純責任鍊範例: Log紀錄器 (4/4)

```
$data = "balabalcacheabanlabala hahalalalala";
$dbLogger = new dbLogger();
$sessionLogger = new sessionLogger();
$cacheLogger = new cacheLogger();

$sessionLogger->setSuccessor($cacheLogger);
$dbLogger->setSuccessor($sessionLogger);

$msg = $dbLogger->handleRequest($data); // 順序為 dbLogger -> sessionLogger -> cacheLogger

echo $msg; // cache log寫入
exit;
```

CoR(10/13):不純責任鍊範例:髒話過濾器 (1/4)

現在我們要設計一個髒話過濾系統，可以將輸入的句子中的各種髒話自動過濾掉。此髒話過濾器一開始只能過濾中文，之後要能夠彈性的增加或減少其他語言的髒話過濾功能。



CoR(11/13):不純責任鍊範例:髒話過濾器 (2/4)

```
// Handler
abstract class DirtyWordFilter
{
    protected $keyWord = array();
    protected $handler = null;
    public function setSuccessor(DirtyWordFilter $handler) { // 設定責任要傳遞的下家
        $this->handler = $handler;
    }
    protected function handleRequest($request) {
        if($this->handler != null) { // 如果要傳遞的下家存在，就將request往下傳，如果下家不存在則拋出處理完的request
            return $this->handler->handleRequest($request);
        } else {
            return $request;
        }
    }
}
```

CoR(12/13):不純責任鍊範例:髒話過濾器 (3/4)

```
// ConcreteHandler
class ChineseDirtyWordFilter extends DirtyWordFilter
{
    public function __construct() {
        $this->keyWord = array(
            "豬頭", "笨蛋", "白癡", "三八", "王八蛋"
        );
    }

    public function handleRequest($request) {
        foreach($this->keyWord as $row) {
            $request = str_replace($row, "***", $request);
        }
        return parent::handleRequest($request);
    }
}
```

```
// ConcreteHandler
class EnglishDirtyWordFilter extends DirtyWordFilter
{
    public function __construct() {
        $this->keyWord = array(
            "son of bitch", "idiot", "asshole", "bullshit"
        );
    }

    public function handleRequest($request) {
        foreach($this->keyWord as $row) {
            $request = str_replace($row, "???", $request);
        }
        return parent::handleRequest($request);
    }
}
```

CoR(13/13):不純責任鍊範例:髒話過濾器 (4/4)

```
$chineseDirtyWordHandler = new ChineseDirtyWordFilter(); // 中文髒話過濾
$englishDirtyWordHandler = new EnglishDirtyWordFilter(); // 英文髒話過濾

$msg = "你這個人真三八，是一個idiot，講的話都是bullshit，真像一個笨蛋加白癡，王八蛋!!";

// $msg = $chineseDirtyWordHandler->handleRequest($msg);
// echo $msg; // 你這個人真***，是一個idiot，講的話都是bullshit，真像一個***加***，***!!

// $msg = $englishDirtyWordHandler->handleRequest($msg);
// echo $msg; // 你這個人真三八，是一個???, 講的話都是bull???, 真像一個笨蛋加白癡，王八蛋!!

$chineseDirtyWordHandler->setSuccessor($englishDirtyWordHandler); // 設定承接response的下家

$msg = $chineseDirtyWordHandler->handleRequest($msg);
echo $msg; // 你這個人真***，是一個???, 講的話都是???, 真像一個***加***，***!!
```

Command(命令模式)

Command(1/13): UML(1/1)

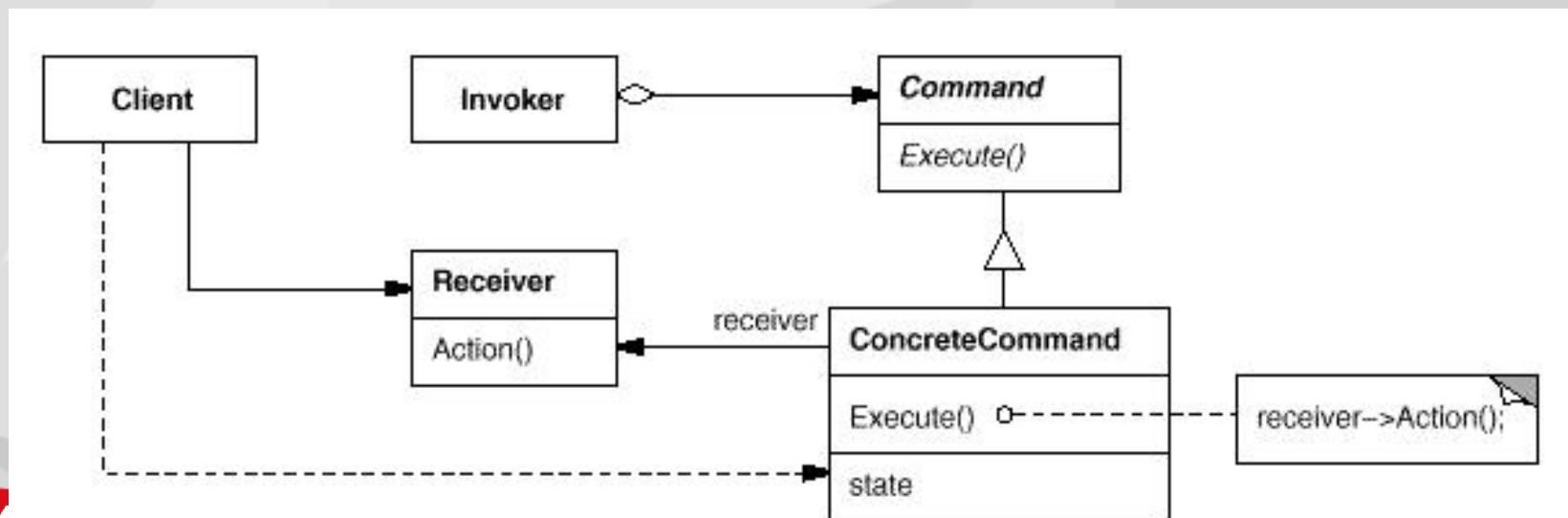
Command : 定義命令的介面，聲明執行的方法。

ConcreteCommand : 命令介面實現物件，是“虛”的實現；通常會持有接收者，並調用接收者的功能來完成命令要執行的操作。

Receiver : 接收者，真正執行命令的物件。任何類都可能成為一個接收者，只要它能夠實現命令要求實現的相應功能。

Invoker : 要求命令物件執行請求，通常會持有命令物件，可以持有很多的命令物件。這個是用戶端真正觸發命令並要求命令執行相應操作的地方，也就是說相當於使用命令物件的入口。

Client : 創建具體的命令物件，並且設置命令物件的接收者。注意這個不是我們常規意義上的用戶端，而是在組裝命令物件和接收者，或許，把這個Client稱為裝配者會更好理解，因為真正使用命令的用戶端是從Invoker來觸發執行。



Command(2/13): 模式分析(1/1)

優點：

- 降低系統的耦合度
- 新的命令可以很容易地加入到系統中
- 可以比較容易地設計一個組合命令

缺點：

- 使用命令模式可能會導致某些系統有過多的具體命令類別。因為針對每一個命令都需要設計一個具體命令類別，因此某些系統可能需要大量具體命令類別，這將影響命令模式的使用

使用時機：

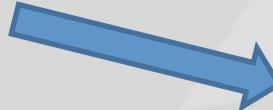
- 系統需要將請求調用者和請求接收者解耦，使得調用者和接收者不直接交互
- 系統需要在不同的時間指定請求、將請求排隊和執行請求
- 系統需要支援命令的撤銷(Undo)操作和恢復(Redo)操作
- 系統需要將一組操作組合在一起，即支援巨集命令

Command(3/13): 模式說明(1/1)



Command(4/13): 範例1 (1/6)

現在我們要設計一個遙控汽車的遙控器，這個遙控器上有八個按鈕，初期我們只會有四個按鍵是有功能的，其餘四個保留做為未來擴充之用，同時我們希望按鍵上的功能不是直接設計在遙控器上



Command(5/13): 範例1 (2/6)

```
// Receiver
abstract class Vehicle
{
    public abstract function powerOn();
    public abstract function powerOff();
    public abstract function move();
    public abstract function stop();
}

// ConcreteReceiver
class Car extends Vehicle
{
    public function powerOn() {
        echo "汽車引擎發動<BR/>";
    }

    public function powerOff() {
        echo "汽車引擎關閉<BR/>";
    }

    public function move() {
        echo "汽車前進<BR/>";
    }

    public function stop() {
        echo "汽車停止<BR/>";
    }

    public function turnLeft() {
        echo "汽車左轉<BR/>";
    }

    public function turnRight() {
        echo "汽車右轉<BR/>";
    }
}
```

```
// ConcreteCommand
class PowerOnCommand implements Command
{
    private $vehicle;

    public function __construct(Vehicle $vehicle) {
        $this->vehicle = $vehicle;
    }

    public function execute() {
        $this->vehicle->powerOn();
    }
}

class PowerOffCommand implements Command
{
    private $vehicle;

    public function __construct(Vehicle $vehicle) {
        $this->vehicle = $vehicle;
    }

    public function execute() {
        $this->vehicle->powerOff();
    }
}

class TurnLeftCommand implements Command
{
    private $vehicle;

    public function __construct(Vehicle $vehicle) {
        $this->vehicle = $vehicle;
    }

    public function execute() {
        $this->vehicle->turnLeft();
    }
}
```

```
class TurnRightCommand implements Command
{
    private $vehicle;

    public function __construct(Vehicle $vehicle) {
        $this->vehicle = $vehicle;
    }

    public function execute() {
        $this->vehicle->turnRight();
    }
}

// 空指令
class NoCommand implements Command
{
    public function execute(){}
}
```

Hiiir

Command(6/13): 範例1 (3/6)

```
// Invoker
class RemoteControl
{
    private $commands = array(8);

    // 建立八個沒任何命令的空按鈕
    function __construct() {
        for($i=0; $i<8; $i++) {
            $this->commands[$i] = new NoCommand();
        }
    }

    // 設定按鈕命令
    function setCommand($slot, Command $cmd) {
        //array_push($this->commands, $cmd);
        $this->commands[$slot] = $cmd;
    }

    // 按下按鈕
    function execute($slot) {
        /*
        foreach($this->commands as $key => $value){
            $this->commands[$key]->execute();
        */
        $this->commands[$slot]->execute();
    }
}
```

Command(7/13): 範例1 (4/6)

```
$remoteControl = new RemoteControl();

$car = new Car();

$carPowerOn = new PowerOnCommand($car);
$carPowerOff = new PowerOffCommand($car);
$carTurnLeft = new TurnLeftCommand($car);
$carTurnRight = new TurnRightCommand($car);

$remoteControl->setCommand(0, $carPowerOn);      // 將按鈕0號設為發動汽車引擎
$remoteControl->setCommand(1, $carPowerOff);      // 將按鈕1號設為關閉汽車引擎
$remoteControl->setCommand(2, $carTurnLeft);       // 將按鈕2號設為汽車左轉
$remoteControl->setCommand(3, $carTurnRight);      // 將按鈕3號設為汽車右轉

$remoteControl->execute(0);          // 按下按鈕0號
$remoteControl->execute(1);          // 按下按鈕1號
$remoteControl->execute(2);          // 按下按鈕2號
$remoteControl->execute(3);          // 按下按鈕3號
```



汽車引擎發動
汽車引擎關閉
汽車左轉
汽車右轉

Command(8/13): 範例1 (5/6)

如果現在想要一鍵就讓遙控車能夠蛇行...該怎麼做呢?

```
// 如果想加一個蛇行的功能時...實作巨集命令
class MarcoCommand implements Command
{
    private $commands = array();

    public function __construct($commands) {
        $this->commands = $commands;
    }

    public function execute() {
        foreach($this->commands as $command) {
            $command->execute();
        }
    }
}
```

Command(9/13): 範例1 (6/6)

```
// 設定蛇行巨集，開啟引擎、左轉、右轉、關閉引擎  
$macroCmd = new MarcoCommand(array($carPowerOn, $carTurnLeft, $carTurnRight, $carPowerOff));  
  
// 將按鈕4號設為蛇行指令  
$remoteControl->setCommand(4, $macroCmd);  
  
// 執行按鈕4號  
$remoteControl->execute(4);
```



汽車引擎發動
汽車左轉
汽車右轉
汽車引擎關閉

Command(10/13): 範例2 (1/4)

現在我們要設計簡單的文字編輯器，具有貼上某字串到主文章上，或將主文章上某字串剪掉，並具有Undo, Redo的功能，該怎麼實作呢？

```
// Receiver
abstract class TextEditor
{
    public abstract function cut($cutText);
    public abstract function paste($pasteText);
    public abstract function getText();
    public abstract function setText($text);
    protected abstract function showText();
}

// ConcreteReceiver
class DommyWord extends TextEditor
{
    private $text;

    public function __construct($text) {
        $this->text = $text;
    }

    public function cut($cutText){
        $this->text = str_replace($cutText, "", $this->text);
        $this->showText();
    }

    public function paste($pasteText) {
        $this->text .= $pasteText;
        $this->showText();
    }
}
```

```
public function getText() {
    return $this->text;
}

public function setText($text) {
    $this->text = $text;
    $this->showText();
}

protected function showText() {
    echo $this->text.'<BR/>';
}
```

Command(11/13): 範例2 (2/4)

```
// Command 這邊用abstract class也可以
interface Command
{
    public function execute($text);
    public function undo();
    public function redo();
}

// ConcreteCommand
class CutCommand implements Command
{
    private $textEditor;
    private $text = array(); // 原始數據備份陣列
    private $textRedo = array(); // 輸入數據備份陣列

    public function __construct(TextEditor $textEditor) {
        $this->textEditor = $textEditor;
    }

    public function execute($text) {
        // 備份舊資料
        array_push($this->text, $this->textEditor->getText());

        // 裁切資料
        $this->textEditor->cut($text);
    }
}
```

```
public function undo() {
    if (count($this->text) <= 0) {
        return false;
    }
    $text = array_pop($this->text);
    array_push($this->textRedo, $this->textEditor->getText());
    $this->textEditor->setText($text);
}

public function redo() {
    if (count($this->textRedo) <= 0) {
        return false;
    }
    $textRedo = array_pop($this->textRedo);
    array_push($this->text, $this->textEditor->getText());
    $this->textEditor->setText($textRedo);
}
```

Command(12/13): 範例2 (3/4)

```
class PasteCommand implements Command
{
    private $textEditor;
    private $text = array(); // 原始數據備份陣列
    private $textRedo = array(); // 輸入數據備份陣列

    public function __construct(TextEditor $textEditor) {
        $this->textEditor = $textEditor;
    }

    public function execute($text) {

        // 備份舊資料
        array_push($this->text, $this->textEditor->getText());

        // 貼上資料
        $this->textEditor->paste($text);
    }

    public function undo(){
        if (count($this->text) <= 0) {
            return;
        }
        $text = array_pop($this->text);
        array_push($this->textRedo, $this->textEditor->getText());
        $this->textEditor->setText($text);
    }

    public function redo(){
        if (count($this->textRedo) <= 0) {
            return;
        }
        $textRedo = array_pop($this->textRedo);
        array_push($this->text, $this->textEditor->getText());
        $this->textEditor->setText($textRedo);
    }
}
```

```
// Invoker
class ControlManager
{
    private $undoList = array();
    private $redoList = array();

    public function __construct() {
    }

    private function storeCommand(Command $cmd) {
        array_push($this->undoList, $cmd);
    }

    public function clearAllCommand() {
        $this->undoList = array();
        $this->redoList = array();
    }

    public function undo(){
        if (count($this->undoList) <= 0) {
            return;
        }

        $cmd = array_pop($this->undoList);
        $cmd->undo();
        array_push($this->redoList, $cmd);
    }

    public function redo(){

        if (count($this->redoList) <= 0) {
            return;
        }

        $cmd = array_pop($this->redoList);
        $cmd->redo();
        array_push($this->undoList, $cmd);
    }

    public function execute(Command $cmd, $text) {
        $this->storeCommand($cmd);
        $cmd->execute($text);
    }
}
```

Command(13/13): 範例2 (4/4)

```
$dummyWord = new DummyWord("今天天氣真好");
$dummyWordPaste = new PasteCommand($dummyWord);
$dumyWordCut = new CutCommand($dummyWord);

$controlManager = new ControlManager();

$controlManager->execute($dummyWordPaste, ",哇哈哈"); // 今天天氣真好,哇哈哈
$controlManager->execute($dummyWordPaste, ",你好嗎"); // 今天天氣真好,哇哈哈,你好嗎

$controlManager->undo();      // 今天天氣真好,哇哈哈
$controlManager->undo();      // 今天天氣真好
$controlManager->redo();       // 今天天氣真好,哇哈哈
$controlManager->redo();       // 今天天氣真好,哇哈哈,你好嗎

echo "<br/>";

$controlManager->execute($dummyWordCut, "今天"); // 天氣真好,哇哈哈,你好嗎

// 三次undo
$controlManager->undo(); // 今天天氣真好,哇哈哈,你好嗎
$controlManager->undo(); // 今天天氣真好,哇哈哈
$controlManager->undo(); // 今天天氣真好

// 到第四次undo已經沒有資料了，所以沒有任何字串出現
$controlManager->undo();

// 三次redo
$controlManager->redo(); // 今天天氣真好,哇哈哈
$controlManager->redo(); // 今天天氣真好,哇哈哈,你好嗎
$controlManager->redo(); // 天氣真好,哇哈哈,你好嗎
```



今天天氣真好,哇哈哈
今天天氣真好,哇哈哈,你好嗎
今天天氣真好,哇哈哈
今天天氣真好
今天天氣真好,哇哈哈
今天天氣真好,哇哈哈,你好嗎

天氣真好,哇哈哈,你好嗎
今天天氣真好,哇哈哈,你好嗎
今天天氣真好,哇哈哈
今天天氣真好
今天天氣真好,哇哈哈
今天天氣真好,哇哈哈,你好嗎
天氣真好,哇哈哈,你好嗎

Interpreter(解釋器模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

Interpreter(1/9): UML(1/1)

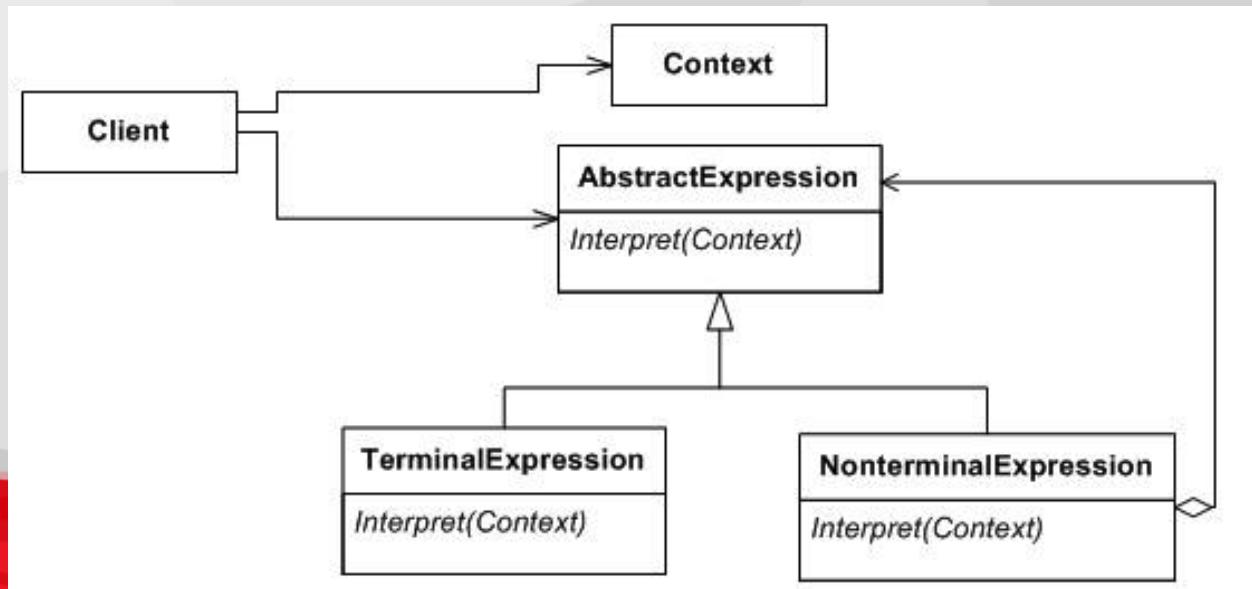
AbstractExpression：宣告一個抽象的解釋操作，此介面為語法樹所有節點所共享

TerminalExpression：實現與文法中的結束符號相關聯的解釋操作

NonterminalExpression：實現與文法中的非結束符號相關聯的解釋操作

Context: Interpreter需要解釋的信息內容

Client：負責建立語法樹，該語法樹由**NonterminalExpression**與**TerminalExpression**裝配而成



Interpreter(2/9): 模式分析(1/1)

優點：

- 由於文法是許多類別的集合，所以可以輕易地改變與擴充文法。
- 可藉由在類別結構中加入新的方法，可以同時增加新的行為，例如輸出格式美化或複雜的程式檢查。

缺點：

- 對於過於複雜的文法難以維護。
- 執行效率較低。
- 應用場景有限。

使用時機：

- 需要實踐一個簡單的語言時。
- 一些重複出現的問題可以用一種簡單的語言來進行表達。
- 文法較為簡單，而且效率不是關鍵問題時。

Interpreter(3/9): 範例(1/7)

如果我們想寫一個計算機可以輸入字串，自動進行四則計算，要怎麼做呢？

"10 - 5 - 5 + 12 - 16"



Interpreter(4/9): 範例(2/7)

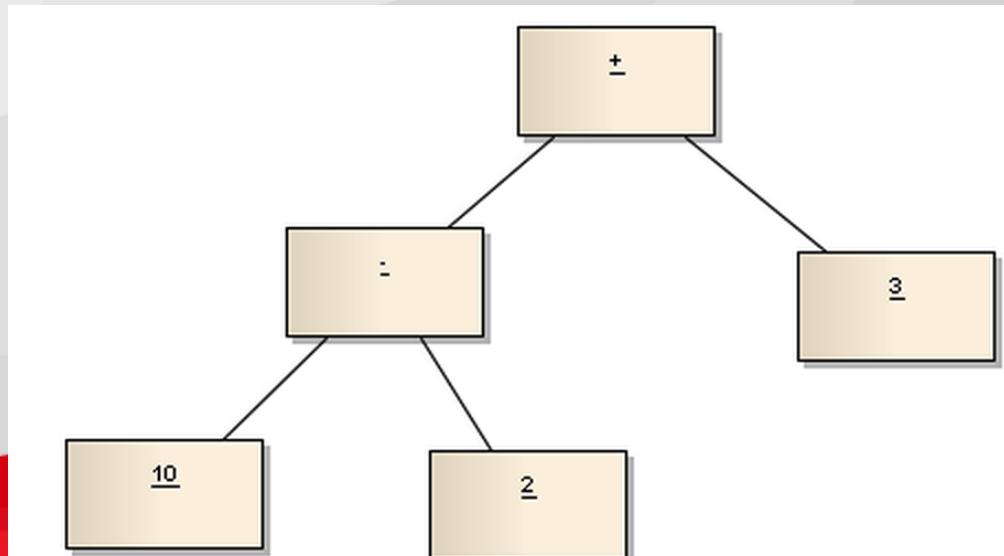
分析文法：

- + = AddExpression = Expression + Expression
- - = SubtractExpression = Expression - Expression
- Expression = NumberExpression | AddExpression | SubtractExpression

Expression1 = 10 - 2 + 3 =

SubstractExpression + NumberExpression =

Expression + Expression = AddExpression = 11



Interpreter(5/9): 範例(3/7)

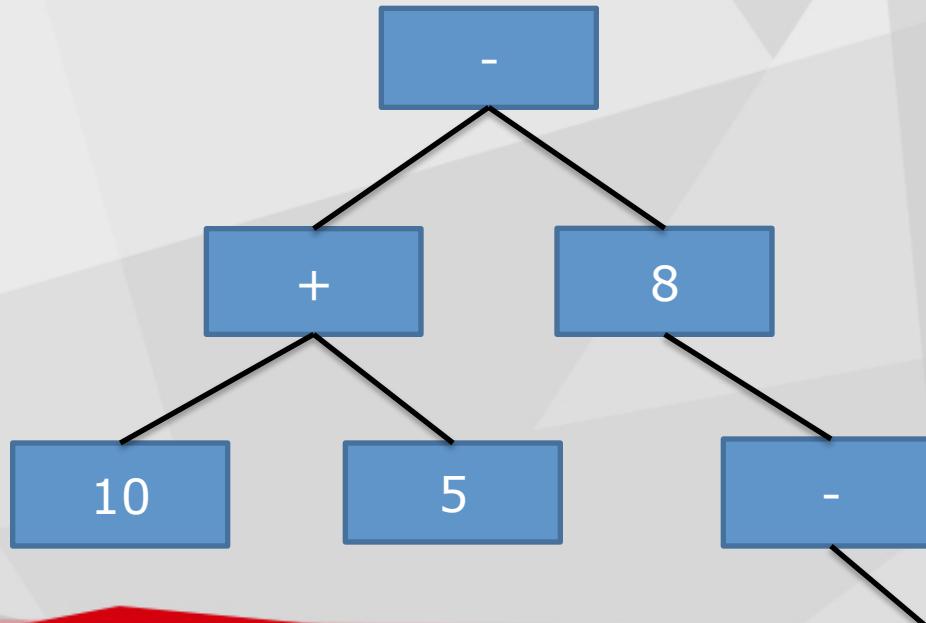
分析文法：

Expression1 = 10 + 5 - 8 - 2 =

AddExpression – NumberExpression - NumberExpression =

SubtractExpression – NumberExpression =

Expression - Expression = SubtractExpression = 5



Interpreter(6/9): 範例(4/7)

```
// AbstractExpression  AbstractExpression
interface AbstractExpression
{
    public function interpret();
}

// TerminalExpression TerminalExpression
class NumberExpression implements AbstractExpression
{
    protected $number = 0;

    public function __construct($number) {
        $this->number = $number;
    }

    public function interpret() {
        return $this->number;
    }
}
```

Interpreter(7/9): 範例(5/7)

NonterminalExpression

```
// NonterminalExpression
class AddExpression implements AbstractExpression
{
    protected $leftExpression;
    protected $rightExpression;

    public function __construct($left, $right) {
        $this->leftExpression = $left;
        $this->rightExpression = $right;
    }

    public function interpret() {
        return $this->leftExpression->interpret() + $this->rightExpression->interpret();
    }
}

// NonterminalExpression
class SubtractExpression implements AbstractExpression
{
    protected $leftExpression;
    protected $rightExpression;

    public function __construct($left, $right) {
        $this->leftExpression = $left;
        $this->rightExpression = $right;
    }

    public function interpret() {
        return $this->leftExpression->interpret() - $this->rightExpression->interpret();
    }
}
```

NonterminalExpression

Interpreter(8/9): 範例(6/7)

```
// Context
class Calculator
{
    private $statement = "";
    private $expression = null;

    public function build($statement) {
        $left = null;
        $right = null;
        $stack = array();
        $statementArr = explode(" ", $statement);

        for ($i = 0, $statementArrCount = count($statementArr); $i < $statementArrCount; $i++) {

            if ($statementArr[$i] == "+") {
                $left = array_pop($stack);
                $val = $statementArr[++$i];

                $right = new NumberExpression($val);
                $stack[] = new AddExpression($left, $right);

            } else if($statementArr[$i] == "-") {
                $left = array_pop($stack);
                $val = $statementArr[++$i];

                $right = new NumberExpression($val);
                $stack[] = new SubtractExpression($left, $right);

            } else {
                $stack[] = new NumberExpression($statementArr[$i]);
            }
        }
        $this->expression = array_pop($stack);
    }

    public function compute() {
        return $this->expression->interpret();
    }
}
```

Context

Interpreter(9/9):範例(7/7)

```
include_once '.../class/pattern/interpreter.php';

$statement = "1 + 2 + 8 - 50";

$calculator = new Calculator();
$calculator->build($statement);
$result = $calculator->compute();

echo "計算結果:" . $result;
```



計算結果:-39

Iterator(迭代器模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

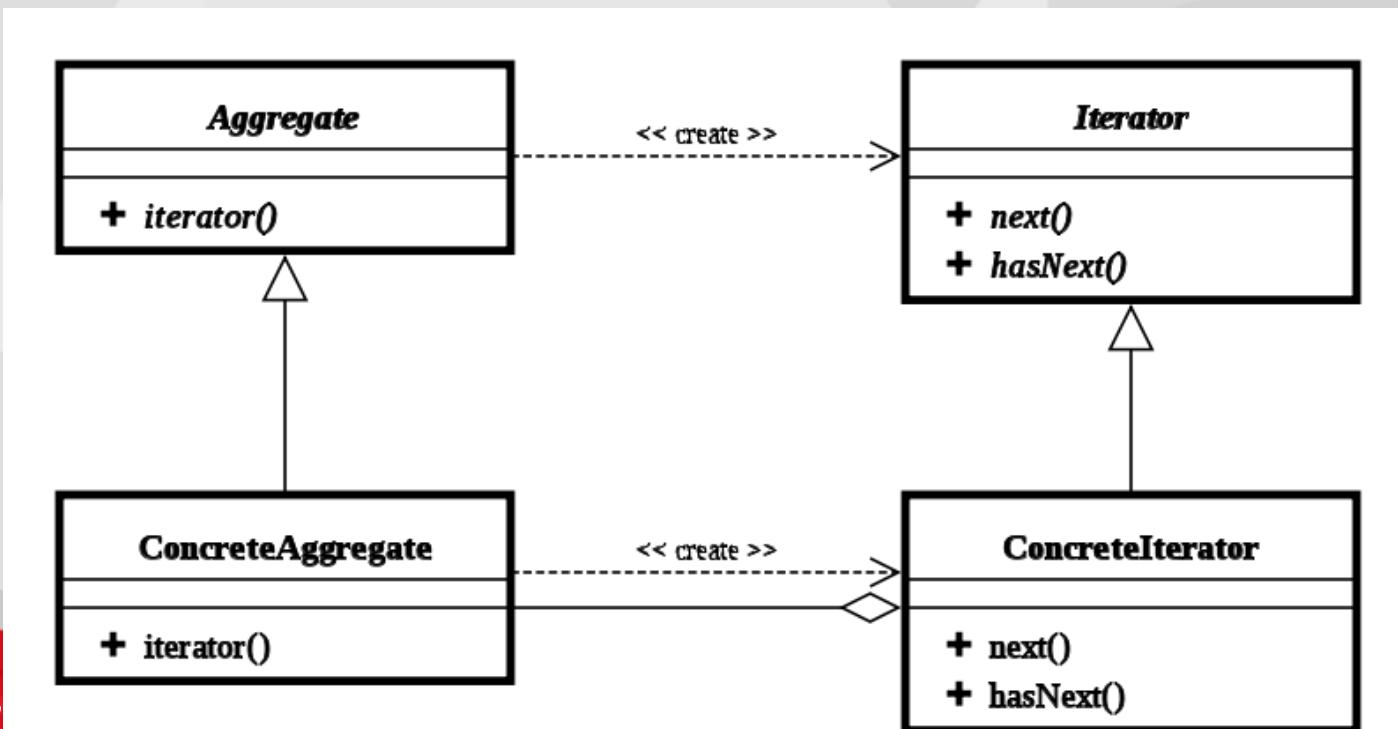
Iterator(1/9): UML(1/1)

Iterator：定義訪問和遍歷元素的介面(接口)。

ConcreteIterator：實現Iterator所定義的介面(接口)，並記錄遍歷當前位置。

Aggregate：提供創建具體Iterator的介面(接口)。

ConcreteAggregate: 實現具體Iterator的介面(接口)。



Iterator(2/9): 模式分析(1/1)

優點：

- 它支援以不同的方式遍歷一個聚合物件。
- 反覆運算器簡化了Aggregate類別。
- 在同一個聚合上可以有多個遍歷。
- 在Iterator模式中，增加新的Aggregate類別和Iterator類別都很方便，無須修改原有代碼，滿足“開閉原則”的要求。

缺點：

- 由於Iterator模式將存儲資料和遍歷資料的職責分離，增加新的Aggregate類別需要對應增加新的Iterator類別，導致類別的個數成對增加，這在一定程度上增加了系統的複雜性。

使用時機：

- 訪問一個聚合物件的內容而無須暴露它的內部表示。
- 需要為聚合物件提供多種遍歷方式。
- 為遍歷不同的聚合結構提供一個統一的介面。
- Iterator模式常跟Composite模式一起使用。

Iterator(3/9): 模式說明(1/2)

外部迭代器(External Iterator)

由客戶端控制迭代，由客戶端負責推進整個迭代過程。

內部迭代器(Internal Iterator)

由迭代器自己控制迭代，客戶端只要將一個操作傳遞給一個內部迭代器，該迭代器會依次在每個元素上應用該操作。(例如我們使用foreach的時候)

Iterator(4/9)：模式說明(2/2)

由於使用迭代來操作聚合物件已經是常用的基本功能之一，所有很多程式設計語言(Java、PHP、C#、C++)的類別庫都已經實現了iterator模式。因此在沒特殊需求的情況下，已經很少自己寫iterator，只需要直接使用定義好的iterator即可。

不同語言的Iterator說明：

- PHP：<http://php.net/manual/en/spl.iterators.php>
- Java：<http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html>
- C#：[http://msdn.microsoft.com/zh-tw/library/dscyy5s0\(v=vs.80\).aspx](http://msdn.microsoft.com/zh-tw/library/dscyy5s0(v=vs.80).aspx)
- C++：<http://www.cplusplus.com/reference/iterator/>

Iterator(5/9):範例(1/6)

在Composite範例中，如果我們想要用迭代器...

攻擊力:4



弓箭手

攻擊力:15



騎兵

攻擊力:44



雷射加農砲

攻擊力:15, 不可載運騎兵



www.shutterstock.com · 35511682

裝甲運兵車

Iterator(6/9):範例(1/4)

```
abstract class Unit
{
    abstract function getName();

    public function getComposite(){
        return null;
    }

    abstract function bombardStrength(); //輸出戰鬥單位的攻擊力
}

// Aggregate
abstract class CompositeUnit extends Unit
{
    private $units = array();

    public function getName() {

    }

    public function getComposite() {
        return $this;
    }

    public function units() {
        return $this->units;
    }

    public function addUnit(Unit $unit) { //將一戰鬥單位加入到軍隊群組中
        if(in_Array($unit, $this->units, true)) {
            return;
        }
        $this->units[] = $unit;
    }

    public function removeUnit(Unit $unit) { //將一戰鬥單位從軍隊群組中移除
        //$this->units = array_udiff($this->units, array($unit), function($a, $b){ return ($a == $b)?0:1; }); //PHP 5.3寫法
        $this->units = array_udiff($this->units, array($unit), create_function('$a, $b', 'return ($a == $b)?0:1;'));
    }

    public function createIterator() {
        return new CompositeIterator($this);
    }
}
```



Iterator(7/9):範例(2/4)

```
// ConcreteAggregate
class Army extends CompositeUnit
{
    public function getName() {
        echo '軍隊<br>';
    }

    public function bombardStrength() { //計算總攻擊力
        $ret = 0;
        foreach( $this->units() as $unit) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}

// ConcreteAggregate
class TroopCarrier extends CompositeUnit
{
    public function getName() {
        echo '裝甲運兵車<br>';
    }

    public function addUnit(Unit $unit) { //將單位加入
        if($unit instanceof Cavalry) {
            throw new Exception("Can't get a horse on the vehicle");
        }
        parent::addUnit($unit);
    }

    public function bombardStrength() {
        return 15;
    }
}
```

```
// 弓箭手
class Archer extends Unit
{
    public function getName() {
        echo '弓箭手<br>';
    }

    public function bombardStrength() {
        return 4;
    }
}

// 雷射加農砲
class LaserCannonUnit extends Unit
{
    public function getName() {
        echo '雷射加農砲<br>';
    }

    public function bombardStrength() {
        return 44;
    }
}

// 騎兵
class Cavalry extends Unit
{
    public function getName() {
        echo '騎兵<br>';
    }

    public function bombardStrength() {
        return 15;
    }
}
```

Iterator(8/9):範例(3/4)

```
// Iterator
interface MyIterator
{
    public function next();
    public function hasNext();
}
```

```
// ConcreteIterator
class CompositeIterator implements MyIterator
{
    protected $units;
    protected $currentIndex = 0;

    public function __construct(CompositeUnit $compositeUnit) {
        $this->units = $compositeUnit->units();
    }

    public function current() {
        return $this->units[$this->currentIndex];
    }

    public function next() {
        if ($this->hasNext()) {
            $component = $this->units[$this->currentIndex];
            $this->currentIndex++;
            return $component;
        } else {
            echo "沒有物件了喔!<br>";
        }
    }

    public function hasNext() {
        if (empty($this->units) || !isset($this->units[$this->currentIndex])) {
            return false;
        } else {
            if (count($this->units) > $this->currentIndex) {
                return true;
            } else {
                return null;
            }
        }
    }
}
```



Iterator(9/9): 範例(4/4)

```
include_once '../class/pattern/iterator.php';

$archer = new Archer();
$main_army = new Army();

$main_army->addUnit($archer);
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCannonUnit());

$sub_army = new Army();

$sub_army->addUnit(new Archer());
$sub_army->addUnit(new Cavalry());
$sub_army->addUnit(new Cavalry());

$main_army->addUnit($sub_army);

compositeIterator = $main_army->createIterator();

//unit = $compositeIterator->current();
//echo $unit->getName();

$unit = $compositeIterator->next();
echo $unit->getName();

$compositeIterator->next();
$compositeIterator->next();

echo "<br>";
```

```
$subCompositeIterator = $unit->createIterator();

$unit = $subCompositeIterator->next();
echo $unit->getName();

$unit = $subCompositeIterator->next();
echo $unit->getName();

$unit = $subCompositeIterator->next();
echo $unit->getName();

$subCompositeIterator->next();
```



弓箭手
弓箭手
雷射加農砲
軍隊
沒有物件了喔!
沒有物件了喔!
沒有物件了喔!

弓箭手
騎兵
騎兵
沒有物件了喔!



Mediator(中介者模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

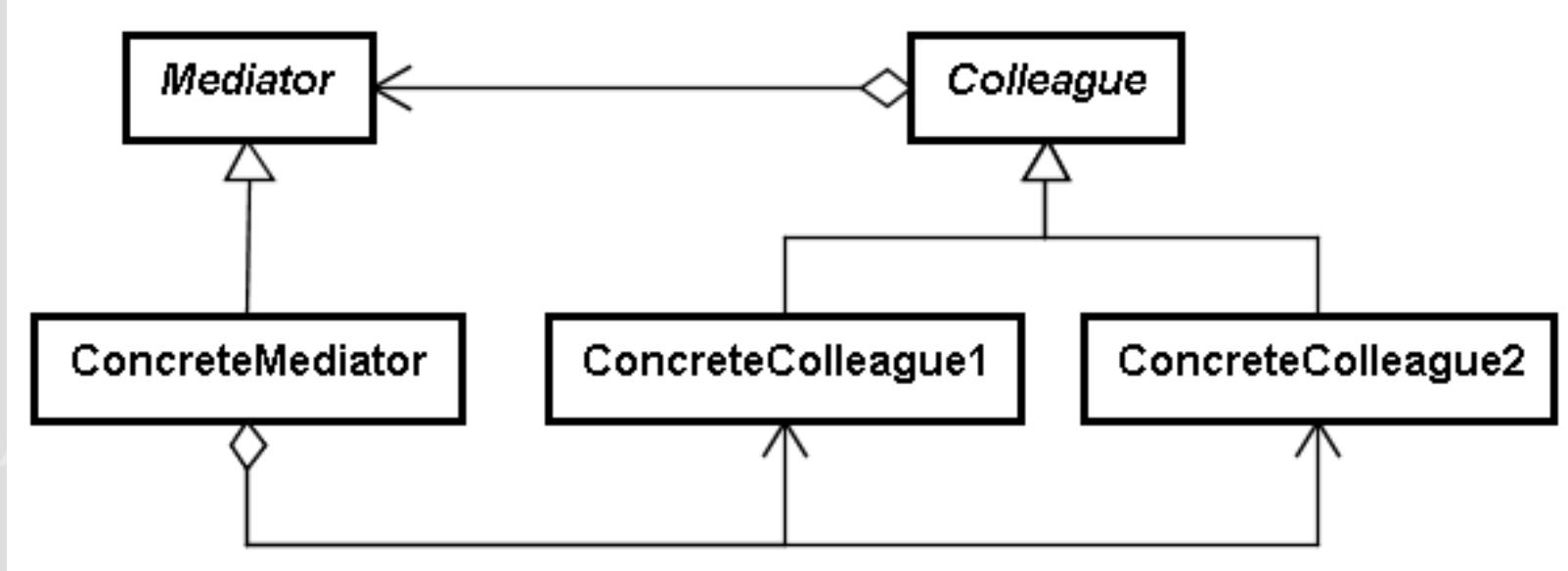
Mediator(1/11):UML(1/1)

Mediator：抽象定義Colleague通信的接口。

ConcreteMediator：具體實現Colleague之間的通信接口。

Colleague：參與通信的抽象Colleague。

ConcreteColleague: 具體實現參與通信的Colleague



Mediator(2/11): 模式分析(1/2)

優點：

- 鬆散耦合:** Mediator模式通過把多個同事物件之間的交互封裝到Mediator物件裡面，從而使得Colleague物件之間鬆散耦合，基本上可以做到互不依賴。這樣一來，同事物件就可以獨立的變化和複用，而不再像以前那樣“牽一髮而動全身”。
- 集中控制交互:** 多個同事物件的交互，被封裝在Mediator物件裡面集中管理，使得這些交互行為發生變化的時候，只需要修改Mediator物件就可以了，當然如果是已經做好的系統，那就擴展Mediator物件，而各個Colleague類別不需要做修改。

缺點：

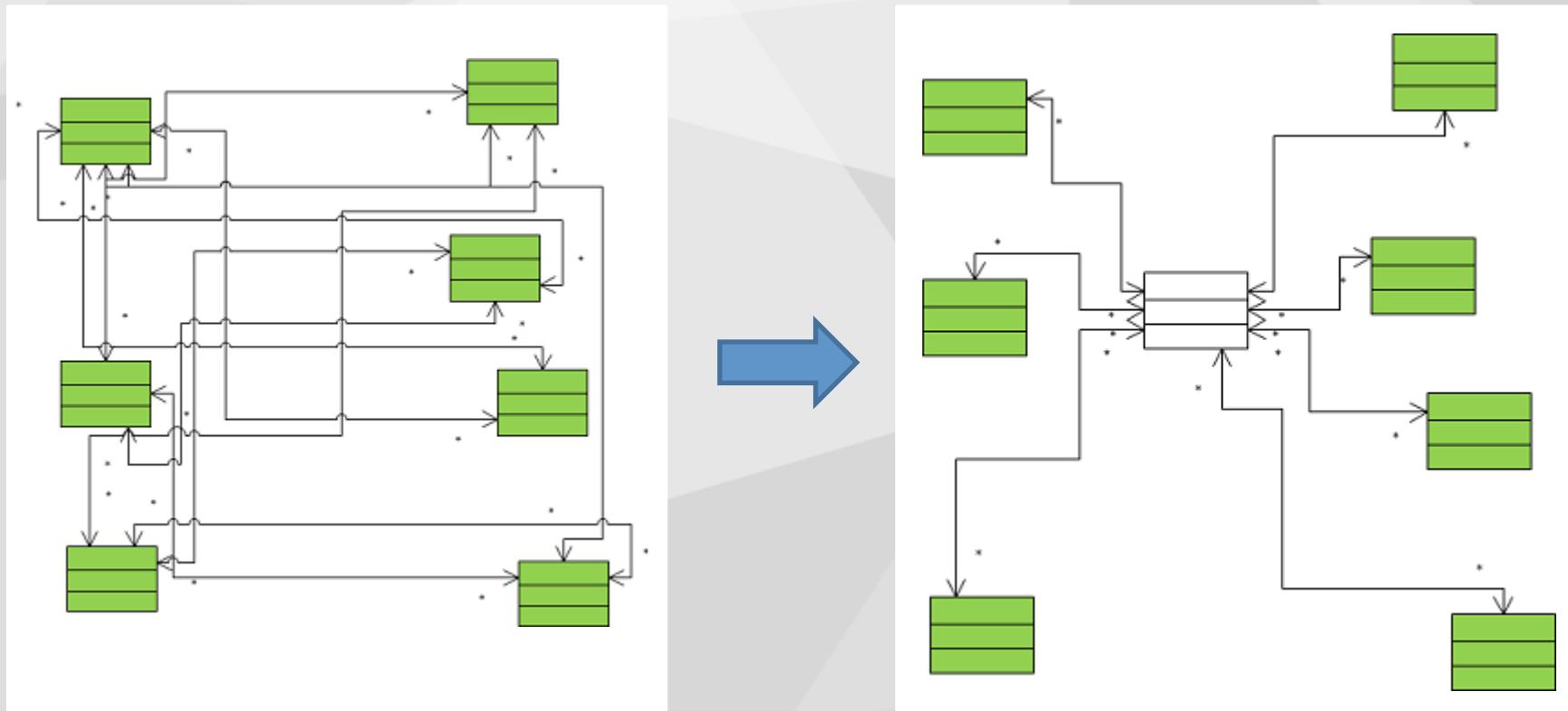
- Mediator承擔了較多的責任，所以一旦Meditor出現了問題，那整個系統就會受到重大的影響甚至崩潰。
- 如果Colleague對象的交互非常多，而且比較複雜，當這些複雜性全部集中到Mediator的時候，會導致Mediator對象變得十分的複雜，而且難於管理和維護。

Mediator(3/11): 模式分析(2/2)

使用時機：

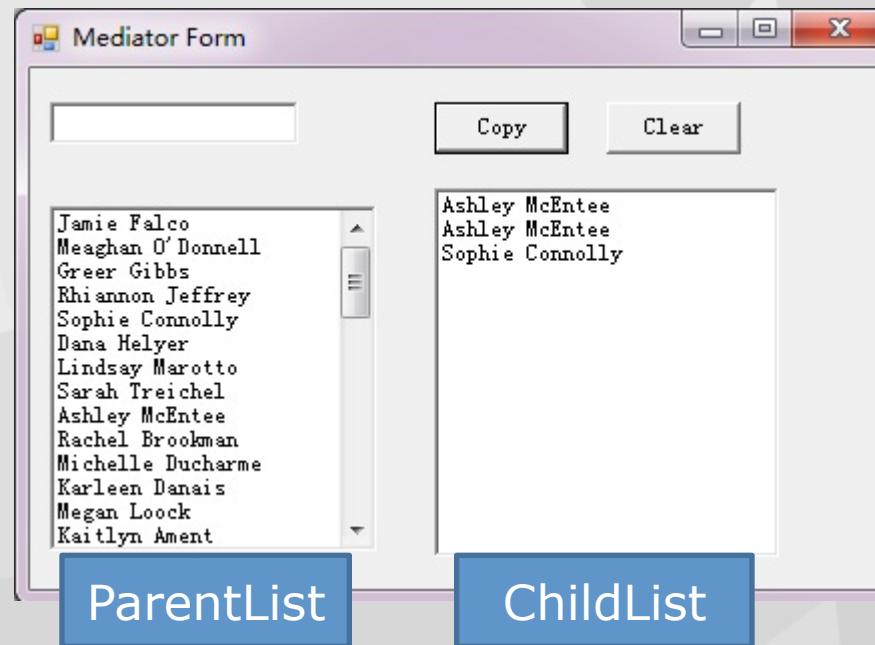
- 如果一組物件之間的通信方式比較複雜，導致相互依賴、結構混亂，可以採用Mediator模式，把這些物件相互的交互管理起來，各個物件都只需要和Mediator交互，從而使得各個物件鬆散耦合，結構也更清晰易懂。
- 如果一個物件引用很多的物件，並直接跟這些物件交互，導致難以複用該物件。可以採用Mediator模式，把這個物件跟其它物件的交互封裝到Mediator物件裡面，這個物件就只需要和Mediator物件交互就可以

Mediator(4/11): 模式說明(1/1)



Mediator(5/11):範例 (1/5)

假設我們要做一個列表選擇程式，當程式啟動的時候，Copy跟Clear按鈕都被禁用，當有項目被選中的時候Copy按鈕可以使用，當右側清單有資訊了以後Clear按鈕可以使用，按下Clear以後兩個按鈕都不可用。



Mediator(6/11):範例 (2/5)

```
// ConcreteMediator
class FormMediator {
    private $copyBtn = null;
    private $clearBtn = null;
    private $parentList = null;
    private $childList = null;

    public function __construct($copyBtn, $clearBtn, $parentList, $childList) {
        $this->copyBtn = $copyBtn;
        $this->clearBtn = $clearBtn;
        $this->parentList = $parentList;
        $this->childList = $childList;
    }

    // 按下複製按鈕
    public function copyBtnClick() {
        if($this->copyBtn->getStatus() == 1)
            echo "複製按鈕按下<br/>";
        else
            echo "複製按鈕目前禁用中<br/>";
    }

    // 按下清除按鈕
    public function clearBtnClick() {
        if ($this->clearBtn->getStatus() == 1) {
            $this->childList->clearData();
            $this->copyBtn->setStatus(2); // 將Copy按鈕設為disabled
            $this->clearBtn->setStatus(2); // 將Clear按鈕設為disabled
            echo "清除按鈕按下，已將子列表的內容清除!<br/>";
        } else {
            echo "清除按鈕目前禁用中<br/>";
        }
    }
}
```

```
// 選擇ParentList的項目
public function selectParentData($count) {
    $data = $this->parentList->getData();
    $this->childList->setData($data[$count]);
    $this->copyBtn->setStatus(1); // 將Copy按鈕設為enabled
    $this->clearBtn->setStatus(2); // 將Clear按鈕設為disabled

    echo "母項目選擇了" . $count . "，以下是子項目列表:<br/>";
    $this->childList->showData();
    echo "<br/>";

}

// 選擇ChildList的項目
public function selectChildData($count) {
    $this->clearBtn->setStatus(1); // 將Clear按鈕設為enabled
    echo "子項目選擇了" . $count . "<br/>";
}
```

Mediator(7/11):範例 (3/5)

```
// Colleague
abstract class Colleague
{
    protected $mediator = null;
    protected $name = "";
    protected $status = 2; //1:enabled. 2:disabled

    public function setStatus($status) {
        $this->status = $status;
    }

    public function getStatus() {
        return $this->status;
    }

    public function setMediator(FormMediator $mediator) {
        $this->mediator = $mediator;
    }

    abstract public function execute();
}
```

```
class CopyButton extends Colleague
{
    public function execute() {
        $this->mediator->copyBtnClick();
    }
}

class ClearButton extends Colleague
{
    public function execute() {
        $this->mediator->clearBtnClick();
    }
}
```

Mediator(8/11):範例 (4/5)

```
abstract class ListBox
{
    protected $mediator = null;
    protected $list = array();

    public function setMediator(FormMediator $mediator) {
        $this->mediator = $mediator;
    }

    public function setData($data) {
        $this->list = $data;
    }

    public function getData() {
        return $this->list;
    }

    public function showData() {
        print_r($this->list);
    }

    public function clearData() {
        $this->list = array();
    }

    abstract public function select($count = 0);
}
```

```
class ParentListBox extends ListBox
{
    public function select($count = 0) {
        $this->mediator->selectParentData($count);
    }
}

class ChildListBox extends ListBox
{
    public function select($count = 0) {
        $this->mediator->selectChildData($count);
    }
}
```

Mediator(9/11):範例 (5/5)

```
$data["台北市"][] = "內湖區";
$data["台北市"][] = "大安區";
$data["台北市"][] = "中正區";
$data["台北市"][] = "中山區";
$data["高雄市"][] = "旗津區";
$data["高雄市"][] = "那馬夏區";
$data["高雄市"][] = "小港區";

$copyBtn = new CopyButton();           // 建立Copy按鈕
$clearBtn = new ClearButton();         // 建立Clear按鈕
$parentList = new ParentListBox();     // 建立母列表
$childList = new ChildListBox();       // 建立子列表

// 建立Mediator
$formMediator = new FormMediator($copyBtn, $clearBtn, $parentList, $childList);

$copyBtn->setMediator($formMediator);
$clearBtn->setMediator($formMediator);
$parentList->setMediator($formMediator);
$childList->setMediator($formMediator);

$copyBtn->execute();    // 複製按鈕目前禁用中
$clearBtn->execute();   // 清除按鈕目前禁用中

$parentList->setData($data);
$parentList->select("台北市");      // 母項目選擇了台北市，以下是子項目列表：
                                         // Array ( [0] => 內湖區 [1] => 大安區 [2] => 中正區 [3] => 中山區
$copyBtn->execute();                // 複製按鈕按下
$clearBtn->execute();               // 清除按鈕目前禁用中

$childList->select("內湖區");        // 子項目選擇了內湖區
$copyBtn->execute();                // 複製按鈕按下
$clearBtn->execute();               // 清除按鈕按下，已將子列表的內容清除！

$copyBtn->execute();                // 複製按鈕目前禁用中
$clearBtn->execute();               // 清除按鈕目前禁用中
```

Mediator(10/11):延伸討論 (1/2)

其一：是否有必要為同事物件定義一個公共的父類？

為了使用Mediator，就讓這些Colleague物件繼承一個父類別，這是很不好的；再說了，這個父類別目前也沒有什麼特別的公共功能，也就是說繼承它也得不到多少好處。

在實際開發中，很多相互交互的物件本身是沒有公共父類別的，強行加上一個父類別，會讓這些物件實現起來特別彆扭。

其二：Colleague類別有必要持有Mediator物件嗎？

Colleague類別需要知道Mediator物件，以便當它們發生改變的時候，能夠通知Mediator物件，但是，是否需要作為屬性，並通過構造方法傳入，這麼強的依賴關係呢？也可以有簡單的方式去通知Mediator物件，比如把Mediator物件做成Singleton，直接在Colleague類別的方法裡面去調用Mediator物件。

Mediator(11/11):延伸討論 (2/2)

其三：是否需要Mediator介面？

在實際開發中，很常見的情況是不需要Mediator介面的，而且Mediator物件也不需要創建很多個實例，因為Mediator是用來封裝和處理同事物件的關係的，它一般是沒有狀態需要維護的，因此Mediator通常可以實現成單例。

其四：Mediator物件是否需要持有所有的Colleague？

雖說Mediator物件需要知道所有的Colleague類別，這樣Mediator才能與它們交互。但是是否需要做為屬性這麼強烈的依賴關係，而且Mediator物件在不同的關係維護上，可能會需要不同的Colleague物件的實例，因此可以在Mediator處理的方法裡面去創建、或者獲取、或者從參數傳入需要的Colleague物件。

其五：Colleague物件只是提供一個公共的方法，來接受Colleague物件的通知嗎？

從示例就可以看出來，在公共方法裡，還是要去區分到底是誰調過來，這還是簡單的，還沒有去區分到底是什麼樣的業務觸發調用過來的，因為不同的業務，引起的與其它物件的交互是不一樣的。

因此在實際開發中，通常會提供具體的業務通知方法，這樣就不用再去判斷到底是什麼對象，具體是什麼業務了。

Memento(備忘錄模式)

Hiiir

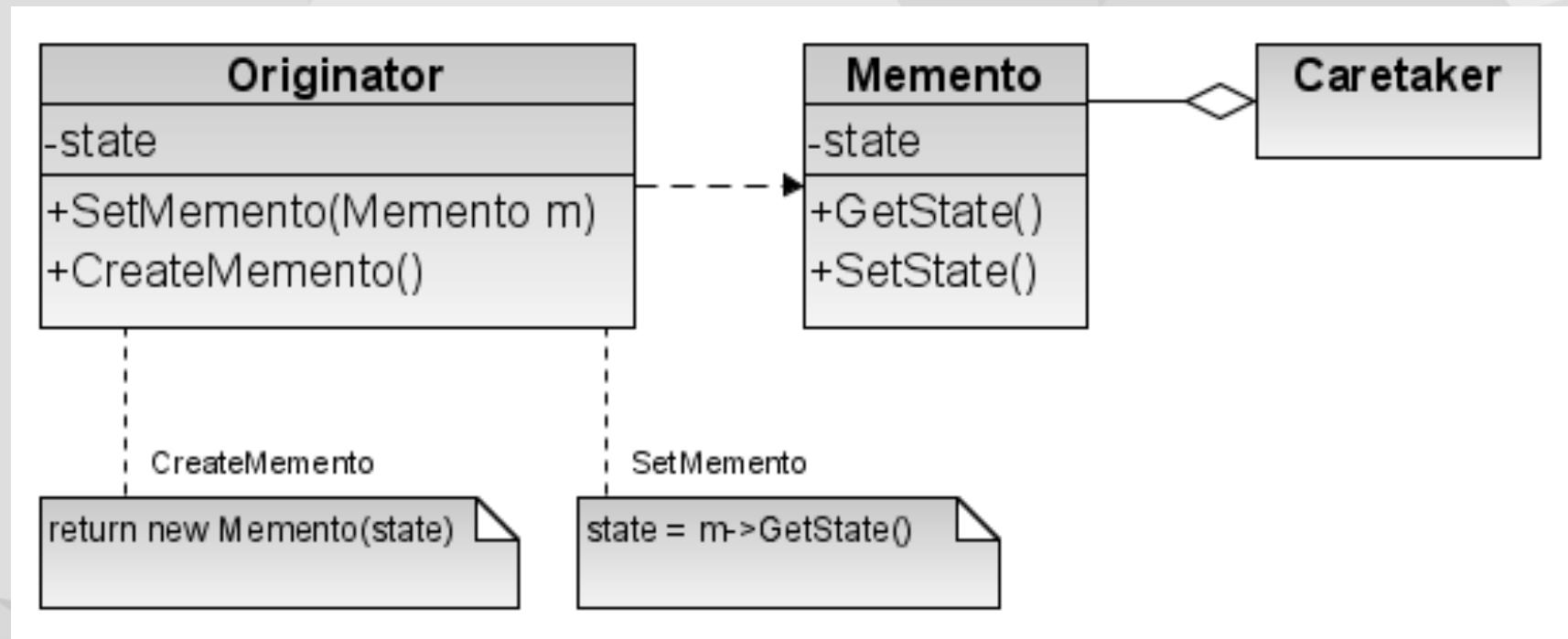
行動×商務 社群×媒體
mobile e-commerce social media

Memento(1/9): UML(1/1)

Originator : 需要Memento保存狀態以便恢復的物件

Memento : 由Originator創建，儲存Originator物件的內部狀態；保護其內容不被Originator物件以外的其他物件所讀取。

Caretaker : 負責在適當時機使用Memento物件來保存/恢復Originator物件的狀態。



Memento(2/9): 模式分析(1/1)

優點：

- 將被儲存的狀態放在外面，不要和關鍵物件混在一起，這可以幫助維護內聚力。
- 讓關鍵物件的資料封裝不受破壞。
- 提供了容易實踐的恢復能力。

缺點：

- 儲存和回復狀態的過程，可能相當耗費時間與效能。

使用時機：

- 當系統必須要保存一個物件在某一個時刻的狀態，以方便需要時能恢復到先前的狀態，並保持關鍵物件的封裝時。

Memento(3/9):範例 (1/6)

假設我們現在要設計一個RPG遊戲，並且要能在角色死亡時，將角色狀態回到最後一次儲存時的狀態，該怎麼做呢？



Memento(4/9):未使用Memento範例(2/6)

```
// 遊戲角色
class Role
{
    private $hp = 0; // 體力
    private $mp = 0; // 魔力
    private $dex = 0; // 敏捷
    private $atk = 0; // 攻擊力
    private $def = 0; // 防禦力

    public function __construct($hp, $mp, $dex, $atk, $def) {
        $this->hp = $hp;
        $this->mp = $mp;
        $this->dex = $dex;
        $this->atk = $atk;
        $this->def = $def;
    }

    public function getHp() {
        return $this->hp;
    }

    public function getMp() {
        return $this->mp;
    }

    public function getDex() {
        return $this->dex;
    }

    public function getAtk() {
        return $this->atk;
    }
}
```

```
public function getDef() {
    return $this->def;
}

public function setState($hp, $mp, $dex, $atk, $def) {
    $this->hp = $hp;
    $this->mp = $mp;
    $this->dex = $dex;
    $this->atk = $atk;
    $this->def = $def;
}

public function showState() {
    echo "角色當前狀態:<br/>";
    echo "體力:{$this->hp}<br/>";
    echo "魔力:{$this->mp}<br/>";
    echo "敏捷:{$this->dex}<br/>";
    echo "攻擊力:{$this->atk}<br/>";
    echo "防禦力:{$this->def}<br/><br/>";
}

public function fight() {
    $this->hp = 0;
    $this->mp = 0;
    $this->dex = 0;
    $this->atk = 0;
    $this->def = 0;

    echo "開打，被魔王秒殺...<br/>";
}
```

Memento(5/9):未使用Memento範例(3/6)

```
// 未使用Memento範例
$jack = new Role(100, 30, 60, 40, 35); // 建立遊戲角色
$jack->showState();

$backup = new Role(0, 0, 0, 0, 0);
$backup->setState($jack->getHp(), $jack->getMp(), // 備份目前狀態,
                    $jack->getDex(), $jack->getAtk(), // 但是這邊暴露了實作細節
                    $jack->getDef());

$jack->fight(); // 開打，死掉了
$jack->showState();

$jack->setState($backup->getHp(), $backup->getMp(), // 回復之前狀態
                  $backup->getDex(), $backup->getAtk(),
                  $backup->getDef());

$jack->showState();
```



角色當前狀態:
體力:100
魔力:30
敏捷:60
攻擊力:40
防禦力:35

開打，被魔王秒殺...

角色當前狀態:
體力:0
魔力:0
敏捷:0
攻擊力:0
防禦力:0

角色當前狀態:
體力:100
魔力:30
敏捷:60
攻擊力:40
防禦力:35

Memento(6/9): 使用Memento範例(4/6)

```
// 遊戲角色(Originator)
class Role
{
    private $hp = 0; // 體力
    private $mp = 0; // 魔力
    private $dex = 0; // 敏捷
    private $atk = 0; // 攻擊力
    private $def = 0; // 防禦力

    public function __construct($hp, $mp, $dex, $atk, $def) {
        $this->hp = $hp;
        $this->mp = $mp;
        $this->dex = $dex;
        $this->atk = $atk;
        $this->def = $def;
    }

    public function saveState() {
        return new Memento($this->hp, $this->mp, $this->dex, $this->atk, $this->def);
    }

    public function recoveryState(Memento $memento) {
        $this->hp = $memento->getHp();
        $this->mp = $memento->getMp();
        $this->dex = $memento->getDex();
        $this->atk = $memento->getAtk();
        $this->def = $memento->getDef();
    }
}
```

```
public function showState() {
    echo "角色當前狀態:<br/>";
    echo "體力:{$this->hp}<br/>";
    echo "魔力:{$this->mp}<br/>";
    echo "敏捷:{$this->dex}<br/>";
    echo "攻擊力:{$this->atk}<br/>";
    echo "防禦力:{$this->def}<br/><br/>";
}

public function fight()
{
    $this->hp = 0;
    $this->mp = 0;
    $this->dex = 0;
    $this->atk = 0;
    $this->def = 0;

    echo "開打，被魔王秒殺...<br/>";
}
```

Memento(7/9): 使用Memento範例(5/6)

```
class Memento
{
    private $hp = 0; // 體力
    private $mp = 0; // 魔力
    private $dex = 0; // 敏捷
    private $atk = 0; // 攻擊力
    private $def = 0; // 防禦力

    public function __construct($hp, $mp, $dex, $atk, $def) {
        $this->hp = $hp;
        $this->mp = $mp;
        $this->dex = $dex;
        $this->atk = $atk;
        $this->def = $def;
    }

    public function getHp() {
        return $this->hp;
    }

    public function getMp() {
        return $this->mp;
    }

    public function getDex() {
        return $this->dex;
    }

    public function getAtk() {
        return $this->atk;
    }

    public function getDef() {
        return $this->def;
    }
}
```

```
class Caretaker
{
    private $memento;

    public function getMemento() {
        return $this->memento;
    }

    public function setMemento(Memento $memento) {
        $this->memento = $memento;
    }
}
```

Memento(8/9): 使用Memento範例(6/6)

```
// 使用Memento範例
$jack = new Role(100, 30, 60, 40, 35); // 建立遊戲角色
$jack->showState();

$caretaker = new Caretaker();

$caretaker->setMemento($jack->saveState()); // 備份目前狀態

$jack->fight(); // 開打，死掉了
$jack->showState();

$jack->recoveryState($caretaker->getMemento()); // 回復之前狀態
$jack->showState();
```



角色當前狀態:
體力:100
魔力:30
敏捷:60
攻擊力:40
防禦力:35

開打，被魔王秒殺...
角色當前狀態:
體力:0
魔力:0
敏捷:0
攻擊力:0
防禦力:0

角色當前狀態:
體力:100
魔力:30
敏捷:60
攻擊力:40
防禦力:35

Memento(9/9):Comment vs. Memento

Comment：支持多層狀態的回覆

Mediator：僅支持一層狀態的回覆

Command模式在每一個undo中，可以使用Memento來保存對象的狀態

- Command can use Memento to maintain the state required for an undo operation. [GoF, p242]

Observer(觀察者模式)

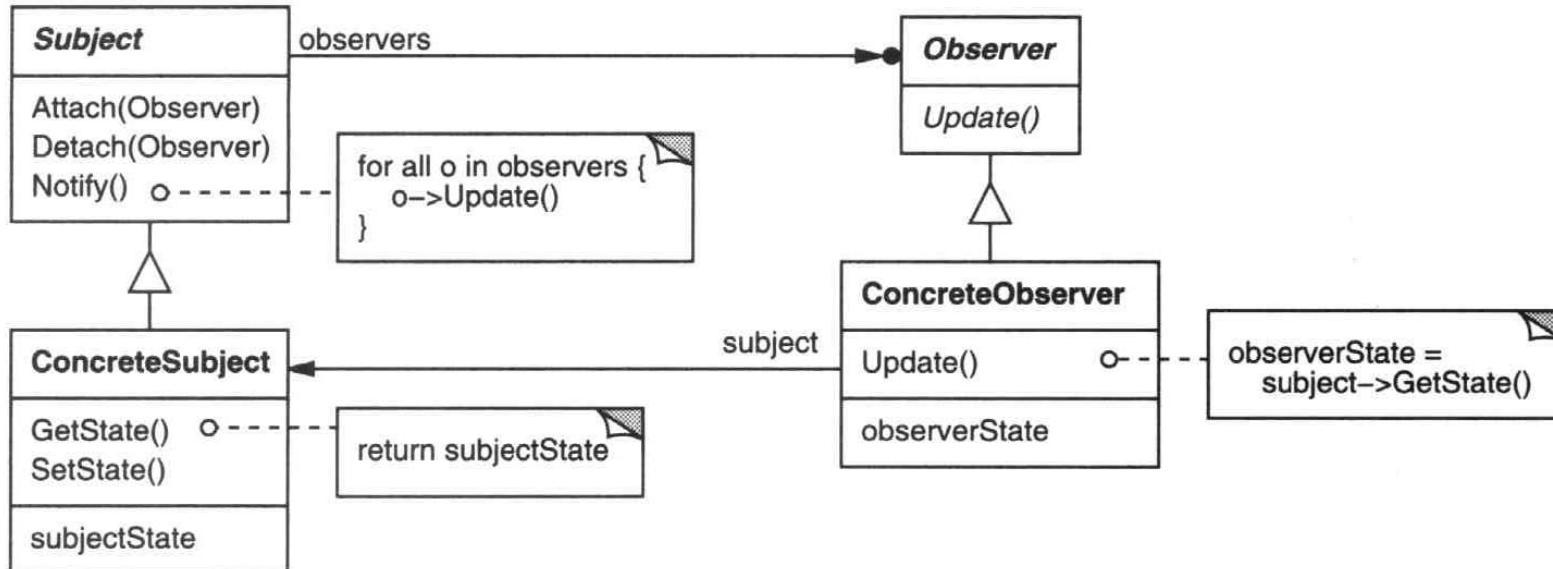
Observer(1/16): UML(1/1)

Observer：將自己註冊到被觀察物件（Subject）中，被觀察對象將觀察者存放在一容器（Container）裡。

Subject：被觀察物件發生了某種變化，從容器中得到所有註冊過的觀察者，將變化通知觀察者。

ConcreteSubject：將有關狀態存入各個Concrete Observer對象。當它的狀態發生改變時，向它的各個觀察者發出通知

Concrete Observer：存儲有關狀態，這些狀態應與目標的狀態保持一致。實現Observer的更新介面以使自身狀態與目標的狀態保持一致。



Observer(2/16): 模式分析(1/1)

優點：

- 觀察者模式在被觀察者和觀察者之間建立一個抽象的耦合。被觀察者角色所知道的只是一個具體觀察者集合，每一個具體觀察者都符合一個抽象觀察者的介面。被觀察者並不認識任何一個具體觀察者，它只知道它們都有一個共同的介面。由於被觀察者和觀察者沒有緊密地耦合在一起，因此它們可以屬於不同的抽象化層次
- 觀察者模式支援廣播通信。被觀察者會向所有的登記過的觀察者發出通知

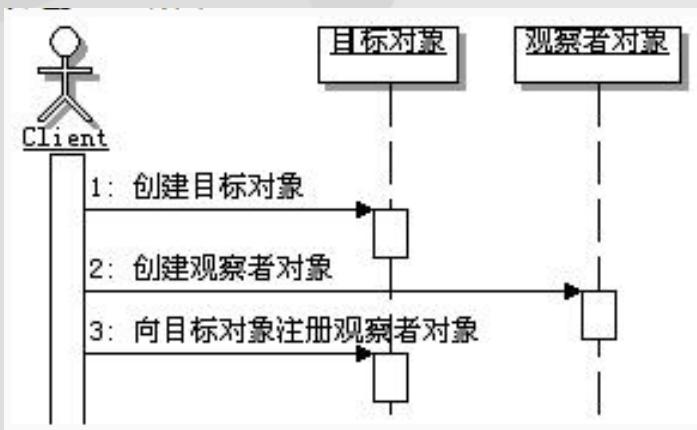
缺點：

- 如果一個被觀察者物件有很多直接和間接的觀察者的話，將所有的觀察者都通知到會花費很多時間
- 如果在被觀察者之間有迴圈依賴的話，被觀察者會觸發它們之間進行迴圈調用，導致系統崩潰。在使用觀察者模式時要特別注意這一點
- 雖然觀察者模式可以隨時使觀察者知道所觀察的物件發生了變化，但是觀察者模式沒有相應的機制使觀察者知道所觀察的物件是怎麼發生變化的

使用時機：

- 當一個抽象模型有兩個方面，其中一個方面依賴於另一方面。將這二者封裝在獨立的物件中以使它們可以各自獨立地改變和複用
- 當對一個物件的改變需要同時改變其它物件，而不知道具體有多少物件有待改變
- 當一個物件必須通知其它物件，而它又不能假定其它物件是誰。換言之，你不希望這些物件是緊密耦合的

Observer(3/16): 模式說明(1/1)



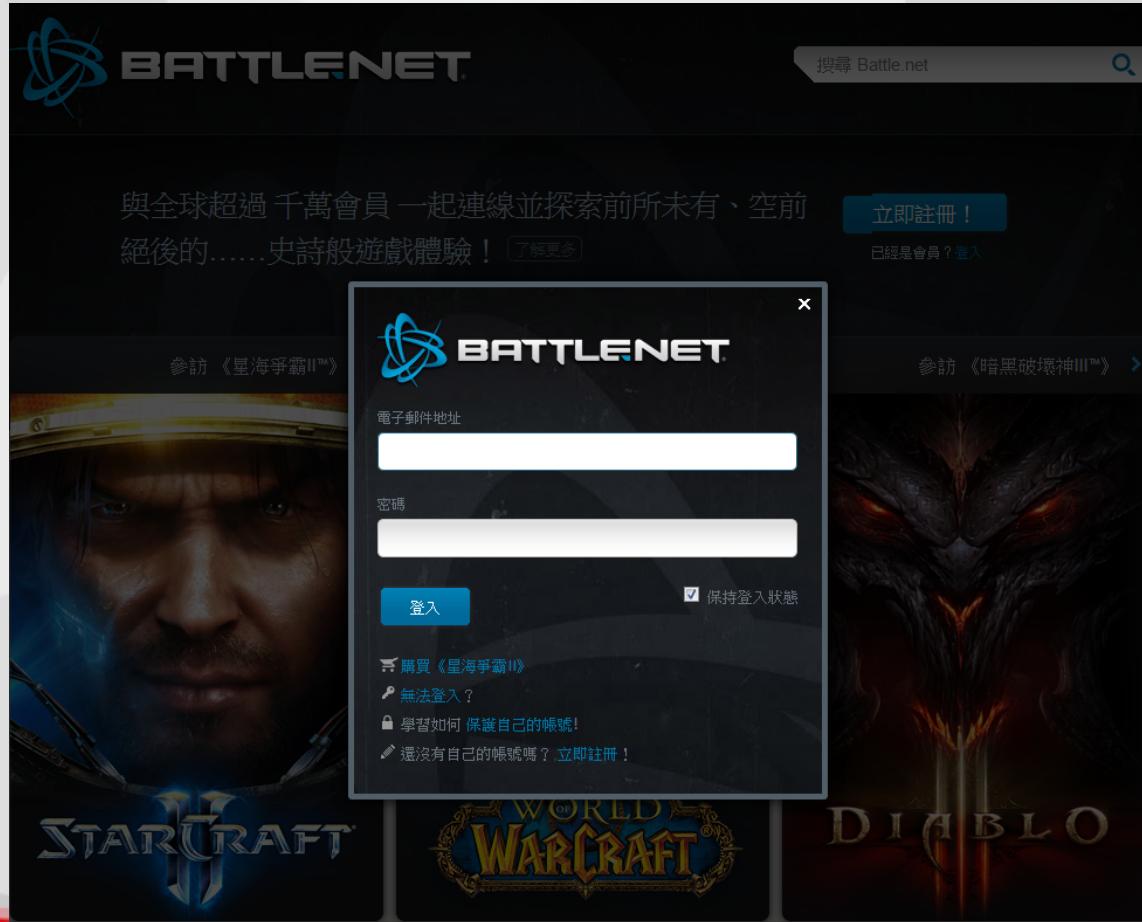
觀察者模式準備階段示意圖



觀察者模式運行階段示意圖

Observer(4/16):範例 (1/10)

假設我們現在要幫Blizzard公司做Battle.net的用戶登入系統..



Hiiir

Observer(5/16):範例 (2/10)

我們很快就把用戶登入系統的類別寫完了..但是突然PM跑來說，要將用戶的IP記錄到log中，並且在用戶登入失敗時，發一封郵件給管理員...

```
class Login
{
    const LOGIN_USER_UNKNOW = 1;
    const LOGIN_WRONG_PASS = 2;
    const LOGIN_ACCESS = 3;
    const LOGIN_ERROR37 = 4;

    private $status = array();

    public function handleLogin($pass, $user, $ip) {
        switch(rand(1,4)){
            case 1:
                $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
                $ret = true;
                break;

            case 2:
                $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
                $ret = false;
                break;

            case 3:
                $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
                $ret = false;
                break;

            case 4:
                $this->setStatus(self::LOGIN_ERROR37, $user, $ip);
                $ret = false;
                break;
        }
        return $ret;
    }
}
```



Observer(6/16):範例 (3/10)

```
class Login
{
    const LOGIN_USER_UNKNOW = 1;
    const LOGIN_WRONG_PASS = 2;
    const LOGIN_ACCESS = 3;
    const LOGIN_ERROR37 = 4;

    private $status = array();

    public function handleLogin($pass, $user, $ip) {
        switch(rand(1,4)){
            case 1:
                $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
                $ret = true;
                break;

            case 2:
                $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
                $ret = false;
                break;

            case 3:
                $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
                $ret = false;
                break;

            case 4:
                $this->setStatus(self::LOGIN_ERROR37, $user, $ip);
                $ret = false;
                break;
        }
        return $ret;
    }

    // 如果客戶要求你將用戶登入的IP地址記到LOG中..
    Logger::logIP($user, $ip, $this->getStatus());

    // 如果客戶要求每次登入發生錯誤時，都要寄一封Email到管理員信箱中..
    if (!$ret) {
        Notifier::mailWarning($user, $ip, $this->getStatus());
    }
}

private function setStatus($status, $user , $ip) {
    $this->status = array($status, $user, $ip);
}

public function getStatus() {
    return $this->status;
}
```

```
// log紀錄類別
class Logger
{
    public static function logIP($user, $ip, $status) {
        echo "log user IP address<br/>";
    }
}

// 郵件發送類別
class Notifier
{
    public static function mailWarning($user, $ip, $status) {
        echo "mail Warning to admin<br/>";
    }
}
```

直接硬幹把代碼加到功能滿足需求，
但是PM過幾天又說要陸續加入其他功能時...



Observer(7/16):範例 (4/10)

硬幹到底哪裡不好...?



直接在代碼中加入功能，會破壞設計。Login類別很快會緊緊嵌入這個特殊的系統中，如果要把Login類單獨提取出來使用，就要將舊系統的特殊代碼移除，因此而走上剪切黏貼代碼的開發道路....

Observer(8/16):範例 (5/10)

讓我們開始用Observer吧!

```
interface Observer  
{  
    public function update(Subject $subject);  
}  
  
class SecurityMonitor implements Observer  
{  
    public function update(Subject $subject) {  
        $status = $subject->getStatus(); // 缺點，不能確定調用的對象是一個Login對象  
        if($status[0] == Login::LOGIN_WRONG_PASS){  
            // 發送郵件給系統管理員  
            echo __CLASS__." sending mail to sysadmin";  
        }  
    }  
}
```

Observer(9/16):範例 (6/10)

```
interface Subject {  
    public function attach(Observer $observer);  
    public function detach(Observer $observer);  
    public function notify();  
}
```

```
class Login implements Subject {
```

```
    const LOGIN_USER_UNKNOWN = 1;    }  
    const LOGIN_WRONG_PASS = 2;  
    const LOGIN_ACCESS = 3;  
    const LOGIN_ERROR37 = 4;
```

```
    private $status = array();  
    private $observers;
```

```
    public function __construct() {  
        $this->observers = array();  
    }
```

```
    public function handleLogin($pass, $user, $ip) {  
        switch(rand(1,4)) {  
            case 1:  
                $this->setStatus(self::LOGIN_ACCESS, $user, $ip);  
                $ret = true;  
                break;  
  
            case 2:  
                $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);  
                $ret = false;  
                break;  
  
            case 3:  
                $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);  
                $ret = false;  
                break;  
  
            case 4:  
                $this->setStatus(self::LOGIN_ERROR37, $user, $ip);  
                $ret = false;  
                break;  
        }  
        $this->notify();  
        return $ret;  
    }
```

Subject

ConcreteSubject

```
private function setStatus($status, $user, $ip) {  
    $this->status = array($status, $user, $ip);  
}  
  
public function getStatus() {  
    return $this->status;  
}  
  
public function attach(Observer $observer){ // 註冊觀察者  
    $this->observers[] = $observer;  
}  
  
public function detach(Observer $observer){ // 解除註冊特定觀察者  
    $newobservers = array();  
    foreach ($this->observers as $obs) {  
        if ($obs !== $observer) {  
            $newobservers[] = $obs;  
        }  
    }  
    $this->observers = $newobservers;  
}  
  
public function notify(){ // 發出通知  
    foreach ($this->observers as $obs) {  
        $obs->update($this);  
    }  
}
```

Observer(10/16):範例 (7/10)

```
$login = new Login(); // 建立登入類別  
$login->attach(new SecurityMonitor()); // 將安全監控類別註冊  
$login->handleLogin("", "hank", "192.168.1.101");
```



SecurityMonitor: sending mail to sysadmin

Observer(11/16):範例 (8/10)

發現問題了嗎? SecurityMonitor類別會使用Login的getStatus()方法，但是Observer使用的物件類別是Subject，所以我們沒辦法保證使用的物件類別是Login，那要怎麼改進呢?

```
interface Observer
{
    public function update(Subject $subject);
}

class SecurityMonitor implements Observer
{
    public function update(Subject $subject) {
        $status = $subject->getStatus(); // 缺點，不能確定調用的對象是一個Login對象
        if($status[0] == Login::LOGIN_WRONG_PASS){
            // 發送郵件給系統管理員
            echo __CLASS__." sending mail to sysadmin";
        }
    }
}
```

Observer(12/16):範例 (9/10)

```
interface Observer
{
    public function update(Subject $subject);
}

// 為了保持Observable接口的通用性，由Observer類別負責保證它們的主體是正確的類型
abstract class LoginObserver implements Observer
{
    private $login;

    public function __construct(Login $login) {
        $this->login = $login;
        $login->attach($this);
    }

    public function update(Subject $subject) {
        if ($subject === $this->login) {
            $this->doUpdate($subject);
        }
    }

    abstract function doUpdate(Login $login);
}
```

```
class SecurityMonitor extends LoginObserver
{
    public function doUpdate(Login $login) {
        $status = $login->getStatus();
        if ($status[0] == Login::LOGIN_WRONG_PASS) {
            // 發送郵件給系統管理員
            echo __CLASS__ . ": sending mail to sysadmin <br>";
        }
    }
}

class GeneralLogger extends LoginObserver
{
    public function doUpdate(Login $login) {
        $status = $login->getStatus();
        // 紀錄登入log
        echo __CLASS__ . ":add login data to log <br>";
    }
}

class PartnershipTool extends LoginObserver
{
    public function doUpdate(Login $login) {
        $status = $login->getStatus();
        echo __CLASS__ . ":set cookie if IP matches a list <br>";
    }
}
```

多一個LoginObserver，並且讓ConcreteObserver繼承LoginObserver

Observer(13/16):範例 (10/10)

```
$login = new Login();
new SecurityMonitor($login);
new GeneralLogger($login);
new PartnershipTool($login);
$login->handleLogin("", "hank", "192.168.1.101");
```



SecurityMonitor: sending mail to sysadmin
GeneralLogger:add login data to log
PartnershipTool:set cookie if IP matches a list

Observer(14/16):範例 (10/10)

另外在PHP5.3以後，我們可以用PHP內建的SPL(Standard PHP Library, PHP標準類別庫)來實作觀察者模式

```
class Login implements SplSubject
{
    const LOGIN_USER_UNKNOWN = 1;
    const LOGIN_WRONG_PASS = 2;
    const LOGIN_ACCESS = 3;
    const LOGIN_ERROR37 = 4;

    private $status = array();
    private $storage;

    public function __construct() {
        $this->storage = new SplObjectStorage(); // PHP >= 5.3
    }

    public function handleLogin($pass, $user, $ip) {
        switch(rand(1,4)){
            case 1:
                $this->setStatus(self::LOGIN_ACCESS, $user, $ip);
                $ret = true;
                break;

            case 2:
                $this->setStatus(self::LOGIN_WRONG_PASS, $user, $ip);
                $ret = false;
                break;

            case 3:
                $this->setStatus(self::LOGIN_USER_UNKNOWN, $user, $ip);
                $ret = false;
                break;

            case 4:
                $this->setStatus(self::LOGIN_ERROR37, $user, $ip);
                $ret = false;
                break;
        }
        $this->notify();
        return $ret;
    }
}
```

```
private function setStatus($status, $user , $ip) {
    $this->status = array($status, $user, $ip);
}

public function getStatus() {
    return $this->status;
}

public function attach(SplObserver $observer) {
    $this->storage->attach($observer);
}

public function detach(SplObserver $observer) {
    $this->storage->detach($observer);
}

public function notify() {
    foreach($this->storage as $obs){
        $obs->update($this);
    }
}
```

Observer(15/16):範例 (10/10)

另外在PHP5.3以後，我們可以用PHP內建的SPL(Standard PHP Library, PHP標準類別庫)來實作觀察者模式

```
// 為了保持Observable接口的通用性，由Observer類別負責保證它們的主體是正確的類型
abstract class LoginObserver implements SplObserver
{
    private $login;
    public function __construct(Login $login) {
        $this->login = $login;
        $login->attach($this);
    }

    public function update(SplSubject $subject) {
        if($subject === $this->login){
            $this->doUpdate($subject);
        }
    }

    abstract function doUpdate(Login $login);
}

class SecurityMonitor extends LoginObserver
{
    public function doUpdate(Login $login) {
        $status = $login->getStatus();
        if ($status[0] == Login::LOGIN_WRONG_PASS) {
            // 發送郵件給系統管理員
            echo __CLASS__." sending mail to sysadmin <br/>";
        }
    }
}
```

```
class GeneralLogger extends LoginObserver
{
    public function doUpdate(Login $login) {
        $status = $login->getStatus();
        // 紀錄登入log
        echo __CLASS__." add login data to log <br/>";
    }
}

class PartnershipTool extends LoginObserver
{
    public function doUpdate(Login $login) {
        $status = $login->getStatus();
        echo __CLASS__." set cookie if IP matches a list <br/>";
    }
}
```

Observer(16/16):Observer vs. Mediator

Observer：核心思想在於訂閱-發布。Observer模式通過引入Observer與Subject來達到將通信分散化的目的。

Mediator：核心思想是使本來一群關係錯綜複雜的群體，通過引入一個中間人，把原來成網狀的關係梳理成星形結構。
Mediator封裝了物件之間的通信，從而將通信集中在一個Mediator物件中。

State(狀態模式)

Hiiir

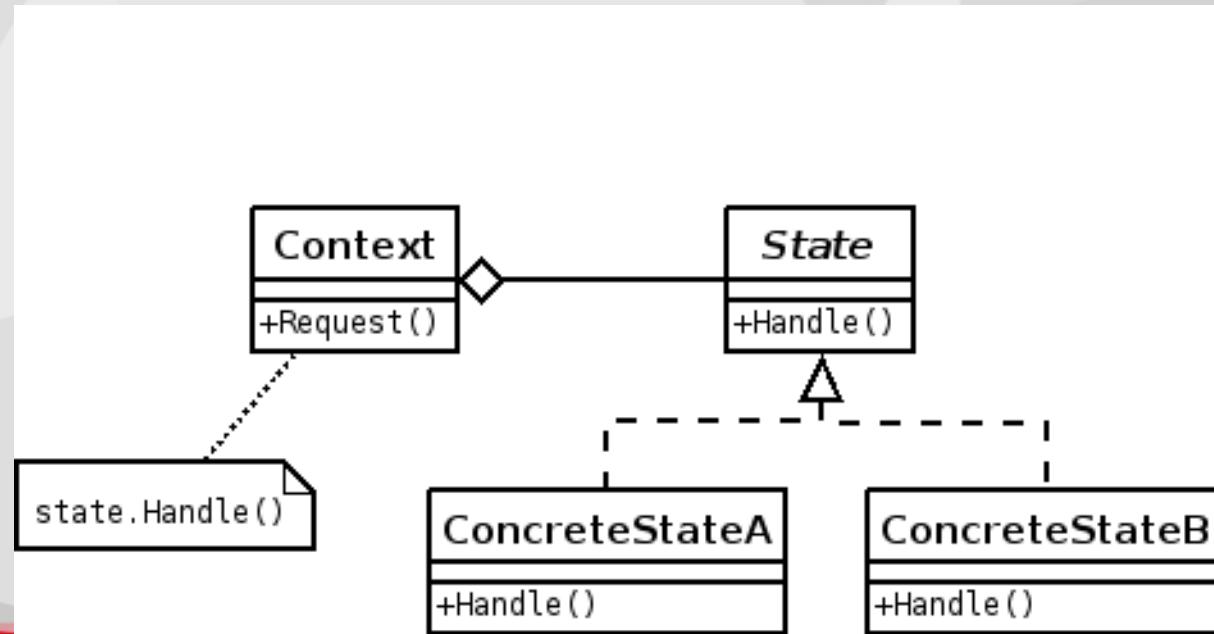
行動×商務 社群×媒體
mobile e-commerce social media

State(1/15): UML(1/1)

State : 定義一個介面，用以封裝物件的一個特定狀態所對應的行為

ConcreteState : 每一個具體狀態類都實現環境 (Context) 的一個狀態所對應的行為

Context : 定義用戶端的介面，並且保留一個具體狀態類的實例。這個具體狀態類的實例給出環境物件的現有狀態



State(2/15): 模式分析(1/1)

優點：

- 避免了為判斷狀態而產生的巨大的if或case語句。 將物件行為交給狀態類維護後，對於上層程式而言，僅需要維護狀態之間的轉換規則

缺點：

- 狀態模式的使用必然會增加系統類別和物件的個數
- 狀態模式的結構與實現都較為複雜，如果使用不當將導致程式結構和代碼的混亂

使用時機：

- 當一個物件的行為取決於它的狀態，並且它必須在運行時根據狀態改變它的行為，或者一個操作中還有龐大的多分支的條件陳述式(if...else, switch)，並且這些分支依賴於該物件的狀態時，就可以使用狀態模式

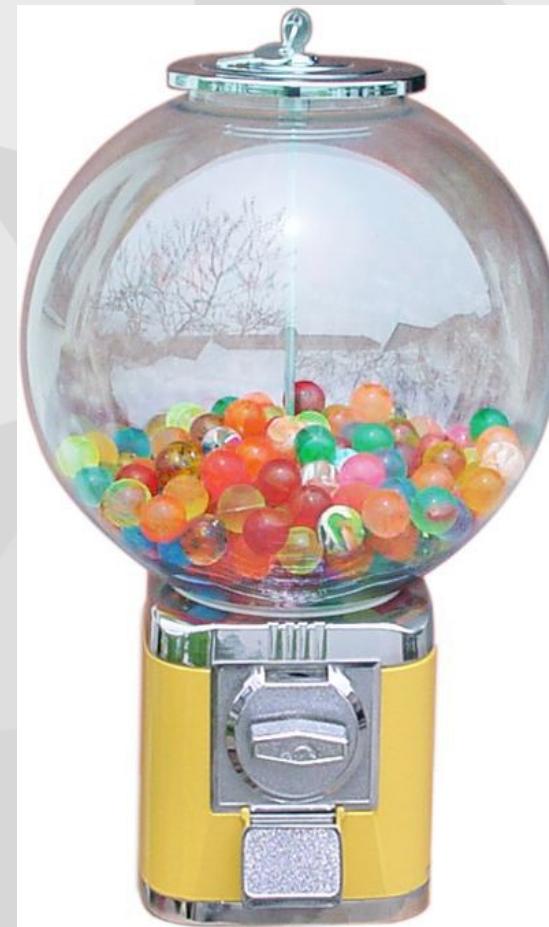
State(3/15): 範例(1/13)

我們現在要設計一個糖果機，它有以下四個狀態：

1. 沒有硬幣狀態
2. 有硬幣狀態
3. 糖果售出狀態
4. 糖果售罄狀態

可執行的動作也有四個，分別為：

1. 插入硬幣
2. 退回硬幣
3. 轉動曲柄
4. 發放糖果



State(4/15): 範例(2/13)

一般我們會這樣寫....

```
class CandyMachine
{
    const STATUS_NO_COIN = 1; // 沒有硬幣狀態
    const STATUS_HAS_COIN = 2; // 有硬幣狀態
    const STATUS SOLD = 3; // 糖果售出狀態
    const STATUS_SOLDOUT = 4; // 糖果售罄狀態

    private $state; // 糖果機目前狀態
    private $count = 0; // 糖果數量

    // 初始化糖果機與糖果數量
    public function __construct($count) {
        $this->count = $count;

        if ($this->count > 0) {
            $this->state = CandyMachine::STATUS_NO_COIN;
        } else {
            $this->state = CandyMachine::STATUS_SOLDOUT;
        }

        echo "初始化糖果機成功，共有" . $count . "顆糖果<br/>";
    }

    // 取得目前剩餘糖果數量
    public function getCount() {
        return $this->count;
    }
}
```

```
public function setState($state){
    $this->state = $state;
}

public function getState(){
    return $this->state;
}

public function releaseCandy() {
    if ( $this->count > 0 ) {
        $this->count = $this->count - 1;
        echo "得到一顆糖果，還有有糖果 {$this->count} 顆<br/>";
    }
}

// 插入硬幣
public function insertCoin() {
    switch($this->state) {
        case CandyMachine::STATUS_NO_COIN:
            $this->setState(CandyMachine::STATUS_HAS_COIN);
            echo "你插入了一枚硬幣<br/>";
            break;

        case CandyMachine::STATUS_HAS_COIN:
            echo "已經有硬幣了，無法插入<br/>";
            break;

        case CandyMachine::STATUS SOLD:
            echo "請稍等，我們正在給你糖果<br/>";
            break;

        case CandyMachine::STATUS_SOLDOUT:
            echo "很抱歉，所有糖果都已售罄，無法插入<br/>";
            break;
    }
}
```

State(5/15): 範例(3/13)

```
// 退回硬幣
public function ejectCoin() {
    switch($this->state) {
        case CandyMachine::STATUS_NO_COIN:
            echo "你還沒插入硬幣呢!<br/>";
            break;

        case CandyMachine::STATUS_HAS_COIN:
            $this->setState(CandyMachine::STATUS_NO_COIN);
            echo "退回硬幣<br/>";
            break;

        case CandyMachine::STATUS SOLD:
            echo "很抱歉，你已經轉動了曲柄，無法退回<br/>";
            break;

        case CandyMachine::STATUS SOLDOUT:
            echo "所有糖果已售罄，無法插入硬幣，故不退幣<br/>";
            break;
    }
}
```

```
// 轉動曲柄
public function turnCrank() {
    switch($this->state){
        case CandyMachine::STATUS_NO_COIN:
            echo "你還沒插入硬幣，無法轉動曲柄<br/>";
            break;

        case CandyMachine::STATUS_HAS_COIN:
            if ($this->count > 0){
                $this->setState(CandyMachine::STATUS SOLD);
                echo "你轉動了曲柄<br/>";
            } else {
                $this->setState(CandyMachine::STATUS SOLDOUT);
                echo "你轉動了曲柄，但是已經沒有糖果了<br/>";
            }
            break;

        case CandyMachine::STATUS SOLD:
            echo "很抱歉，你正在轉動曲柄，所以此命令無效<br/>";
            break;

        case CandyMachine::STATUS SOLDOUT:
            echo "很抱歉，已經沒有糖果了，無法轉動曲柄<br/>";
            break;
    }
}
```

State(6/15): 範例(4/13)

```
// 發放糖果
public function dispense() {
    switch($this->state){
        case CandyMachine::STATUS_NO_COIN:
            echo "你還沒插入硬幣，無法發放糖果<br/>";
            break;

        case CandyMachine::STATUS_HAS_COIN:
            echo "請先轉動曲柄<br/>";
            break;

        case CandyMachine::STATUS SOLD:
            $this->releaseCandy();
            if($this->count > 0)
                $this->setState(CandyMachine::STATUS_NO_COIN);
            else {
                $this->setState(CandyMachine::STATUS SOLDOUT);
                echo "沒有糖果了.\n";
            }
            break;

        case CandyMachine::STATUS SOLDOUT:
            echo "很抱歉，已經沒有糖果了，無法發放<br/>";
            break;
    }
}
```

State(7/15): 範例(5/13)

```
include_once '../../class/pattern/state.php';

$candyMachine = new CandyMachine(4);

echo '<br/>';

$candyMachine->insertCoin();
$candyMachine->insertCoin();
$candyMachine->ejectCoin();
$candyMachine->ejectCoin();
$candyMachine->turnCrank();
$candyMachine->dispense();

echo '<br/>';

$candyMachine->insertCoin();
$candyMachine->turnCrank();
$candyMachine->dispense();

echo '<br/>';

$candyMachine->turnCrank();
$candyMachine->dispense();
```

```
echo '<br/>';

$candyMachine->insertCoin();
$candyMachine->turnCrank();
$candyMachine->ejectCoin();
$candyMachine->dispense();

echo '<br/>';

$candyMachine->insertCoin();
$candyMachine->turnCrank();
$candyMachine->dispense();
$candyMachine->ejectCoin();
```

State(8/15): 範例(6/13)

初始化糖果機成功，共有2顆糖果

你插入了一枚硬幣
已經有硬幣了，無法插入
退回硬幣
沒插入硬幣，無法退回
你還沒插入硬幣，無法轉動曲柄
你還沒插入硬幣，無法發放糖果

你插入了一枚硬幣
你轉動了曲柄
得到一顆糖果，還有糖果 1 顆

你還沒插入硬幣，無法轉動曲柄
你還沒插入硬幣，無法發放糖果

你插入了一枚硬幣
你轉動了曲柄
很抱歉，你已經轉動了曲柄，無法退回
得到一顆糖果，還有糖果 0 顆
沒有糖果了

很抱歉，所有糖果都已售罄，無法插入
很抱歉，已經沒有糖果了，無法轉動曲柄
很抱歉，已經沒有糖果了，無法發放
所有糖果已售罄，無法插入硬幣，故不退幣

State(9/15): 範例(7/13)

如果這時候想要加上有10%機率送出兩顆糖果的功能時...

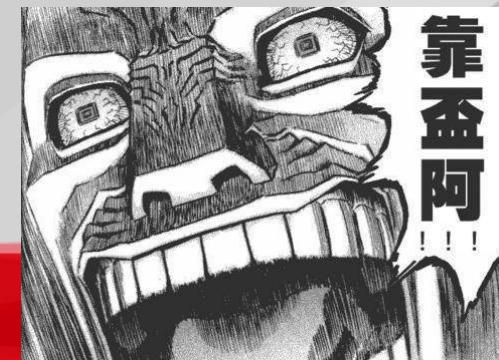
```
class CandyMachine
{
    const STATUS_NO_COIN = 1;    // 沒有硬幣狀態
    const STATUS_HAS_COIN = 2;   // 有硬幣狀態
    const STATUS SOLD = 3;       // 糖果售出狀態
    const STATUS SOLDOUT = 4;    // 糖果售罄狀態

    const STATUS_DOUBLE = 5;      ← 多加一個送兩顆糖果的狀態
}
```

```
public function insertCoin(); // 插入硬幣
public function ejectCoin(); // 退回硬幣
public function turnCrank(); // 轉動曲柄
public function dispense(); // 發放糖果
```

每一個function都要新增加Double狀態的條件判斷....

Hiiir



State_(10/15): 範例_(8/13)

```
class CandyMachine
{
    private $noCoinState;
    private $hasCoinState;
    private $soldState;
    private $soldoutState;
    private $doubleState; // 送兩顆糖果狀態

    private $state;        // 糖果機目前狀態
    private $count = 0;    // 糖果數量

    public function __construct( $count = 0 ) {
        $this->noCoinState = new NoCoinState($this);
        $this->hasCoinState = new HasCoinState($this);
        $this->soldState = new SoldState($this);
        $this->soldoutState = new SoldoutState($this);
        $this->doubleState = new DoubleState($this); // 多初始化送兩顆糖果的狀態
        $this->count = $count;
    }

    if($this->count > 0) {
        $this->state = $this->noCoinState;
    } else {
        $this->state = $this->soldoutState;
    }

    echo "初始化糖果機成功，共有".$count."顆糖果<br/>";
}

// 取得目前剩餘糖果數量
public function getCount() {
    return $this->count;
}

public function setState(State $state) {
    $this->state = $state;
}
```

```
public function getState() {
    return $this->state;
}

public function releaseCandy() {
    if ( $this->count > 0 ) {
        $this->count = $this->count - 1;
        echo "得到一顆糖果，還有糖果 {$this->count} 顆<br/>";
    }
}

public function insertCoin() {
    $this->state->insertCoin();
}

public function ejectCoin() {
    $this->state->ejectCoin();
}

public function turnCrank() {
    $this->state->turnCrank();
}

public function dispense() {
    $this->state->dispense();
}

public function getNoCoinState () {
    return $this->noCoinState;
}

public function getHasCoinState () {
    return $this->hasCoinState;
}

public function getSoldOutState () {
    return $this->soldoutState;
}

public function getSoldState () {
    return $this->soldState;
}

public function getDoubleState() {
    return $this->doubleState;
}
```



State(11/15): 範例(9/13)

```
// 狀態模式  
interface State  
{  
    public function insertCoin(); // 插入硬幣  
    public function ejectCoin(); // 退回硬幣  
    public function turnCrank(); // 轉動曲柄  
    public function dispense(); // 發放糖果  
}
```

State

```
class NoCoinState implements State  
{  
    private $machine;  
  
    public function __construct(CandyMachine $candyMachine) {  
        $this->machine = $candyMachine;  
    }  
  
    public function insertCoin() {  
        $this->machine->setState($this->machine->getHasCoinState());  
        echo "你插入了一枚硬幣<br/>";  
    }  
  
    public function ejectCoin() {  
        echo "沒插入硬幣，無法退回<br/>";  
    }  
  
    public function turnCrank() {  
        echo "你還沒插入硬幣，無法轉動曲柄<br/>";  
    }  
  
    public function dispense() {  
        echo "你還沒插入硬幣，無法發放糖果<br/>";  
    }  
}
```

ConcreteState

State(12/15): 範例(10/13)

```
class HasCoinState implements State
{
    private $machine;

    public static function random() { // 亂數計算是否可以獲得兩顆糖果
        if(rand(1,10) == 1) {
            return true;
        } else {
            return false;
        }
    }

    public function __construct(CandyMachine $candyMachine) {
        $this->machine = $candyMachine;
    }

    public function insertCoin() {
        echo "已經有硬幣了，無法插入<br/>";
    }

    public function ejectCoin() {
        $this->machine->setState($this->machine->getNoCoinState());
        echo "退回硬幣<br/>";
    }

    public function turnCrank() {
        if(self::random() == true) {
            $this->machine->setState($this->machine->getSoldState());
        } else {
            $this->machine->setState($this->machine->getDoubleState()); // 變為兩顆糖果狀態
        }
        echo "你轉動了曲柄<br/>";
    }

    public function dispense() {
        echo "你還沒轉動曲柄，無法發放糖果<br/>";
    }
}
```

```
class SoldState implements State
{
    private $machine;

    public function __construct(CandyMachine $candyMachine) {
        $this->machine = $candyMachine;
    }

    public function insertCoin() {
        echo "請稍等，我們正在給你糖果<br/>";
    }

    public function ejectCoin() {
        echo "很抱歉，你已經轉動了曲柄，無法退回<br/>";
    }

    public function turnCrank() {
        echo "很抱歉，你已經轉動曲柄，不能再轉了<br/>";
    }

    public function dispense() {
        $this->machine->releaseCandy();
        if ( $this->machine->getCount() > 0 ) {
            $this->machine->setState($this->machine->getNoCoinState());
        } else {
            echo "沒有糖果了<br/>";
            $this->machine->setState($this->machine->getSoldOutState());
        }
    }
}
```



State(13/15): 範例(11/13)

```
class SoldoutState implements State
{
    private $machine;

    public function __construct(CandyMachine $candyMachine) {
        $this->machine = $candyMachine;
    }

    public function insertCoin() {
        echo "很抱歉，所有糖果都已售罄，無法插入<br/>";
    }

    public function ejectCoin() {
        echo "所有糖果已售罄，無法插入硬幣，故不退幣<br/>";
    }

    public function turnCrank() {
        echo "很抱歉，已經沒有糖果了，無法轉動曲柄<br/>";
    }

    public function dispense() {
        echo "很抱歉，已經沒有糖果了，無法發放<br/>";
    }
}
```

ConcreteState

State(14/15): 範例(12/13)

加上有10%機率送出兩顆糖果的功能...

```
// 兩顆糖果狀態
class DoubleState implements State
{
    private $machine;

    public function __construct(CandyMachine $candyMachine) {
        $this->machine = $candyMachine;
    }

    public function insertCoin() {
        echo "請稍等，我們正在給你糖果<br/>";
    }

    public function ejectCoin() {
        echo "很抱歉，你已經轉動了曲柄，無法退回<br/>";
    }

    public function turnCrank() {
        echo "很抱歉，你已經轉動曲柄，不能再轉了<br/>";
    }
}
```

```
public function dispense() {
    $this->machine->releaseCandy();
    if ( $this->machine->getCount() > 0 ) {

        echo "免費贈送一顆糖果<br/>";
        $this->machine->releaseCandy(); // 如果還有糖果，再發一顆

        if ( $this->machine->getCount() > 0 )
            $this->machine->setState( $this->machine->getNoCoinState() );
        else {
            echo "免費贈送一顆糖果，但是機器裡面沒有糖果了<br/>";
            $this->machine->setState( $this->machine->getSoldOutState() );
        }
    } else {
        echo "沒有糖果了<br/>";
        $this->machine->setState( $this->machine->getSoldOutState() );
    }
}
```

State(15/15):範例 (13/13)

初始化糖果機成功，共有4顆糖果

你插入了一枚硬幣
已經有硬幣了，無法插入
退回硬幣
沒插入硬幣，無法退回
你還沒插入硬幣，無法轉動曲柄
你還沒插入硬幣，無法發放糖果

你插入了一枚硬幣
你轉動了曲柄
得到一顆糖果，還有糖果 3 顆
免費贈送一顆糖果
得到一顆糖果，還有糖果 2 顆

你還沒插入硬幣，無法轉動曲柄
你還沒插入硬幣，無法發放糖果

你插入了一枚硬幣
你轉動了曲柄
很抱歉，你已經轉動了曲柄，無法退回
得到一顆糖果，還有糖果 1 顆
免費贈送一顆糖果
得到一顆糖果，還有糖果 0 顆
免費贈送一顆糖果，但是機器裡面沒有糖果了

很抱歉，所有糖果都已售罄，無法插入
很抱歉，已經沒有糖果了，無法轉動曲柄
很抱歉，已經沒有糖果了，無法發放
所有糖果已售罄，無法插入硬幣，故不退幣

Strategy(策略模式)

Hiiir

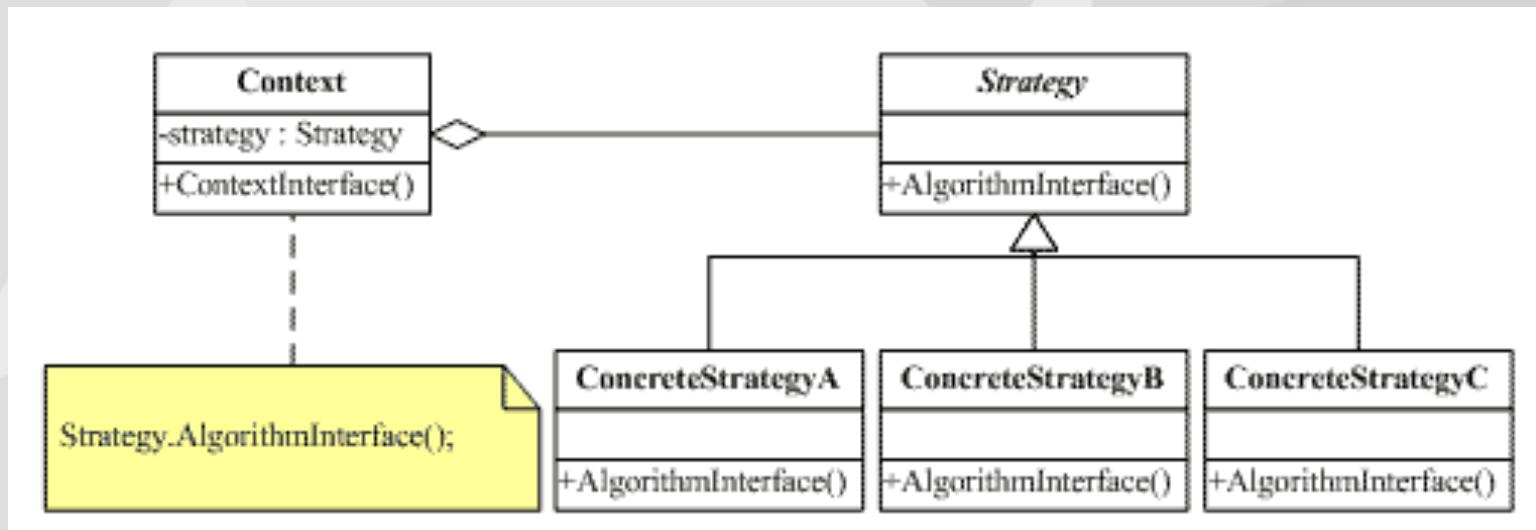
行動×商務 社群×媒體
mobile e-commerce social media

Strategy(1/9): UML(1/1)

Context : 持有一個Strategy類別的物件

Strategy : 通常由一個介面或者抽象類別實現

ConcreteStrategy : 包裝了相關的演算法和行為



Strategy(2/9)：模式分析(1/1)

優點：

- 多個類別只區別在表現行為不同，可以使用**Strategy**模式，在運行時動態選擇具體要執行的行為
- 需要在不同情況下使用不同的策略(演算法)，或者策略還可能在未來用其它方式來實現
- 對客戶隱藏具體策略(演算法)的實現細節，彼此完全獨立

缺點：

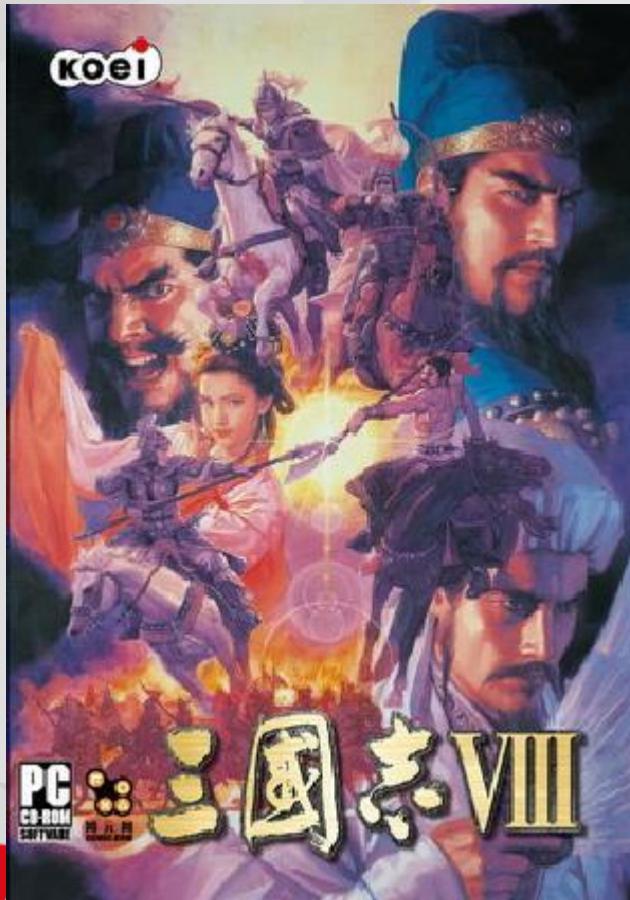
- 因為每個具體策略類別都會產生一個新類別，所以會增加系統需要維護的類別數量

使用時機：

- 多個類別只區別在表現行為不同，可以使用**Strategy**模式，在運行時動態選擇具體要執行的行為
- 需要在不同情況下使用不同的策略(演算法)，或者策略還可能在未來用其它方式來實現
- 當一個物件必須通知其它物件，而它又不能假定其它物件是誰。換言之，你不希望這些物件是緊密耦合的

Strategy(3/9):範例 (1/4)

假設我們現在要做一款偽・三國志，裡面的文官武將可以使用不同的法術或兵法，要怎麼做呢？



Hill

火牛陣

嘲弄術

Strategy(4/9):範例 (2/4)

```
// 武將
class Officer {
    private $Name;
    private $Mp;      // 技能點數
    private $Strategy; // 兵法

    public function __construct($name, $mp, Strategy $strategy) {
        $this->Name = $name;
        $this->Mp = $mp;
        $this->Strategy = $strategy;
    }

    // 使用兵法
    public function useStrategy() {
        return $this->Name . " 使用了" . $this->Strategy->StrategyName() . ", " . $this->Strategy->useStrategy($this);
    }

    public function setStrategy(Strategy $strategy) {
        $this->Strategy = $strategy;
    }

    public function getMp(){}
        return $this->Mp;
    }

    public function setMp($mp){
        $this->Mp = $mp;
    }
}
```



Strategy(5/9): 範例 (3/4)

```
abstract class Strategy
{
    abstract function useStrategy(Officer $officer);
    abstract function StrategyName();
}

// 火牛陣
class FireBullStrategy extends Strategy ConcreteStrategy
{
    public function useStrategy(Officer $officer) {
        $mp = $officer->getMp() - 10;
        $officer->setMp($mp);
        return '敵方受到 30 點傷害，並且造成混亂，還剩 ' . $mp . ' 點MP';
    }

    public function StrategyName() {
        return "火牛陣";
    }
}

// 嘲弄術
class ScoffStrategy extends Strategy {
    public function useStrategy(Officer $officer) {
        $mp = $officer->getMp() - 5;
        $officer->setMp($mp);
        return '敵方士氣下降 15 點，還剩 ' . $mp . ' 點MP';
    }

    public function StrategyName() {
        return "嘲弄術";
    }
}
```

Strategy(6/9):範例 (4/4)

```
include_once '../class/pattern/strategy.php';

$officer = new Officer("諸葛亮", 100, new FireBullStrategy());
echo $officer->useStrategy();

echo '<br/>';

$officer->setStrategy(new ScoffStrategy());
echo $officer->useStrategy();
```

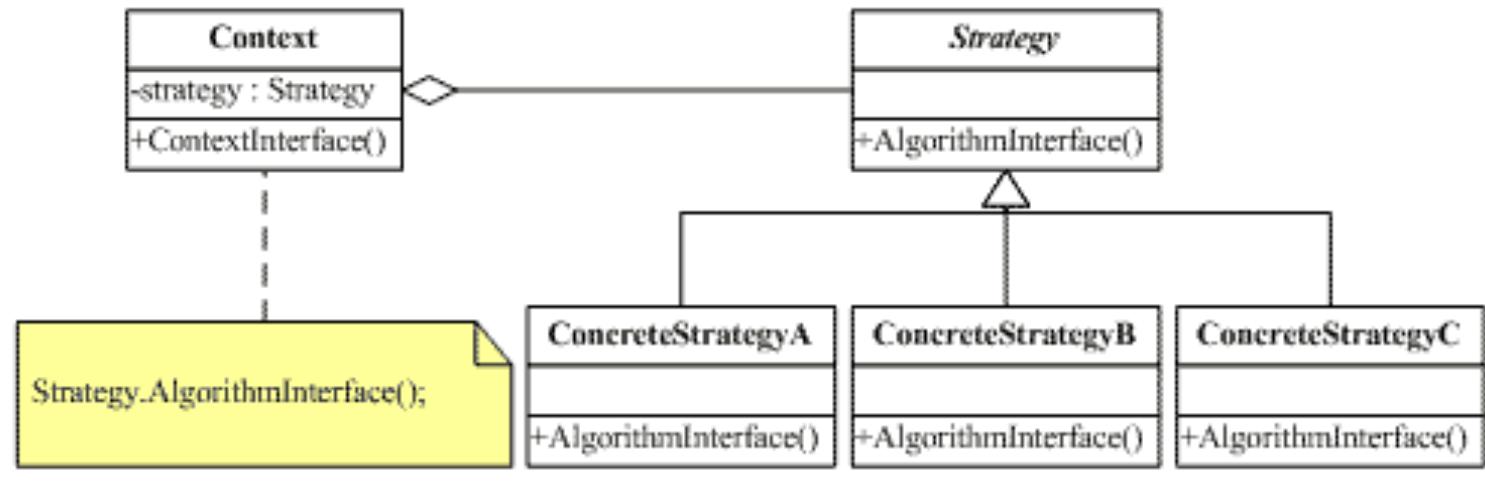


諸葛亮 使用了火牛陣，敵方受到 30 點傷害，並且造成混亂，還剩 90 點MP

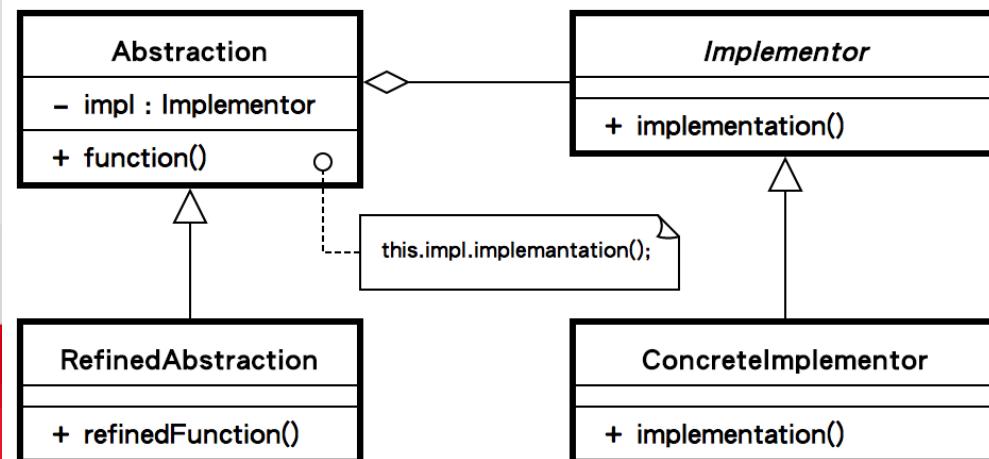
諸葛亮 使用了嘲弄術，敵方士氣下降 15 點，還剩 85 點MP

Strategy(7/9): Strategy vs. Bridge(1/3)

Strategy

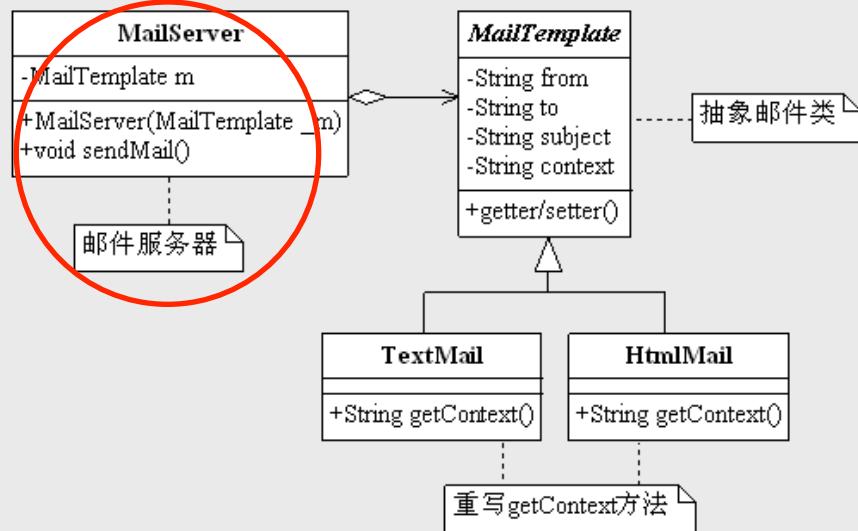


Bridge

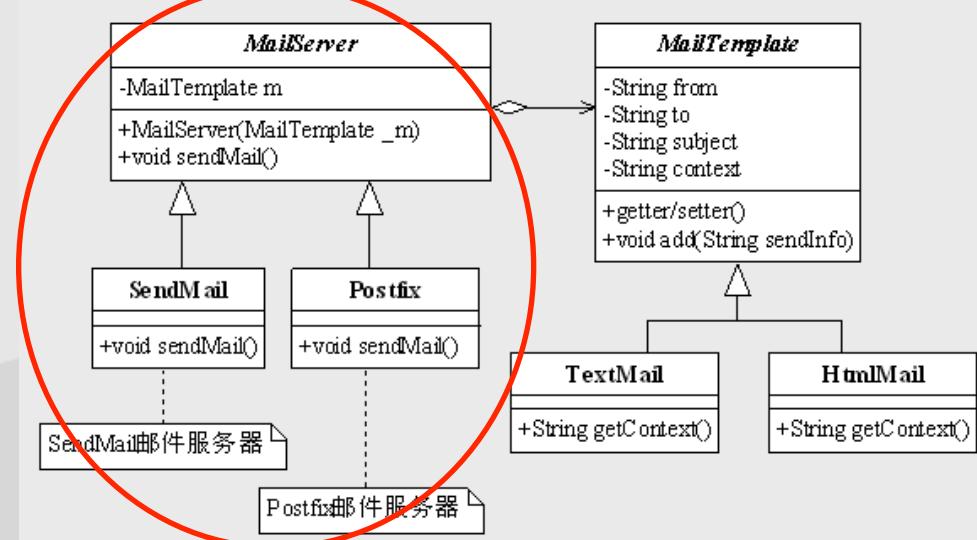


Strategy(8/9): Strategy vs. Bridge(2/3)

Strategy



Bridge



Strategy(9/9): Strategy vs. Bridge(3/3)

策略模式和橋樑模式是如此相似，我們只能從它們的意圖上來分析，

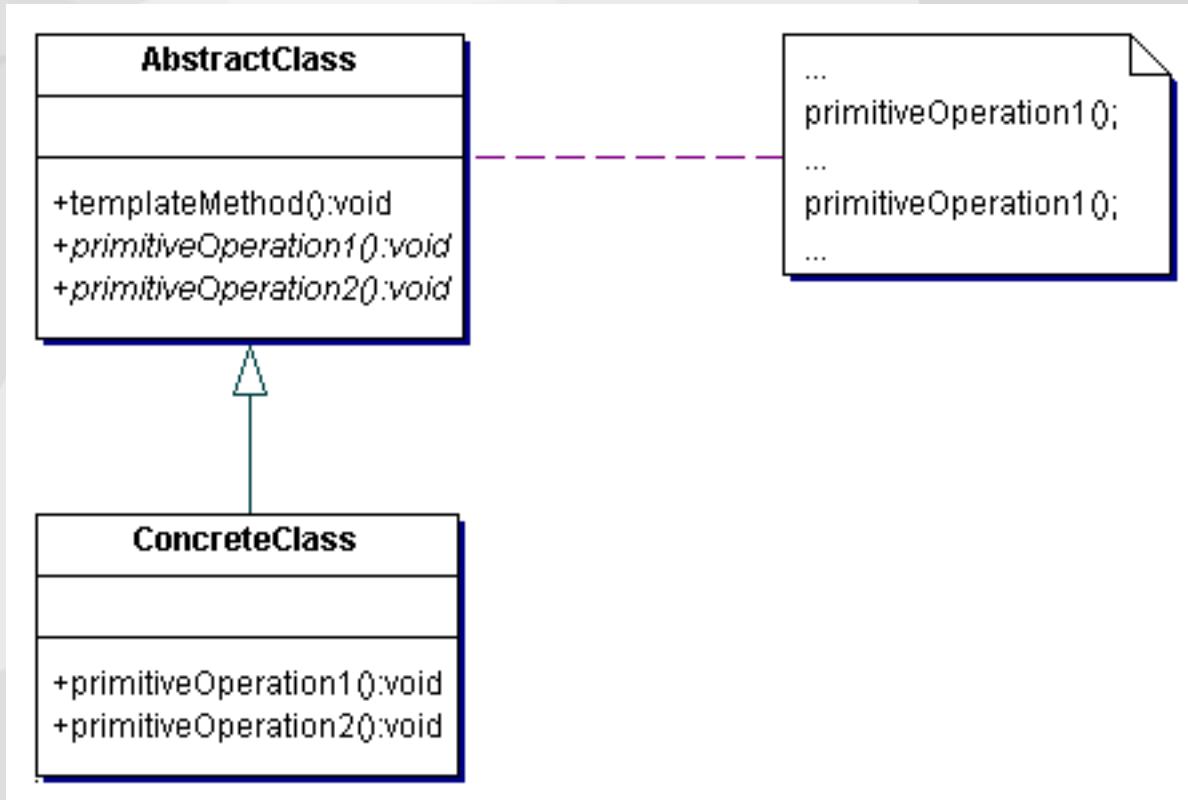
Strategy是一個行為模式，旨在封裝一系列的行為，在例子中我們認為把郵件的必要資訊（寄件者、收件人、標題、內容）封裝成一個物件就是一個行為，封裝的格式（演算法）不同，行為也就不同。

Bridge是一個結構模式，旨在解決在不破壞封裝的情況下，如何抽取出它的抽象部分和實現部分。它的前提是不破壞封裝，讓抽象部分和實現部分都可以獨立地變化，在例子中，我們的郵件伺服器和郵件模版是不是都可以獨立地變化？不管是郵件伺服器還是郵件範本，只要繼承了抽象類別就可以繼續擴展。

精簡地來說，策略模式是使用繼承和多態建立一套可以自由切換演算法的模式，橋樑模式是在不破壞封裝的前提下解決抽象和實現都可以獨立擴展的模式。

Template Method(樣板方法模式)

Template Method(1/14): UML(1/1)



Template Method_(2/14): 模式分析(1/1)

優點：

- 定義了一個演算法的步驟，並允許子類別為一個或多個步驟，提供其實踐方式。

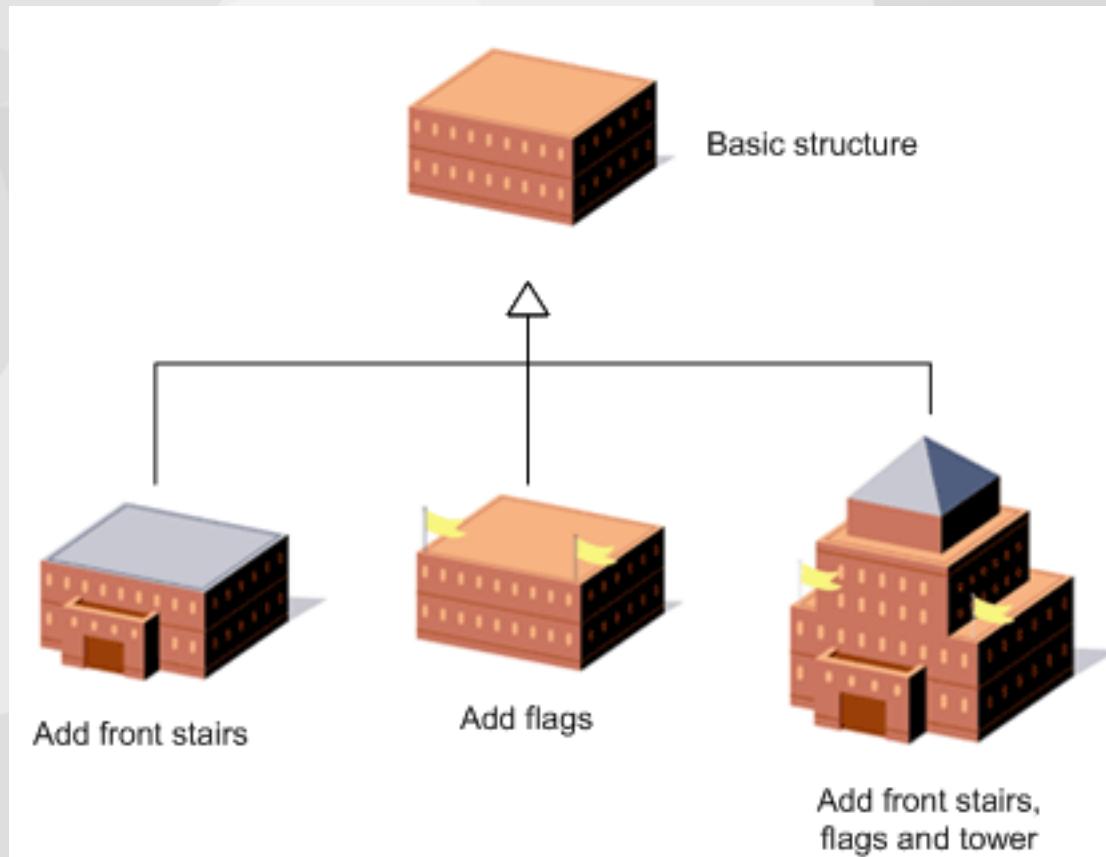
缺點：

- 每個不同的行為都要定義一個子類別，導致類別的數量增加
- 如果演算法內的步驟切得太細，越多子類別實踐部分就會越大費周章
- 如果演算法內的步驟太少，將會導致失去較多的彈性

使用時機：

- 需要定義了一個演算法的步驟，並允許次類別為一個或多個步驟提供其實踐方式。讓次類別在不改變演算法架構的情況下，重新定義演算法中的某些步驟。

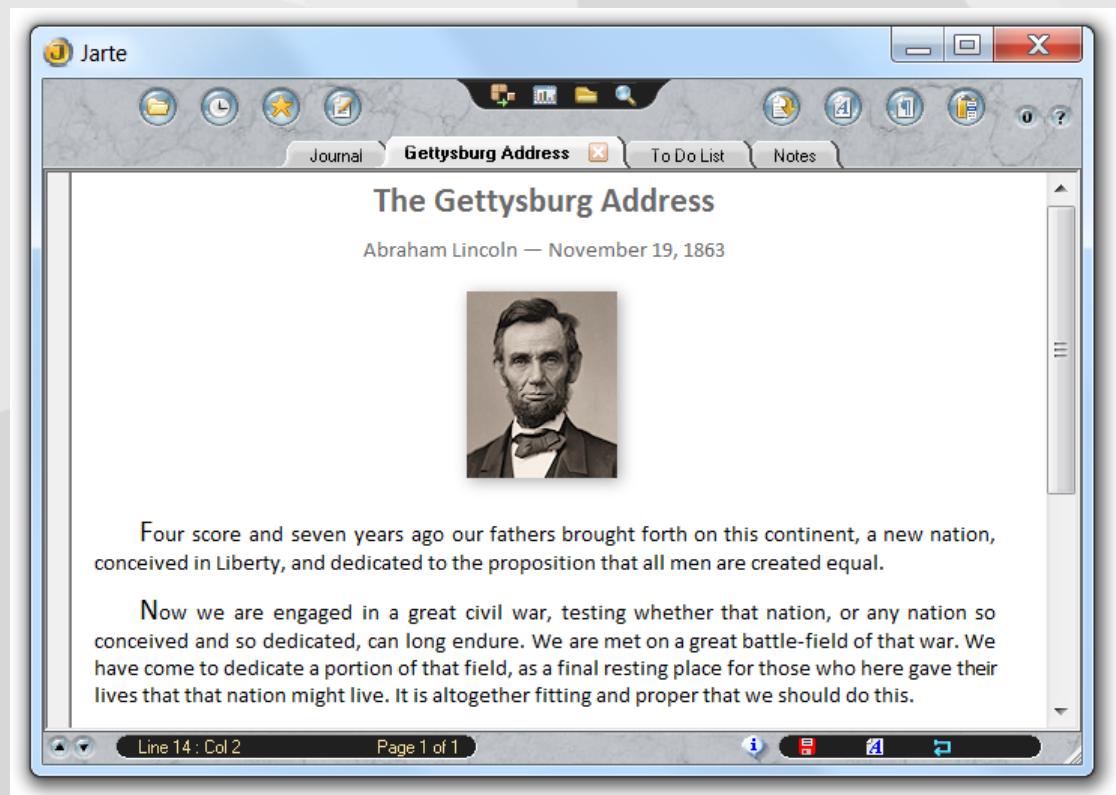
Template Method_(3/14): 模式說明(1/1)



Template Method(4/14): 範例1(1/2)

今天我們需要做一個可以讀不同格式的文章閱讀器，我們發現
讀取不同格式文件都採用相同的演算法：

1. 開啓檔案
2. 讀取檔案
3. 顯示文件內容
4. 關閉檔案



Template Method(5/14): 範例1(2/2)

```
abstract class TextReader
{
    // templateMethod()
    public function readFileProcess() {
        $this->openFile();
        $this->readFile();
        $this->showFile();
        $this->closeFile();
    }

    // 共同的實現部分可以放在父類別，避免程式碼重複
    protected function openFile() {
        echo "開啟檔案<br>";
    }

    protected abstract function readFile(); // primitiveOperation()
    protected abstract function showFile(); // primitiveOperation()
    protected abstract function closeFile(); // primitiveOperation()
}
```

```
class WordTextReader extends TextReader
{
    protected function readFile() {
        echo "讀取Word .doc格式檔案<br>";
    }

    protected function showFile() {
        echo "顯示Word .doc格式檔案<br>";
    }

    protected function closeFile() {
        echo "關閉Word .doc格式檔案<br>";
    }
}

class AdobeTextReader extends TextReader
{
    protected function readFile() {
        echo "讀取Adobe .pdf格式檔案<br>";
    }

    protected function showFile() {
        echo "顯示Adobe .pdf格式檔案<br>";
    }

    protected function closeFile() {
        echo "顯示Adobe .pdf格式檔案<br>";
    }
}
```



Template Method(6/14): 範例1(3/7)

```
$wordTextReader = new WordTextReader(); //建立word文字閱讀器  
$adobeTextReader = new AdobeTextReader(); //建立adobe文字閱讀器  
  
$wordTextReader->readFileProcess();  
echo "<br>";  
  
$adobeTextReader->readFileProcess();
```



開啟檔案
讀取Word .doc格式檔案
顯示Word .doc格式檔案
關閉Word .doc格式檔案

開啟檔案
讀取Adobe .pdf格式檔案
顯示Adobe .pdf格式檔案
顯示Adobe .pdf格式檔案

Template Method(7/14):範例2:Hook(1/4)

- Hook是一個方法，被宣告在抽象類別中，預設什麼都不做，或是有預設的實踐內容。
- Hook可以讓子類別實踐演算法中選擇性的部分，如果 Hook對於子類別的實踐並不重要時，子類別可以對Hook置之不理。
- Hook另一使用時機，是子類別能有機會對於演算法中某個步驟作出反應（例如跳出選擇視窗後，讓使用者選擇是否要使用演算法中的某個步驟）。

Template Method(8/14):範例2:Hook(2/4)

```
abstract class TextReader
{
    // templateMethod()
    public function readfileProcess() {
        $this->openFile();
        $this->readFile();

        // 這邊加上了hook，讓子類別可以決定是不是要在演算法中使用parserFile()
        if ($this->hook() == true) {
            $this->parseFile();
        }

        $this->showFile();
        $this->closeFile();
    }

    // 共同的實現部分可以放在父類別，避免程式碼重複
    protected function openFile() {
        echo "開啟檔案<br>";
    }

    protected abstract function readFile();    // primitiveOperation()
    protected abstract function parseFile();   // primitiveOperation()
    protected abstract function showFile();    // primitiveOperation()
    protected abstract function closeFile();   // primitiveOperation()

    protected function hook() { // Hook方法
        return true;
    }
}
```



Template Method(9/14):範例2:Hook(3/4)

```
class WordTextReader extends TextReader
{
    protected function readFile() {
        echo "讀取Word .doc格式檔案<br>";
    }

    protected function parseFile() {
        echo "解析Word .doc格式檔案<br>";
    }

    protected function showFile() {
        echo "顯示Word .doc格式檔案<br>";
    }

    protected function closeFile() {
        echo "關閉Word .doc格式檔案<br>";
    }
}
```

```
class AdobeTextReader extends TextReader
{
    protected function readFile() {
        echo "讀取Adobe .pdf格式檔案<br>";
    }

    protected function parseFile() {
        echo "";
    }

    protected function showFile() {
        echo "顯示Adobe .pdf格式檔案<br>";
    }

    protected function closeFile() {
        echo "顯示Adobe .pdf格式檔案<br>";
    }

    // 覆寫了父類別的Hook
    protected function hook() {
        return false;
    }
}
```

Template Method(10/14): 範例2: Hook(4/4)

```
include_once '../class/pattern/templateMethod.php';

$wordTextReader = new WordTextReader(); // 建立word文字閱讀器
$adobeTextReader = new AdobeTextReader(); // 建立adobe文字閱讀器

$wordTextReader->readFileProcess();

echo "<br>";

$adobeTextReader->readFileProcess();
```



開啓檔案
讀取Word .doc格式檔案
解析Word .doc格式檔案
顯示Word .doc格式檔案
關閉Word .doc格式檔案

開啓檔案
讀取Adobe .pdf格式檔案
顯示Adobe .pdf格式檔案
顯示Adobe .pdf格式檔案

由於adobeTextReader已經複寫了父類別的Hook，所以可以透過Hook針對演算法進行選擇的動作。

Template Method(11/14): 好萊塢守則(1/2)

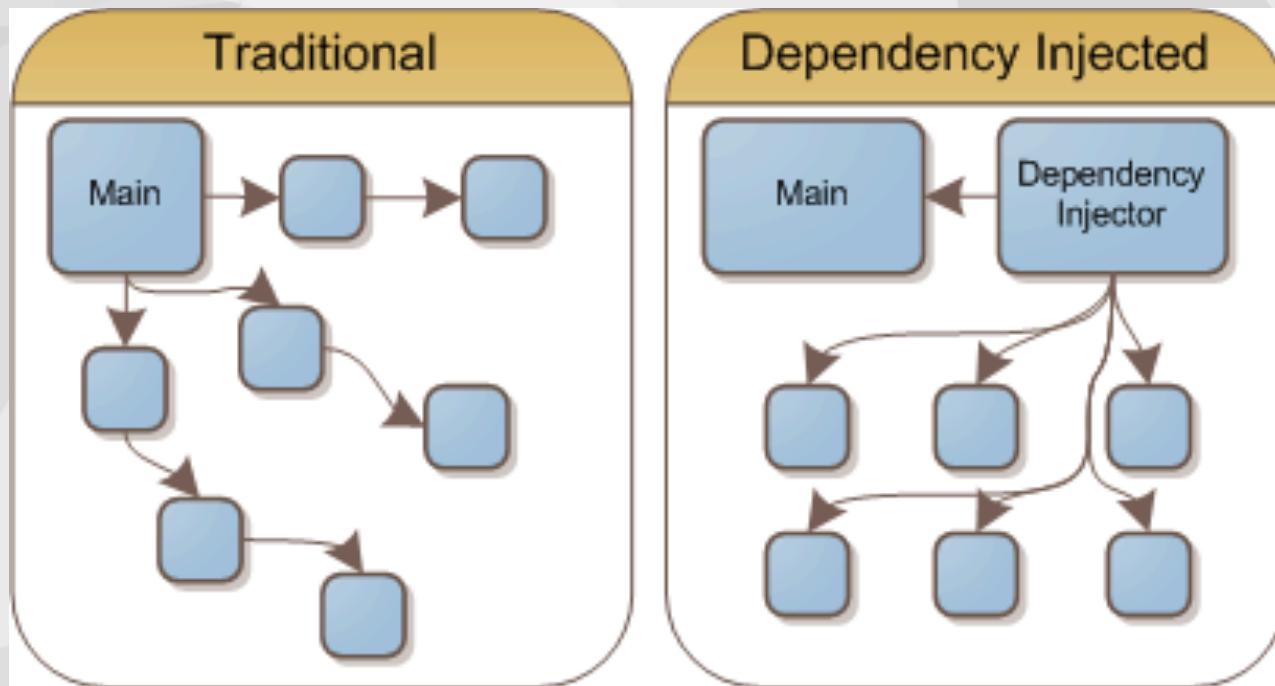
"Hollywood Principle" -- "Don't call us, we'll call you."

在好萊塢混過的人都會瞭解一個原則。當你把簡歷遞交給娛樂公司後，這些公司會告訴他們一句話：不要給我們打電話，我們會給你打。

在模板方法模式中，子類不顯式調用父類的方法，而是通過覆蓋父類的方法來實現某些具體的業務邏輯，父類控制對子類的調用，這種機制被稱為好萊塢原則

由父類完全控制着子類的邏輯，子類不需要調用父類，而通過父類來調用子類，子類可以實現父類的可變部份，卻繼承父類的邏輯，不能改變業務邏輯。

Template Method(12/14): 好萊塢守則(2/2)



Template Method_(13/14): Template Method vs. Strategy & Factory Method_(1/2)

- Template Method 跟 Strategy 都是用來封裝演算法，但是 Template method 是用繼承，Strategy 是用合成。
- Factory Method are often called by template methods.

TemplateMethod: 子類別決定如何實踐演算法中的某些步驟。

Strategy: 將可互換的行為封裝起來，然後使用委託的方式決定要採取哪一個行為。

Factory Method: 由子類別決定實體化哪個具象類別。

Template Method_(14/14): Template Method vs. Strategy & Factory Method_(2/2)

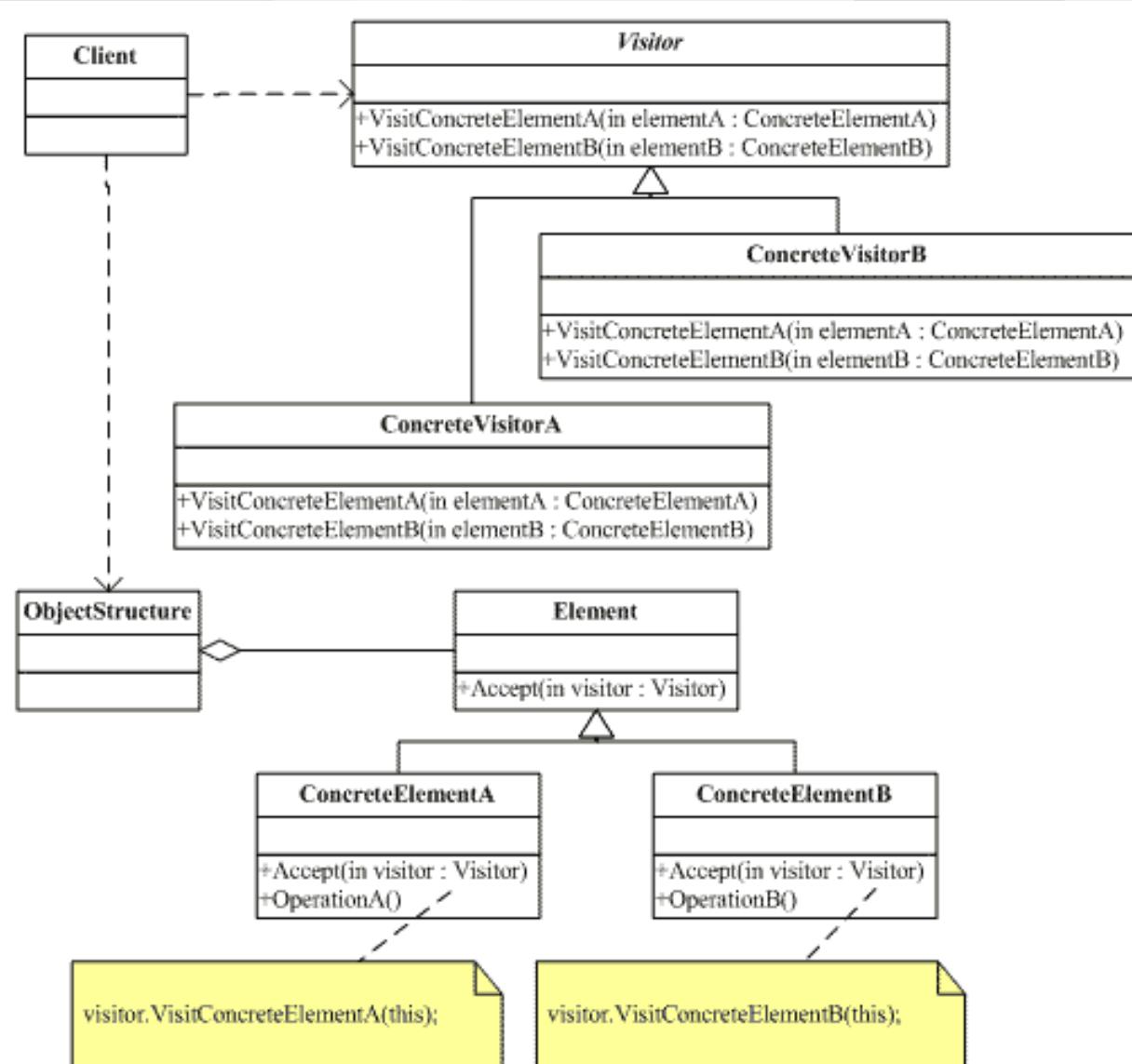
- Template method主要是父類將一個功能的實現分成幾個小的步驟，並且在模版方法中規定了這些步驟的執行順序，同時定義了這些子步驟的protected類型的方法留給子類實現，聲明為protected主要是以防止用戶不恰當的使用這些方法而產生異常。實現者不需要考慮這些子步驟地執行順序，只需要實現具體的功能就行了。其實每一個子類去具體的實現這些子步驟，也有策略的味道，只不過Template method主要是在父類封裝了子步驟地順序，從而簡化用戶的負擔以及減少重複的代碼的目的。
- Strategy模式主要是父類（更常用的是接口）定義一個具體的功能，不同的子類使用不同的策略去具體的實現。其實針對接口編程，一方面達到依賴於接口而不是具體實現的目的，另一方面也是策略模式的一個實現吧，接口規定了功能，不同的實現者提供了不同的策略。常用的Dao模式提供了隔離數據訪問與業務層的同時，也提供了可替換的策略，來方便在不同的數據庫間移植，或者使用不同的技術來實現。

Visitor(訪客模式)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

Visitor(1/10): UML(1/2)



Hiiir

Visitor(2/10):UML(2/2)

Visitor：宣告一個或者多個訪問操作，形成所有的具體元素角色必須實現的介面。

ConcreteVisitor：實現抽象Visitor角色所聲明的介面，也就是抽象Visitor所聲明的各個訪問操作。

Element：宣告一個接受操作，接受一個Visitor物件作為一個變數。

ConcreteElement：實現了抽象元素所規定的接受操作。

ObjectStructure：可以遍歷結構中的所有元素；如果需要，提供一個高層次的介面讓Visitor物件可以訪問每一個元素；如果需要，可以設計成一個複合物件或者一個聚集，如列（List）或集合（Set）。

Visitor(3/10): 模式分析(1/1)

優點：

- 不用修改具體的元素類，就可以增加新的操作。主要是通過元素類的accept方法來接受一個visitor物件來實現
- 將有關元素物件的訪問行為集中到一個訪問者物件中，而不是分散到一個個的元素類中
- 可以跨過類的等級結構訪問屬於不同的等級結構的元素類
- 讓使用者能夠在不修改現有類層次結構的情況下，定義該類層次結構的操作

缺點：

- 增加新的元素類很困難。在訪問者模式中，每增加一個新的元素類都意味著要在抽象訪問者角色中增加一個新的抽象操作，並在每一個具體訪問者類中增加相應的具體操作，違背了“開閉原則”的要求
- 破壞封裝。訪問者模式要求訪問者物件訪問並調用每一個元素物件的操作，這意味著元素物件有時候必須暴露一些自己的內部操作和內部狀態，否則無法供訪問者訪問

使用時機：

- 如果想對一個物件結構，實施一些依賴於物件結構中的具體類的操作
- 如果想對一個物件結構中的各個元素，進行很多不同的而且不相關的操作，為了避免這些操作使得類變得雜亂，可以使用訪問者模式，把這些操作分散到不同的訪問者物件中去，每個訪問者物件實現同一類功能
- 如果物件結構很少變動，但是需要經常給物件結構中的元素物件定義新的操作

Visitor(4/10):範例(1/7)

根據組合模式的例子，我們想要對單位增加一個輸出格式化文字訊息的方法，我們會怎麼做呢？

```
abstract class Unit {
    private $depth = 0;
    public function getComposite() {
        return null;
    }
    abstract function bombardStrength();

    // 新加文字訊息的操作
    public function textDump($num=0){
        $ret = "";
        $pad = $num;
        $ret .= "{$pad}";
        $ret .= get_class($this) . " : ";
        $ret .= "bombard: " . $this->bombardStrength() . "<br/>";
        return;
    }
}
```

```
abstract class CompositeUnit extends Unit {
    private $units = array();
    public function getComposite() {
        return $this;
    }
    protected function units() {
        return $this->units;
    }
    public function addUnit(Unit $unit) {
        if (in_Array($unit, $this->units, true)) {
            return;
        }
        $unit->setDepth($this->getDepth() + 1);
        $this->units[] = $unit;
    }
    public function removeUnit(Unit $unit) {
        // $this->units = array_udiff($this->units, array($unit), function($a, $b){ return ($a == $b)?0:1; }); //PHP 5.3寫法
        $this->units = array_udiff($this->units, array($unit), create_function('$a, $b', 'return ($a == $b)?0:1;'));
    }
    public function textDump($num=0) {
        $ret = parent::textDump($num);
        foreach($this->units as $unit){
            $ret .= $unit->textDump($num + 1);
        }
        return $ret;
    }
}
```

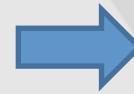
Visitor(5/10):範例(2/7)

```
$main_army = new Army();
$main_army->addUnit(new Archer());
$main_army->addUnit(new LaserCannonUnit());
$main_army->addUnit(new Cavalry());

$sub_army = new Army();
$sub_army->addUnit(new Archer());
$sub_army->addUnit(new Archer());
$sub_army->addUnit(new Archer());

$main_army->addUnit($sub_army);

echo $main_army->textDump();
```



```
(0)Army : bombard: 75
(1)Archer : bombard: 4
(1)LaserCannonUnit : bombard: 44
(1)Cavalry : bombard: 15
(1)Army : bombard: 12
(2)Archer : bombard: 4
(2)Archer : bombard: 4
(2)Archer : bombard: 4
```

Visitor(6/10):範例(3/7)

如果這時候我們還要加上其他的方法...?

能夠輕鬆遍歷所有物件是組合模式一大優勢，但是並非每個需要遍歷所有物件的操作都要寫在Composite類別裡面。

重點在於如何利用組合模式可以遍歷物件的優勢，但同時避免類別過度膨脹

Visitor(7/10):範例(4/7)

Visitor

```
abstract class ArmyVisitor
{
    abstract function visit(Unit $node);

    public function visitArcher(Archer $node) {
        $this->visit($node);
    }

    public function visitCavalry(Cavalry $node) {
        $this->visit($node);
    }

    public function visitLaserCannonUnit(LaserCannonUnit $node) {
        $this->visit($node);
    }

    public function visitTroopCarrierUnit(TroopCarrierUnit $node) {
        $this->visit($node);
    }

    public function visitArmy(Army $node) {
        $this->visit($node);
    }
}
```

ConcreteVisitor

```
class TextDumpArmyVisitor extends ArmyVisitor
{
    private $text = "";

    public function visit(Unit $node) {
        $ret = "";
        $pad = $node->getDepth();
        $ret .= "{$pad}";
        $ret .= get_class($node) . " : ";
        $ret .= "bombard: " . $node->bombardStrength() . "<br/>";
        $this->text .= $ret;
    }

    public function getText() {
        return $this->text;
    }
}
```

Visitor(8/10):範例(5/7)

Element

```
abstract class Unit {  
    private $depth = 0;  
  
    public function getComposite() {  
        return null;  
    }  
  
    abstract function bombardStrength();  
  
    public function accept(ArmyVisitor $visitor) {  
        $method = "visit" . get_class($this);  
        $visitor->$method($this);  
    }  
  
    protected function setDepth($depth) {  
        $this->depth=$depth;  
    }  
  
    public function getDepth() {  
        return $this->depth;  
    }  
}
```

ConcreteElement

```
abstract class CompositeUnit extends Unit {  
    private $units = array();  
  
    public function getComposite() {  
        return $this;  
    }  
  
    protected function units() {  
        return $this->units;  
    }  
  
    public function addUnit(Unit $unit) {  
        if (in_Array($unit, $this->units, true)) {  
            return;  
        }  
  
        $unit->setDepth($this->getDepth() + 1)  
        $this->units[] = $unit;  
    }  
  
    public function removeUnit(Unit $unit) {  
        // $this->units = array_udiff($this->units, array($unit), function($a, $b){ return ($a == $b)?0:1; }); //PHP 5.3寫法  
        $this->units = array_udiff($this->units, array($unit), create_function('$a, $b', 'return ($a == $b)?0:1;'));  
    }  
  
    public function accept(ArmyVisitor $visitor) {  
        parent::accept($visitor);  
        foreach($this->units as $thisunit){  
            $thisunit->accept($visitor);  
        }  
    }  
}
```

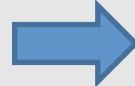
Visitor(9/10): 範例(6/7)

```
$archer = new Archer();

$textdump = new TextDumpArmyVisitor();
$archer->accept($textdump);
echo $textdump->getText();

echo "<br/>";


```



```
(0)Archer : bombard: 4

(0)Archer : bombard: 4
(0)Army : bombard: 75
(1)Archer : bombard: 4
(1)LaserCannonUnit : bombard: 44
(1)Cavalry : bombard: 15
(1)Army : bombard: 12
(1)Archer : bombard: 4
(1)Archer : bombard: 4
(1)Archer : bombard: 4
```

Visitor(10/10):範例(7/7)

如果要在加上一個遍歷方法，負責計算單位要繳的稅..

```
class TaxCollectionVisitor extends ArmyVisitor
{
    private $due = 0;
    private $report = "";

    public function visit(Unit $node) {
        $this->levy($node, 1);
    }

    public function visitArcher(Archer $node) {
        $this->levy($node, 2);
    }

    public function visitCavalry(Cavalry $node) {
        $this->levy($node, 3);
    }

    public function visitTroopCarrierUnit(TroopCarrierUnit $node) {
        $this->levy($node, 5);
    }

    private function levy(Unit $unit, $amount) {
        $this->report .= "Tax levied for " . get_class($unit);
        $this->report .= ":" . $amount . "<br/>";
        $this->due += $amount;
    }

    public function getReport() {
        return $this->report;
    }

    public function getTax() {
        return $this->due;
    }
}
```

```
// 列出每個單位要繳的稅金
$taxcollector = new TaxCollectionVisitor();

$archer->accept($taxcollector);
echo $taxcollector->getReport();

echo "<br/>";

$main_army->accept($taxcollector);
echo $taxcollector->getReport();

echo "<br/>";

// 列出總和
echo "total:" . $taxcollector->getTax();
```



Tax levied for Archer: 2
Tax levied for Archer: 2
Tax levied for Army: 1
Tax levied for Archer: 2
Tax levied for LaserCannonUnit: 1
Tax levied for Cavalry: 3
Tax levied for Army: 1
Tax levied for Archer: 2
Tax levied for Archer: 2
Tax levied for Archer: 2
total:16

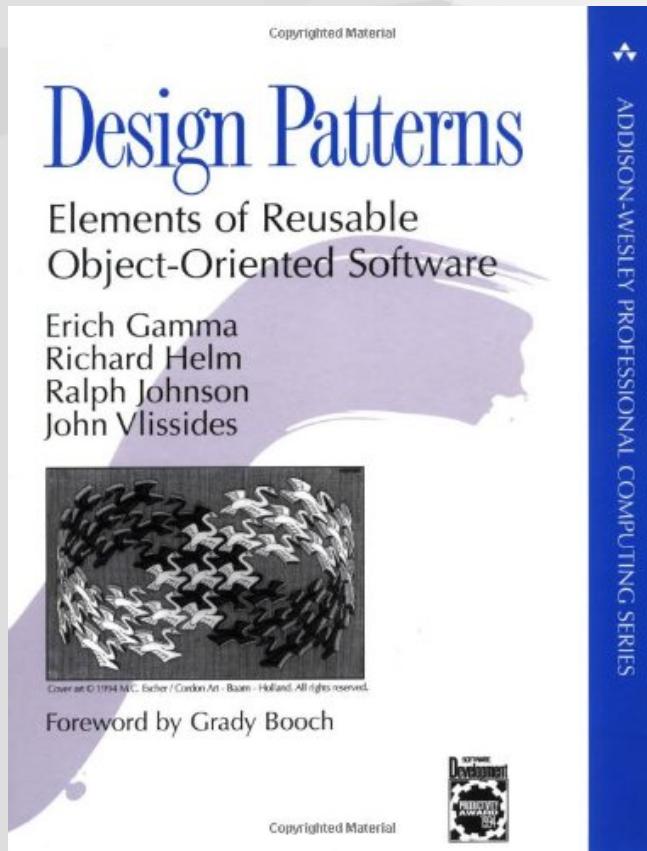
參考資料:書籍

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

參考資料：書籍(1/3)

設計模式



英文版:

http://www.tenlong.com.tw/items/0201633612?item_id=2480

中文版:

http://www.tenlong.com.tw/items/9572054112?item_id=10815

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

參考資料：書籍(2/3)

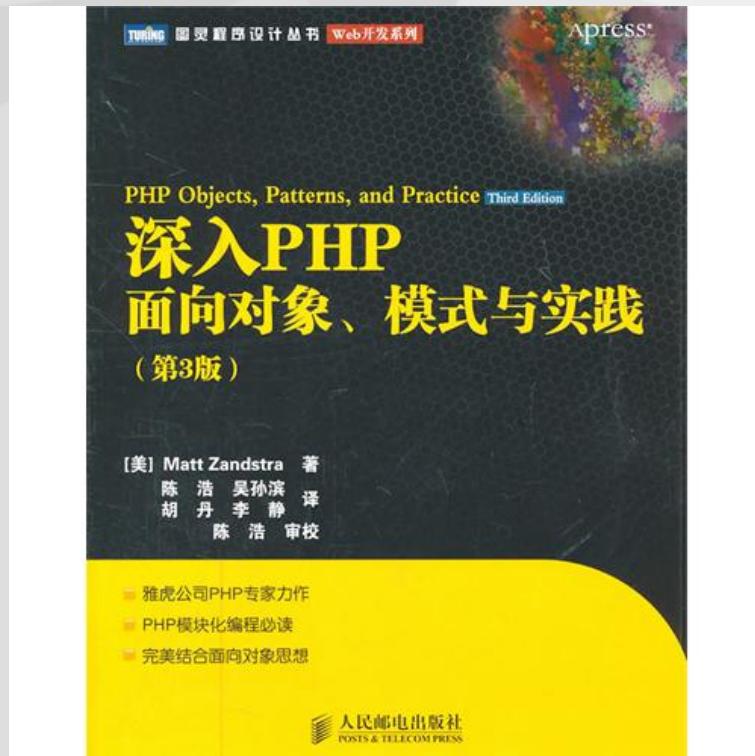
深入淺出設計模式 (Head First Design Patterns)



http://www.tenlong.com.tw/items/9867794524?item_id=33235

參考資料：書籍(3/3)

深入PHP：面向对象、模式与实践(第3版)



[http://product.dangdang.com/
product.aspx?product_id=22459608](http://product.dangdang.com/product.aspx?product_id=22459608)

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

參考資料: 網路

Hiiir

行動×商務 社群×媒體
mobile e-commerce social media

參考資料：網路(1/2)

- 史蒂芬心得筆記—Design Patterns(設計模式)
網址：<http://my.so-net.net.tw/idealist/Patterns/>
- 設計模式筆記
網址：<http://caterpillar.onlyfun.net/Gossip/DesignPattern/DesignPattern.htm>
- Snowball設計模式筆記
網址：<http://blog.csdn.net/yangzl2008/article/category/909721>
- 胖胖禿blog
網址：<http://mmmmmtodd.blogspot.tw/search/label/Design%20Pattern>
- 深入淺出設計模式-Google圖書(不完整)
網址：http://books.google.com.tw/books?id=I3U5PFHMZJ0C&printsec=frontcover&hl=zh-TW&source=gbs_ge_summary_r&cad=0#v=onepage&q=adapter&f=false
- sourcemaking
網址：http://sourcemaking.com/design_patterns

參考資料：網路(2/2)

- 跟屌丝大哥学习设计模式 Blog
網址：<http://www.blogjava.net/qileilove/category/53091.html>
- 何戈洲 专注于.NET中间件开发 Blog
網址：http://www.cnblogs.com/hegezhou_hot/category/260424.html
- VincentCZW Blog
網址：<http://www.cnblogs.com/BeyondAnyTime/category/379487.html>
- Sakura Blog
網址：<http://www.cnblogs.com/glory-jzx/category/379509.html>
- guisu · 程序人生。 Blog
網址：<http://blog.csdn.net/hguisu/article/category/1133340>
- 逆心 .net無涯 回頭是岸 Blog
網址：<http://www.cnblogs.com/kissdodog/category/460052.html>

編輯歷程(2/2)

時間	版本	說明	編輯人
2013.11.20	V1.31	Template Method關於好萊塢原則部分說明修改	Hank Kuo
2013.05.30	V1.3	1. 完成23個模式說明 2. PHP範例原始碼style調整	Hank Kuo
2012.12.20	V1	建立文件，完成12個模式說明	Hank Kuo