

常見設計模式介紹 - Part 1

瞭解不一樣的 PHP 開發方法

Jace Ju

2010/7/18

一切從需求開始

為 JPEG 圖檔加上浮水印

```
$file = 'images/test.jpg';  
if ($image = imagecreatefromjpeg($file)) {  
    $textColor = imagecolorallocate($image, 250, 250, 250);  
    imagestring($image, 5, 25, 400, 'Design Patterns', $textColor);  
    imagejpeg($image, 'output.jpg', 100);  
    imagedestroy($image);  
}
```

程式總是針對需求設計

而需求總是會改變

客戶要求加入對其他格式支援

```
$file = 'images/test.jpg';  
if ($image = imagecreatefromjpeg($file)) {  
    $textColor = imagecolorallocate($image, 250, 250, 250);  
    imagestring($image, 5, 25, 400, 'Design Patterns', $textColor);  
    imagejpeg($image, 'output.jpg', 100);  
    imagedestroy($image);  
}
```

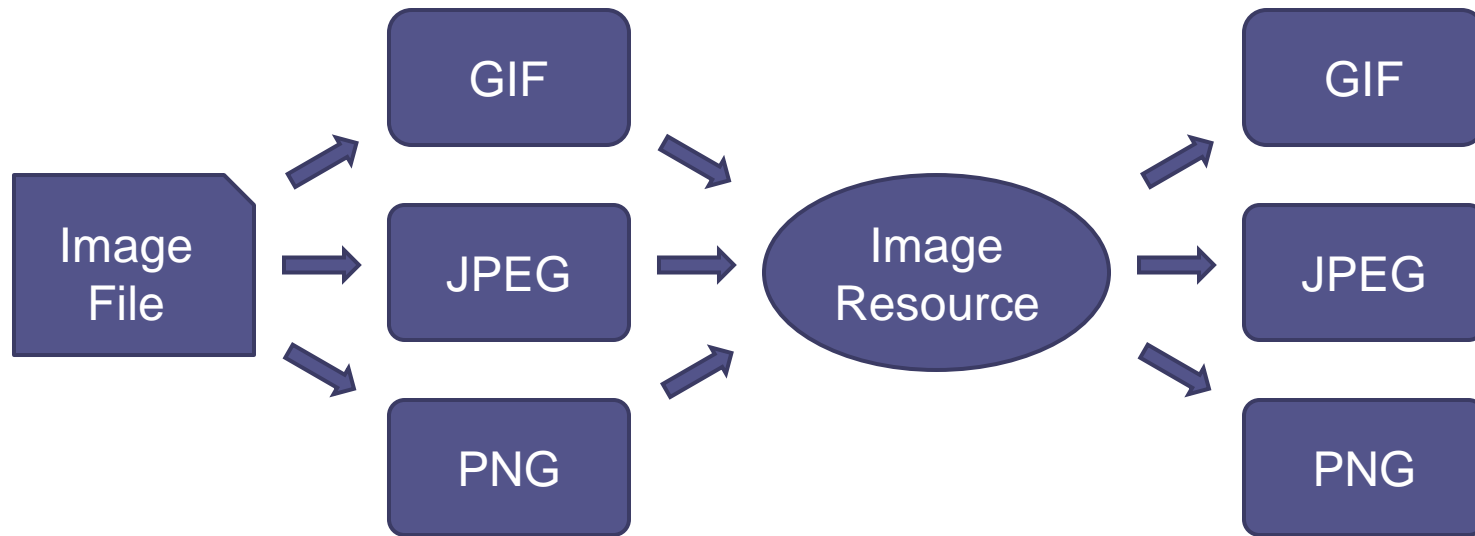
這裡寫死了

這裡也寫死了

為改變做設計

設計

- 我們需要取得圖檔的格式。
- 在建立圖片資源的部份要用格式來做判斷。
- 存檔的時候也要針對格式來做判斷。



封裝檔案資訊的取得

```
// 回傳 jpeg, gif, png 等資訊
function getimagetype($file) {
    $imageInfo = @getimagesize($file);
    return is_array($imageInfo)
        ? str_replace('image/', '', $imageInfo['mime'])
        : null;
}
```

mime 資訊會包含 image/xxx 的字串

封裝通用資源的建立

```
function imagecreatefrom($file, $imageType) {  
    switch ($imageType) {  
        case 'jpeg':  
            $image = imagecreatefromjpeg($file);  
            break;  
        case 'gif':  
            $image = imagecreatefromgif($file);  
            break;  
        case 'png':  
            $image = imagecreatefrompng($file);  
            break;  
        default:  
            $image = null;  
            break;  
    }  
    return $image;  
}
```

現在 `$image` 就是一個通用資源

封裝會變化的輸出

```
function imagesave($image, $filename, $imageType) {  
    switch ($imageType) {  
        case 'jpeg':  
            imagejpeg($image, $filename . '.jpg', 100);  
            break;  
        case 'gif':  
            imagegif($image, $filename . '.gif');  
            break;  
        case 'png':  
            imagepng($image, $filename . '.png');  
            break;  
        default:  
            break;  
    }  
}
```

輸出時也要還原格式

改用封裝後的函式

```
$file = 'images/test.jpg';  
$imageType = getimagetype($file);  
if ($image = imagecreatefrom($file, $imageType) {  
    $textColor = imagecolorallocate($image, 250, 250, 250);  
    imagestring($image, 5, 25, 400, 'Design Patterns', $textColor);  
    imagesave($image, rand() . '-output', $imageType);  
    imagedestroy($image);  
}
```

這裡用封裝後的函式來產生 resource

這裡也用封裝後的函式來存檔

從設計得到經驗

上面例子的設計

- 雖然都是圖，但格式不一樣，所以在載入不同格式檔案時都會用到 **switch** 切換。
- 將建立 **resource** 及寫入檔案的 **switch** 封裝起來。


相似的系統設計

- 客戶要求系統可更換資料庫系統，而雖然都是資料庫，但操作函式不一樣。
- 連結資料庫或執行 **SQL** 語法的函式會用到 **switch** 切換。
- 將資料庫操作上的不同封裝起來。

或是框架的設計...

- 框架可以支援不同格式的設定檔，像是 XML, YAML, INI 等。
- 處理資料格式的載入函式會用到 **switch** 切換。
- 將資料格式操作上的不同封裝起來。

我們得到的經驗



用函式或類別方法把
switch 封裝起來，讓程式
不要針對特定格式來實作

經驗變成模式

從經驗而得到的模式

- 相似的經驗可能會一再出現，只是應用層面不同。
- 所以我們可以把這些類似的經驗稱為「模式」。

所以模式是什麼？

- 「模式是在描述某種一再出現的問題，並描述解決方案的核心，讓你能據以變化出各種招式，解決上萬個類似的問題」 - [DP 中文版]
- 簡單來說，模式就是「將解決問題的經驗舉一反三後再利用」。

模式就像電影公式

常見的电影公式

- 只給壞人一刀，他一定不會死；一定會在主角轉身後，爬起來偷襲！
- 在大爆破的場景裡，主角一定要頭也不回地轉身帥氣離開。



如何表達模式？

阿凡達情節 (有雷)

- 男主角原是侵略的一方，愛上了原住民的女主角；結果後來男主角就反過來幫女主角擊退了侵略者...
- 我們給它一個名字：**外星版的風中奇緣**。

為模式命名的好處？

- 在講解程式時，你可能要摻雜很多知識才能讓別人瞭解。
- 用一個名詞來表示這個模式，讓知道的人可以立刻瞭解及互相溝通。

大師歸納的模式

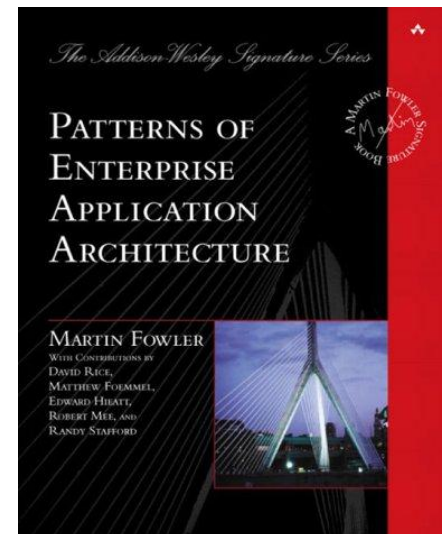
GoF Design Patterns

- 軟體界最早把模式收集成冊的書籍，是軟體開發聖經之一。
- 記錄了 23 種通用型模式。
- GoF: Gang of Four，指本書的四位作者：Erich Gamma, Richard Helm, Ralph Johnson 及 John Vlissides。
- 台灣有譯本：物件導向設計模式。



PoEAA

- Patterns of Enterprise Application Architecture
- 由作者 Martin Fowler 將企業應用軟體架構常見的模式整理成冊。
- 有名的模式像 MVC, ActiveRecord, FrontController 等。



Web 開發常見的模式

Strategy Pattern

常見的問題

- 我們有一個商品類別，包含了商品序號和價格兩種資訊。
- 商品分為 **A** 類及 **B** 類兩種。

處理商品價格

```
class Product
{
    protected $_sn = '';
    protected $_price = 0;

    public function __construct($sn, $price)
    {
        $this->_sn = $sn;
        $this->_price = (int) $price;
    }

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

    public function getPrice()
    {
        return $this->_price;
    }
}
```


處理商品價格

```
$products = array(  
    new Product('A0001', 100),  
    new Product('A0002', 150),  
    new Product('B0001', 300),  
    new Product('B0002', 200),  
);  
  
foreach ($products as $product) {  
    echo $product->getSn(), ' = ', $product->getPrice(), "\n";  
}
```

需求的變更

- 在某個時段把 **A** 類的商品價格打八折。
- 同一時段把 **B** 類的商品價格減 10 元。

處理商品價格

```
class Product
{
    public function __construct($sn, $price)
    {
        $this->_sn = $sn;
        switch (substr($this->_sn, 0, 1)) {
            case 'A':
                $this->_price = (int) $price * 0.8;
                break;
            case 'B':
                $this->_price = (int) $price - 10;
                break;
            default:
                $this->_price = (int) $price;
                break;
        }
    }
    // ...
}
```

在物件建立時，同時計算價格，
計算的規則是以 switch 切換

這樣寫的問題

- 每新增或修改需求，都需要在 `constructor` 裡加入 `switch` 判斷。
- 需求的變化可能更多，可能不是單靠 `switch` 就能處理。
- 一旦需求必須改回來，還得再改一次程式。

處理商品價格

```
class Product
{
    // ...

    public function __construct($sn, $price, Calculator $calculator)
    {
        $this->_sn = $sn;
        $this->_price = $calculator->calculate($price);
    }

    // ...
}
```

把計算規則移到外部物件
\$calculator 上面

處理商品價格

```
abstract class Calculator
{
    public function calculate($price) { return $price; }
}

class Calculator_20PercentOff extends Calculator
{
    public function calculate($price) { return $price * 0.8; }
}

class Calculator_10DollarMinus extends Calculator
{
    public function calculate($price) { return $price - 10; }
}
```

置放計算規則的類別

處理商品價格

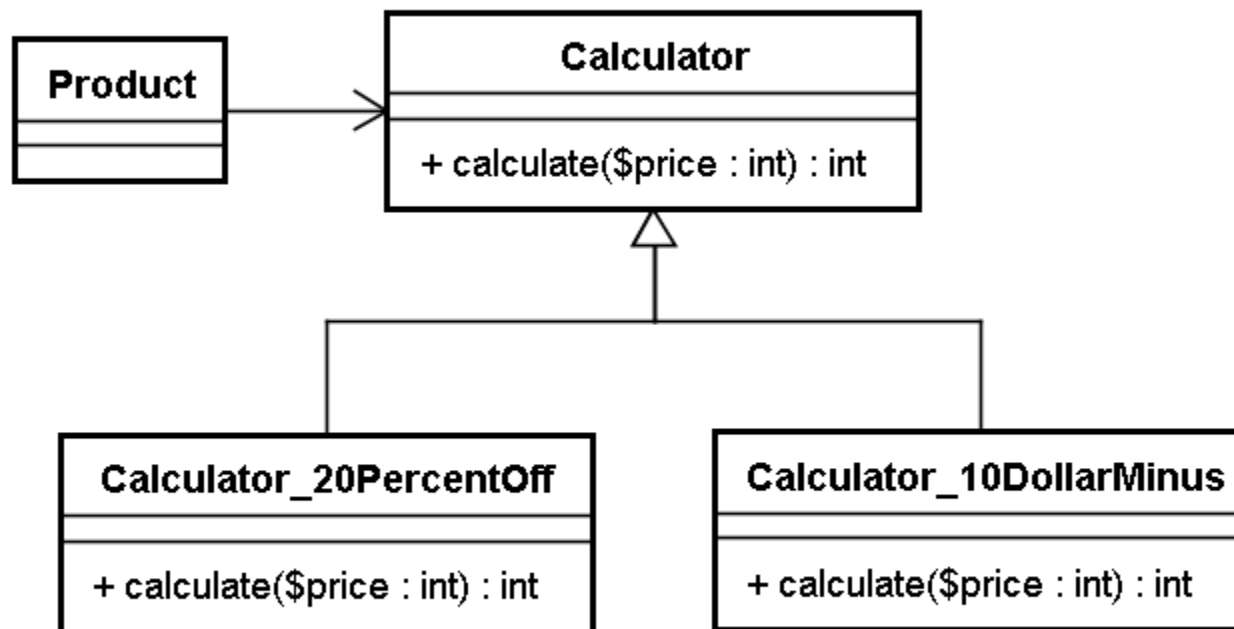
```
$calculatorA = new Calculator_20PercentOff();  
$calculatorB = new Calculator_10DollarMinus();
```

```
$products = array(  
    new Product('A0001', 100, $calculatorA),  
    new Product('A0002', 150, $calculatorA),  
    new Product('B0001', 300, $calculatorB),  
    new Product('B0002', 200, $calculatorB),  
);
```

```
foreach ($products as $product) {  
    echo $product->getSn(), ' = ', $product->getPrice(), "\n";  
}
```

現在計算規則改以外部帶入

處理商品價格



另一種形式的例子

- 我們有一個動態產生的表單類別 (像是一個 `PEAR::QuickForm` 的簡化版本) 。
- 表單物件可以接受不同的表單元件 。
- 表單元件型態不同，顯示的結果也不一樣 。

動態表單

```
class Form
{
    protected $_elements = array();

    public function addText($name, $value) {
        $this->_elements[$name] = array( 'type' => 'text', 'value' =>
$value );
    }

    public function addImage($name, $value)
    {
        $this->_elements[$name] = array( 'type' => 'image', 'value' =>
$value );
    }

    public function addSubmit($name, $value)
    {
        $this->_elements[$name] = array( 'type' => 'submit', 'value' =>
$value );
    }
}
// (接下頁)
```

目前我們有三種不同型態

動態表單

```
// (接上頁)
public function displayElement($name)
{
    $element = $this->_elements[$name];
    $value = $element['value'];
    switch ($element['type']) {
        case 'text':
            echo '<input type="text" name="' . $name . '" value="' .
htmlspecialchars($value) . '" />'; break;
        case 'image':
            echo ''; break;
        case 'submit':
            echo '<button type="submit">' . htmlspecialchars($value) .
'</button>'; break;
        default:
            echo ''; break;
    }
}
// 接下頁
```

可以顯示單一元件

動態表單

```
// (接上頁)
public function display()
{
    echo '<form action="" method="post">', "\n";
    foreach (array_keys($this->_elements) as $name) {
        $this->displayElement($name);
        echo "<br />\n";
    }
    echo '</form>', "\n";
}
```

完整輸出表單

動態表單

```
$form = new Form();
```

```
$form->addText('name', 'Jace Ju');
```

```
$form->addImage('avatar', 'pig.jpg');
```

```
$form->addSubmit('send', 'Send');
```

```
$form->display();
```

顯示表單 HTML

這樣寫的缺點

- `add<Type>` 等方法是寫死的。
- 新增一種型態的輸出時 (例如 `checkbox`)，要在 `Form` 類別新增一個 `addCheckbox` 方法，並在 `switch` 加入一個 `case 'checkbox'`。
- 能不能讓元件自己負責輸出？

動態表單

```
abstract class FormElement
{
    protected $_name = null;

    protected $_value = null;

    public function __construct($name, $value)
    {
        $this->_name = $name;
        $this->_value = $value;
    }

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

    abstract public function output();
}
```

定義一個抽象的 FormElement 類別，
其中 output 方法負責輸出

動態表單

```
class FormElement_Text extends FormElement
{
    public function output()
    {
        return '<input type="text" name="' . $this->_name . '" value="' .
        htmlspecialchars($this->_value) . '" />';
    }
}
```

處理文字輸入欄位的類別，繼承 FormElement，
並覆寫 output 方法，輸出 input 標籤

動態表單

```
class FormElement_Image extends FormElement
{
    public function output()
    {
        return '';
    }
}
```

圖片類別也是繼承 FormElement 類別，並覆寫 output 方法，輸出 img 標籤

動態表單

```
class FormElement_Submit extends FormElement
{
    public function output()
    {
        return '<button type="submit">' . htmlspecialchars($this->_value) .
        '</button>';
    }
}
```

送出按鈕類別也是繼承 FormElement 類別，
並覆寫 output 方法，輸出 button 標籤

動態表單

```
class Form
{
    protected $_elements = array();

    public function addElement(FormElement $element)
    {
        $this->_elements[$element->getName()] = $element;
    }
}
```

// 接下頁

改寫原來的 Form 類別，拿掉 add<Type> 等方法，
改用 addElement 方法

動態表單

// 接上頁

```
public function displayElement($name) {  
    echo $this->_elements[$name]->output();  
}  
  
public function display()  
{  
    echo '<form action="" method="post">', "\n";  
    foreach ($this->_elements as $name => $element) {  
        echo $element->output(), "<br />\n";  
    }  
    echo '</form>', "\n";  
}  
}
```

原來的 displayElement 的 switch 拿掉，直接使用 FormElement 的 output 方法，display 方法也一併更改

動態表單

```
$form = new Form();
```

```
$form->addElement(new FormElement_Text('name', 'Jace Ju'));
```

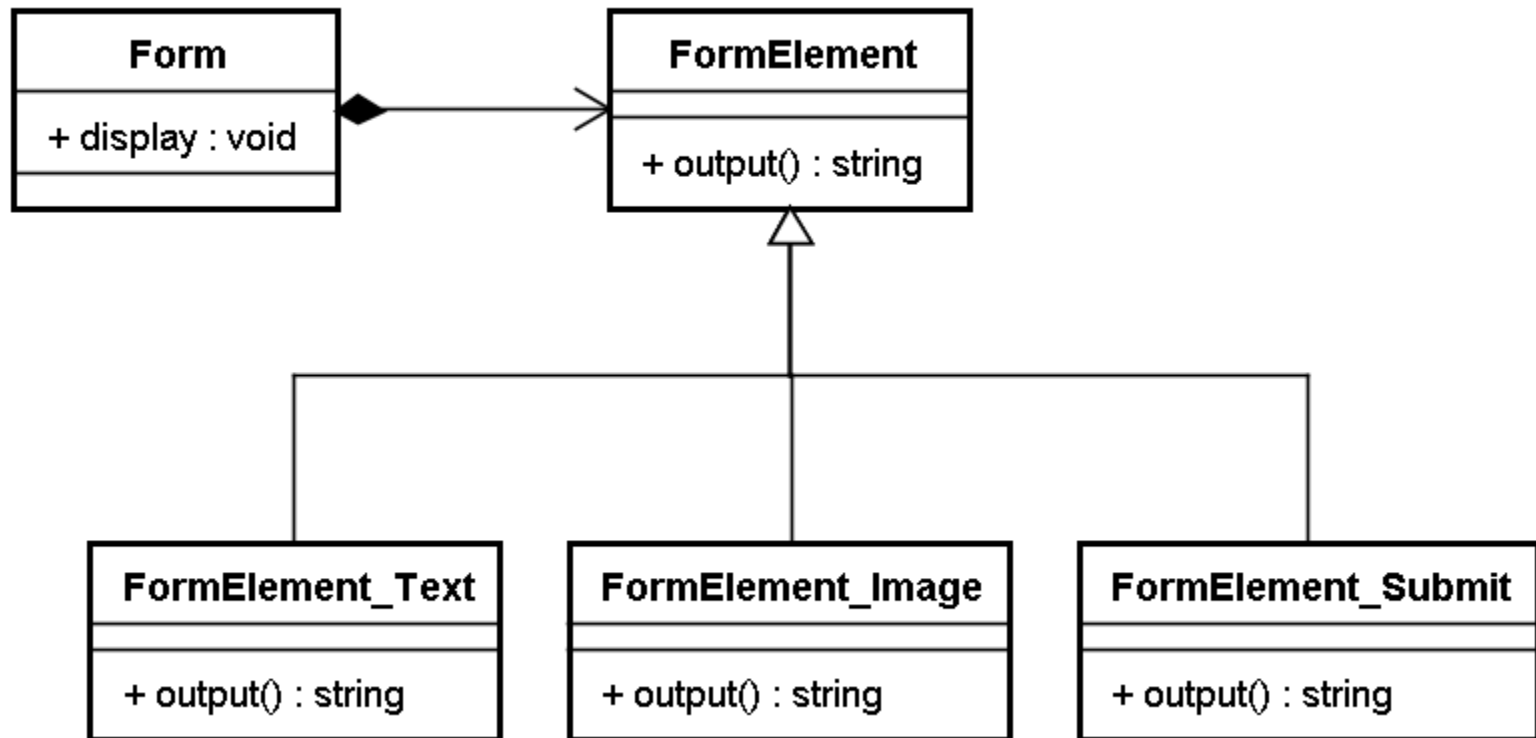
```
$form->addElement(new FormElement_Image('avatar', 'pig.jpg'));
```

```
$form->addElement(new FormElement_Submit('send', 'Send'));
```

```
$form->display();
```

顯示的方式由外部類別來決定

動態表單



Strategy Pattern 小結

- 需要將會改變的規則或演算法抽離出來的場合，
可以考慮使用 **Strategy Pattern** 。
- 抽離出來的規則或演算法可以隨時抽換，而不會
改動到使用主要程式的程式碼。
- 同一演算法若是有不同版本 (例如執行時間/所需
空間的不同)，也可以利用 **Strategy Pattern** 做切
換。



Simple Factory Pattern

常見的問題

- 繼承於相同抽象類的子類別群，如果在用戶端以 **new** 關鍵字來具現的話，就不容易改變；因為我們必須把類別名稱寫死在程式裡。
- 外部呼叫的程式需要知道很多子類別。

動態表單

```
$form = new Form();
```

```
$form->addElement(new FormElement_Text('name', 'Jace Ju'));
```

```
$form->addElement(new FormElement_Image('avatar', 'pig.jpg'));
```

```
$form->addElement(new FormElement_Submit('send', 'Send'));
```

```
$form->display();
```

用戶端必須知道有這三種類別，而且類別名稱寫死在程式裡，未來類別名稱若有變動就得回來更改

動態表單

```
class FormElement
{
    // ...
    public static function create($type, $name, $value)
    {
        switch (strtolower($type)) {
            case 'image':
                return new FormElement_Image($name, $value);
            case 'submit':
                return new FormElement_Submit($name, $value);
            case 'text':
            default:
                return new FormElement_Text($name, $value);
        }
    }
}
```

在 FormElement 類別中加入 create 方法來產生子類別具體化後的物件

動態表單

```
$form = new Form();
```

```
$form->addElement(FormElement::create('text', 'name', 'Jace Ju'));
```

```
$form->addElement(FormElement::create('image', 'avatar', 'pig.jpg'));
```

```
$form->addElement(FormElement::create('submit', 'send', 'Send'));
```

```
$form->display();
```

現在用戶端只要知道 FormElement 類別就好

這樣寫還是有缺點

- 醜陋的 `switch` 又跑出來了。
- 這麼一來新增類別還是需要修改程式碼。

動態表單

```
class FormElement
{
    // ...
    public static function create($type, $name, $value)
    {
        $className = 'FormElement_' . ucfirst(strtolower($type));
        if (class_exists($className)) {
            return new $className($name, $value);
        }
    }
}
```

PHP 的特色之一，把變數名稱當做類別名稱來用

這樣寫還是有缺點

- **FormElement** 這個父類別竟然要知道自己有哪些子類別？
- 用戶還是需要知道 **FormElement** 類別和它的 **create** 方法。
- 能不能讓用戶只知道 **Form** 類別就好？

動態表單

```
class Form
{
    // ...
    public function addElement($type, $name, $value)
    {
        $className = 'FormElement_' . ucfirst(strtolower($type));
        if (class_exists($className)) {
            return new $className($name, $value);
        }
    }
}
```

我們直接把 `FormElement::create` 用來建立表單元件的部份直接搬到 `Form` 裡

動態表單

```
$form = new Form();
```

```
$form->addElement('text', 'name', 'Jace Ju');
```

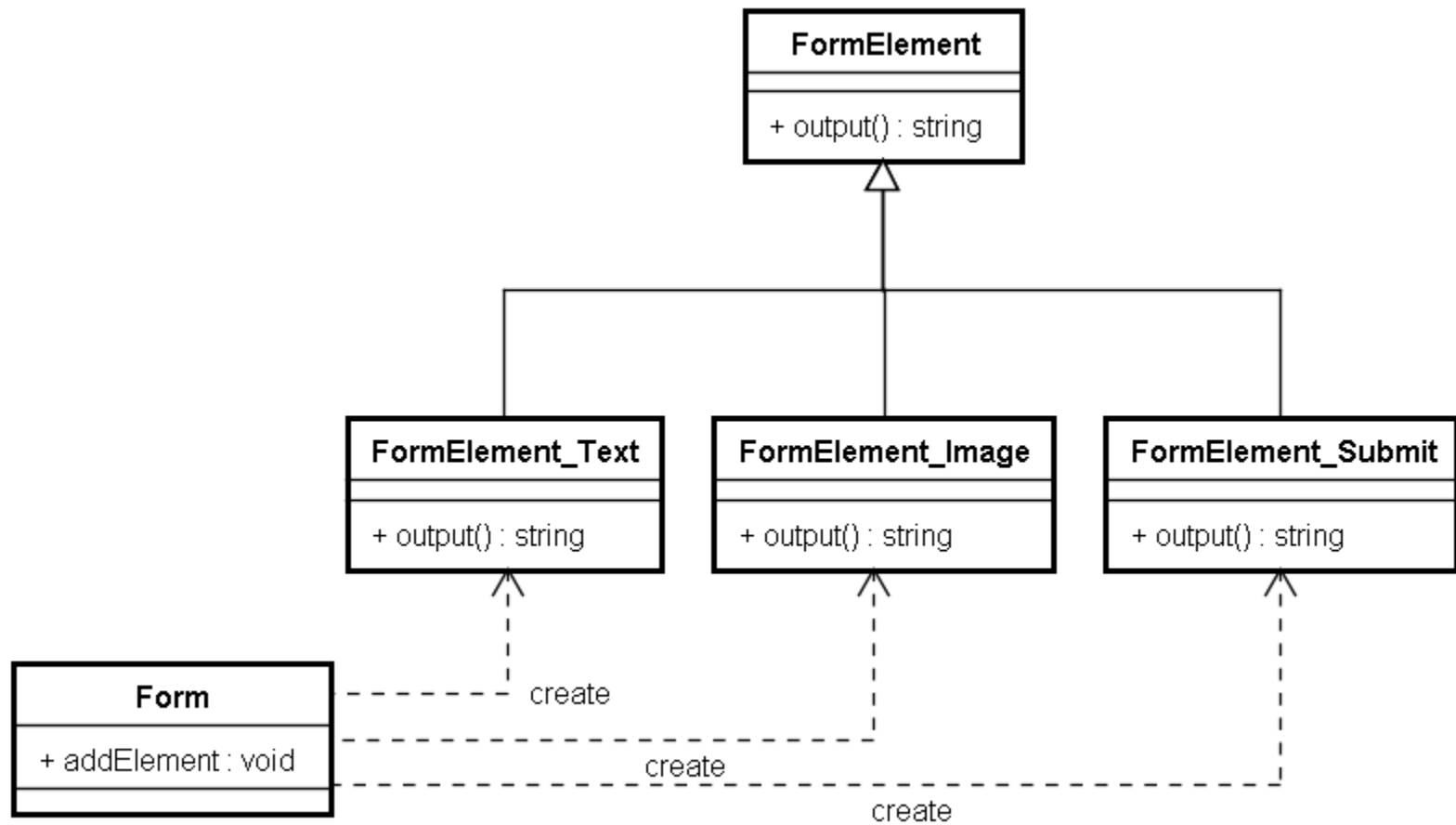
```
$form->addElement('image', 'avatar', 'pig.jpg');
```

```
$form->addElement('submit', 'send', 'Send');
```

```
$form->display();
```

現在用戶端就不必知道 FormElement 類別了

動態表單



迪米特法則 (LoD)

- 用戶端知道的越少越好。
- 儘可能把介面隱藏，只留下用戶端需要知道的。

Simple Factory Pattern 小結

- Simple Factory Pattern 將實體化的責任交給單一類別。
- PHP 在 Simple Factory Pattern 中可以採用變數來指定類別，移除對 switch 的依賴。



Adapter Pattern

常見問題

- 已經有了一個第三方套件，想把它加到我們的系統中。
- 但套件的使用方式跟我們現在的系統無法相容。

發送記錄

```
class MailLogger
{
    protected $_mailer = null;

    public function __construct()
    {
        $this->_mailer = new Mailer();
    }

    public function log($type, $message)
    {
        $this->_mailer->addMail('jaceju@gmail.com');
        $this->_mailer->setFrom('me@example.com');
        $this->_mailer->send($type, $message);
    }
}
```

假設 log 方法已經被大量被
佈署使用，更改它太花成本

發送記錄

```
class Mailer
{
    private $_mailList = array();
    private $_from = '';

    public function addMail($mail) { $this->_mailList[] = $mail; }

    public function setFrom($from) { $this->_from = $from; }

    public function send($subject, $body)
    {
        $headers = "From: {$this->_from}\r\n";
        foreach ($this->_mailList as $mail) {
            mail($mail, $subject, $body, $headers);
        }
    }
}
```

Mailer 是用舊的 mail 函式在送信

發送記錄

```
$logger = new MailLogger();  
$logger->log('Test', 'Go! Go! Go!');
```

MailLogger 的使用方式

新的需求

- 用戶希望透過 **SMTP** 發信。
- 用戶希望可以支援 **HTML**。

發送記錄

```
class MailLogger
{
    protected $_mailer = null;

    public function __construct()
    {
        $this->_mailer = new MailerAdapter();
    }

    public function log($type, $message)
    {
        $this->_mailer->addMail('jaceju@gmail.com');
        $this->_mailer->setFrom('me@example.com');
        $this->_mailer->send($type, $message);
    }
}
```

改用轉接器

發送記錄

```
require_once 'PHPMailer/class.phpmailer.php';  
class MailerAdapter extends Mailer  
{  
    private $_mailer = null;  
  
    public function __construct()  
    {  
        $this->_mailer = new PHPMailer();  
        $this->_mailer->IsSMTP();  
        $this->_mailer->Host = 'msa.hinet.net';  
        $this->_mailer->Username = 'username';  
        $this->_mailer->Password = 'password';  
    }  
}
```

// 接下頁

轉接 PHPMailer

發送記錄

// 接上頁

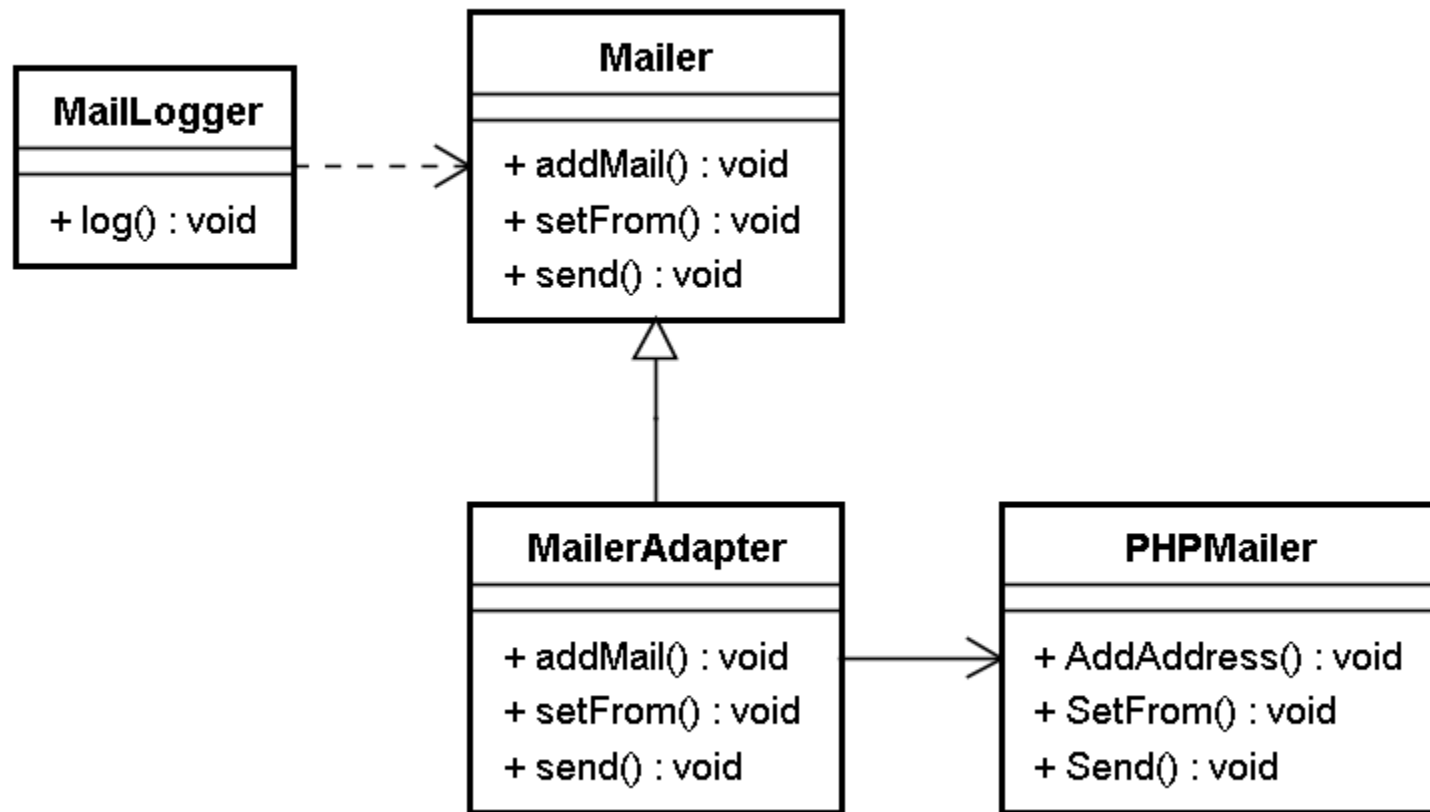
```
public function addMail($mail)
{
    $this->_mailer->AddAddress($mail);
}

public function setFrom($from)
{
    $this->_mailer->SetFrom($from);
}

public function send($subject, $body)
{
    $this->_mailer->Subject = $subject;
    $this->_mailer->MsgHTML($body);
    $this->_mailer->Send();
}
}
```

原來的三個方法都轉介為 PHPMailer 的方法

發送記錄



Adapter Pattern 小結

- Adapter Pattern 可以幫助我們包裝第三方套件來跟現有的系統介面整合。
- Adapter Pattern 不會加入新的介面，而是把第三方套件的介面跟現有系統的介面之間做轉接的動作而已。
- Adapter 的切換可以透過 Simple Factory 或設定檔來實現。如上面我們可以改用設定檔來切換 Mailer 及 MailerAdapter 兩個類別。



Template Method Pattern

客戶的需求

- 我們想要實作一個下載網路相簿的程式。
- 這個程式要支援多家網路相簿。
- 下載的流程大致上是相同的，但照片的擷取的方式可能不太一樣。



相簿下載

```
class PhotoAlbum
{
    protected $_albumType = '';

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

```
    public function download()
    {
        if (!$this->_openUrl()) {
            die ('無效的網址. ');
        }
        $this->_parsePage();
        $this->_savePicture();
    }
```

主要的流程定義

_savePicture 方法做的事不會因為相簿類型不同而改變

```
    protected function _savePicture() { echo "儲存照片\n"; }
```

// 接下頁

相簿下載

// 接上頁

```
protected function _openUrl()  
{  
    switch ($this->_albumType) {  
        case 'wretch':  
            echo "開啟無名相簿網址\n";  
            return true;  
        case 'pixnet':  
            echo "開啟 Pixnet 相簿網址\n";  
            return true;  
        default:  
            return false;  
    }  
}
```

用 switch 判斷要取用
哪個相簿的網址

// 接下頁

相簿下載

// 接上頁

```
protected function _parsePage()  
{
```

```
    switch ($this->_albumType) {
```

```
        case 'wretch':
```

```
            echo "解析無名相簿頁面，並存到陣列中\n";
```

```
            break;
```

```
        case 'pixnet':
```

```
            echo "解析 Pixnet 相簿頁面，並存到陣列中\n";
```

```
            break;
```

```
        default:
```

```
            break;
```

```
    }
```

```
}
```

```
}
```

也要用 switch 判斷頁面的格式，以取得圖片網址

相簿下載

```
$albumList = array(  
    new PhotoAlbum('wretch'),  
    new PhotoAlbum('pixnet'),  
);  
  
foreach ($albumList as $album) {  
    $album->download();  
}
```

用建構子的參數來判斷
要下載哪一種相簿

這樣寫的問題

- 令人不安的 **switch** 又出現了。
- 如果又有新的相簿服務要加入下載，就得更改 **_openUrl** 和 **_parsePage** 兩個方法。

相簿下載

```
class PhotoAlbum
{
    public function download()
    {
        if (!$this->_openUrl()) {
            die ('Invalid url.');
```



```
        }
        $this->_parsePage();
        $this->_savePicture();
    }

    protected function _savePicture() { echo "儲存照片\n"; }

    protected function _openUrl() { return false; }

    protected function _parsePage() {}
}
```

把 `_openUrl` 和 `_parsePage` 交給子類別完成，它們稱為勾子 (Hook)

相簿下載

```
class PhotoAlbum_Wretch extends PhotoAlbum
{
    protected function _openUrl()
    {
        echo "開啟無名相簿網址\n";
        return true;
    }

    protected function _parsePage()
    {
        echo "解析無名相簿頁面，並存到陣列中\n";
    }
}
```

子類別只要實作 `_openUrl` 和 `_parsePage` 兩個勾子方法即可

相簿下載

```
class PhotoAlbum_Pixnet extends PhotoAlbum
{
    protected function _openUrl()
    {
        echo "開啟 Pixnet 相簿網址\n";
        return true;
    }

    protected function _parsePage()
    {
        echo "解析 Pixnet 相簿頁面，並存到陣列中\n";
    }
}
```

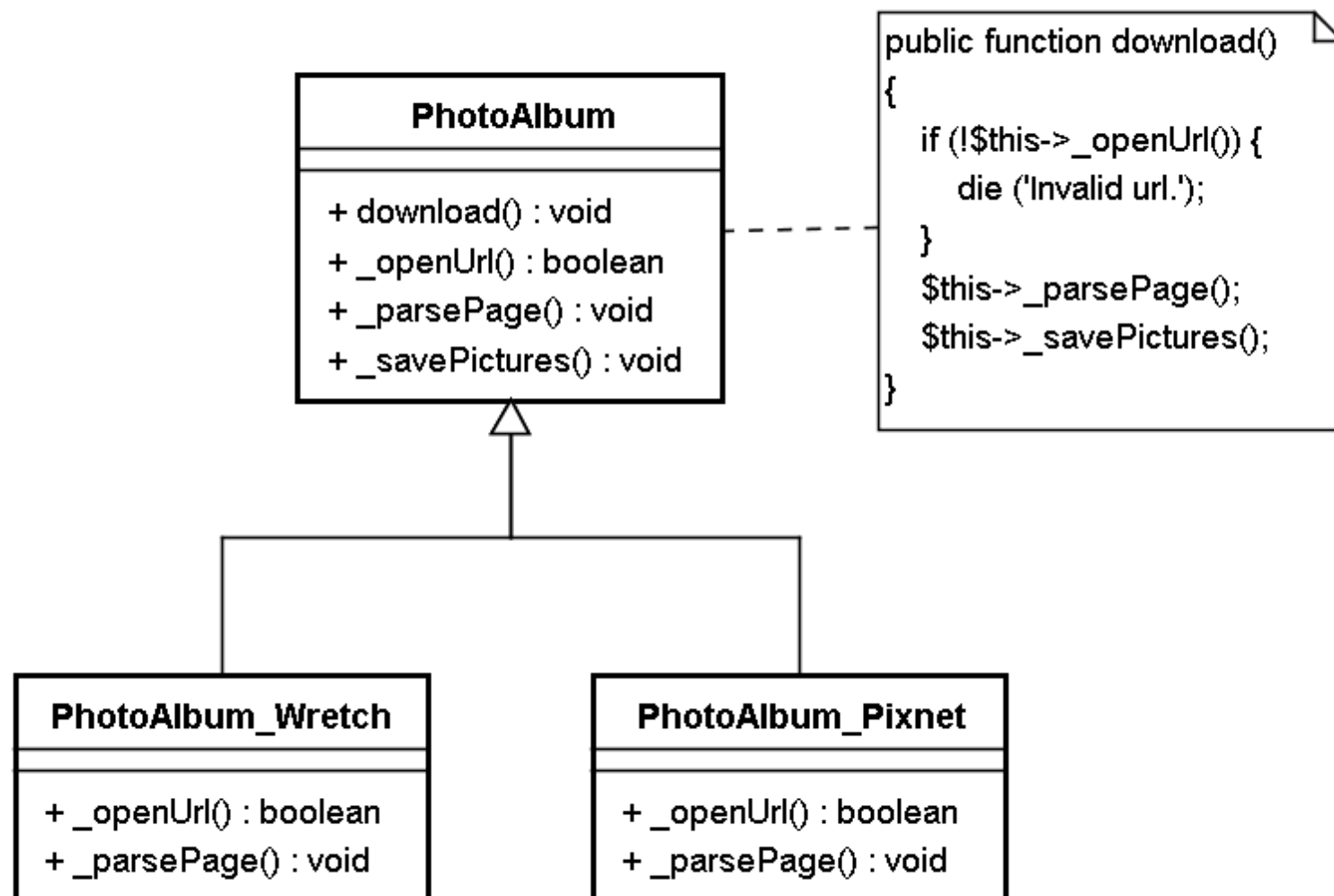
子類別只要實作 `_openUrl` 和 `_parsePage` 兩個勾子方法即可

相簿下載

```
$albumList = array(  
    new PhotoAlbum_Wretch(),  
    new PhotoAlbum_Pixnet(),  
);  
  
foreach ($albumList as $album) {  
    $album->download();  
}
```

改用子類別來判斷要下載
哪一種相簿

相簿下載



另一種表現方式

- 客戶希望能夠替換資料排序的方式。
- 而這個方式可以讓客戶自行決定。

資料排序

```
$sample = array(3, 7, 5, 1, 4, 5, 2);
```

```
sort($sample);
```

```
print_r($sample);
```

排序的方向被寫死了

```
/* Output:
```

```
Array
```

```
(
```

```
    [0] => 1
```

```
    [1] => 2
```

```
    [2] => 3
```

```
    [3] => 4
```

```
    [4] => 5
```

```
    [5] => 5
```

```
    [6] => 7
```

```
)
```

```
*/
```

為什麼是 Template Method ？

- 在 PHP 原始碼裡，陣列排序的方法 (zend_hash_sort) 提供了我們一個勾子。
- 而 sort 方法是使用預設的勾子 (array_data_compare)。
- 我們可以透過這個勾子替換陣列排序的順序。

資料排序

```
function orderByAsc($l, $r) {  
    return ($l == $r) ? 0  
        : (($l < $r) ? -1 : 1);  
}
```

定義兩種不同的排序方法 (實作勾子)

```
function orderByDesc($l, $r) {  
    return ($l == $r) ? 0  
        : (($l > $r) ? -1 : 1);  
}
```

```
$sample = array(3, 7, 5, 1, 4, 6, 2);  
usort($sample, 'orderByDesc');  
print_r($sample);
```

現在我們可以更換排序方向了

Template Method Pattern 小結

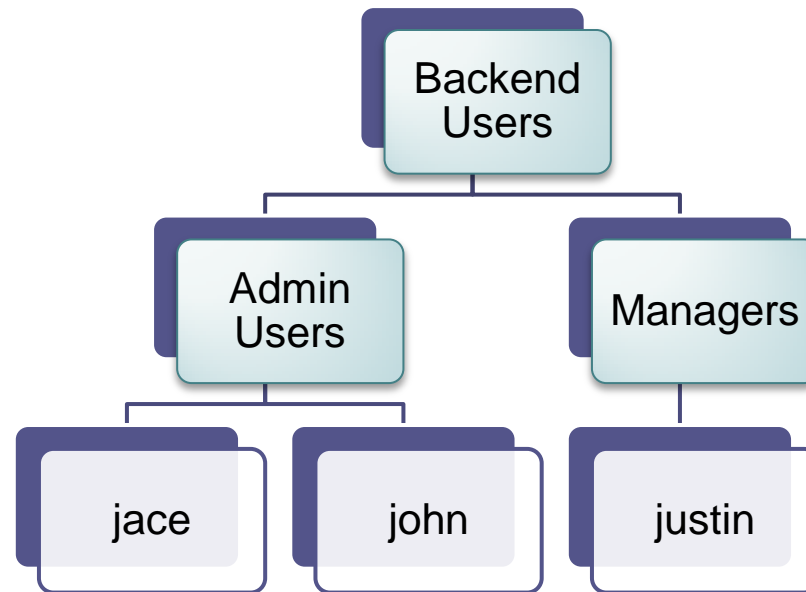
- 在演算法或流程大致相同，但實作細節不同時，
Template Method 可以提供解法方案。
- Template Method 的父類別的流程 (或演算法) 也是實作的一種。
- 實作細節有越多的不同，我們的流程就可以切得越細，提供更多勾子 (Hook)。
- Don't call me, I call you. (好萊塢守則)



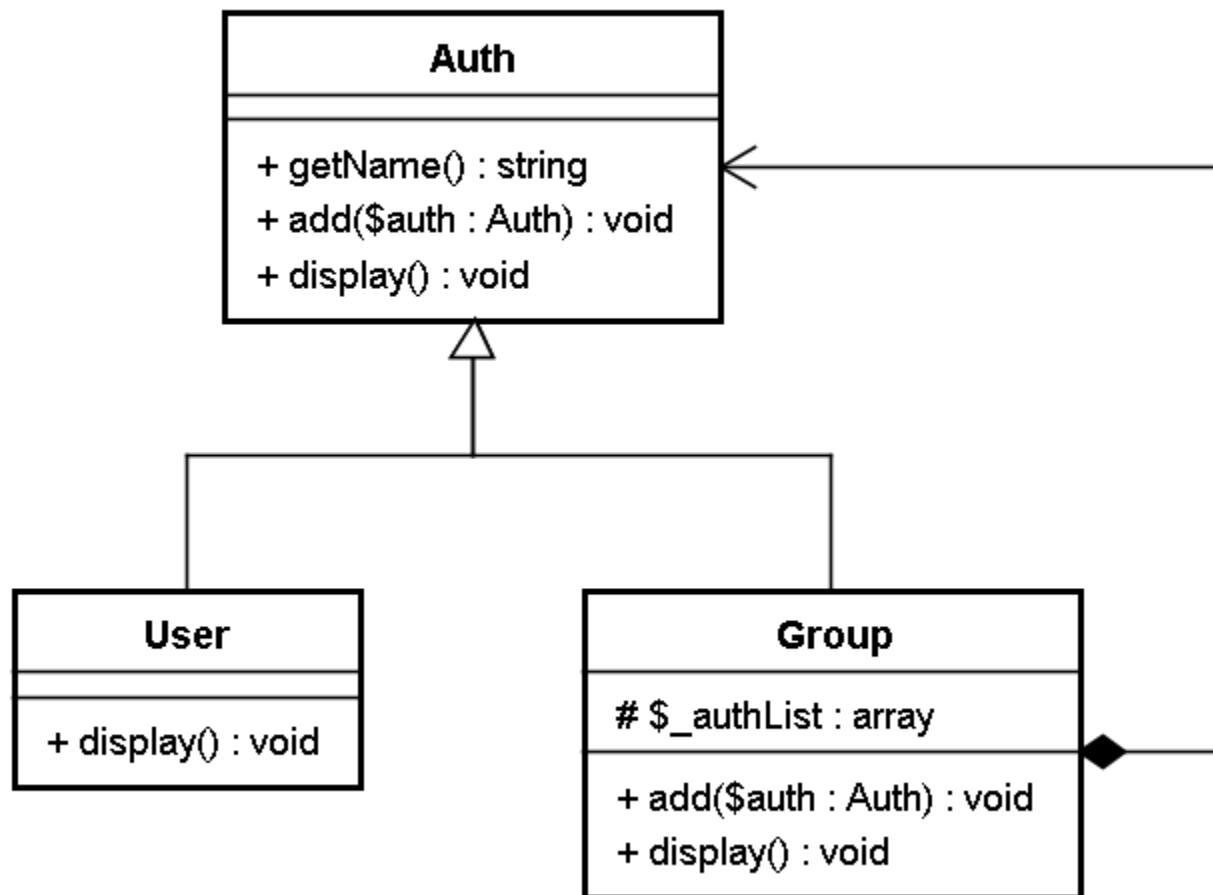
Composite Pattern

問題

- 樹狀型的權限系統中，如何讓使用者加入群組，而群組也可以加入群組？
- 如何更直覺地儲存與呈現樹狀架構？



設計方向



使用者與群組

```
abstract class Auth
{
    protected $_name = '';

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

    public function getName() { return $this->_name; }
    public function add(Auth $auth) {}
    public function display($depth = 0) {}
}
```

我們不必知道它是群組還是使用者

使用者與群組

```
class Group extends Auth
{
    protected $_authList = array();

    public function add(Auth $auth)
    {
        if ($auth === $this) { die('Fail add!'); }
        $this->_authList[$auth->getName()] = $auth;
    }

    public function display($depth = 0)
    {
        echo str_repeat(' ', $depth);
        echo $this->_name, "\n";
        foreach ($this->_authList as $name => $auth) {
            $auth->display($depth + 2);
        }
    }
}
```

我們只要呼叫下一層的 display() 方法，
剩下的事自動會發生

使用者與目錄

```
class User extends Auth
{
    public function display($depth = 0)
    {
        echo str_repeat(' ', $depth);
        echo $this->_name, "\n";
    }
}
```



把自己秀出來就好，其他不管

使用者與目錄

```
$userGroup = new Group('Backend Users');  
$adminGroup = new Group('Admin Users');  
$managerGroup = new Group('Managers');  
$user1 = new User('jace');  
$user2 = new User('john');  
$user3 = new User('justin');  
$adminGroup->add($user1);  
$adminGroup->add($user2);  
$managerGroup->add($user3);  
$userGroup->add($adminGroup);  
$userGroup->add($managerGroup);  
$userGroup->display();
```

呼叫最上層的 display() 後，
其他動作自動完成

```
/*  
Backend Users  
  Admin Users  
    jace  
    john  
  Managers  
    justin  
*/
```

Liskov 替換原則

- **子類別**要能夠替換**父類別**。
- 也就是呼叫的程式不需要區分父類別或子類別。

Composite Pattern 小結

- Composite Pattern 可以用一致的方式來操作群組與個體。
- Composite Pattern 很適合用來處理樹狀結構。
- 要小心別讓群組自己 add 到自己或自己的上一層，以避免無限遞迴。
- 所以可以在物件裡保留一份對上一層群組的參考。



Observer Pattern

問題

- 會員註冊時，可能會同時會需要執行很多動作。
- 例如把會員資料寫入 Web 端資料庫、贈送折價券、寄發通知信等。

會員註冊

```
class Member
{
    public function save()
    {
        $this->_saveData();
        $this->_giveCoupon();
        $this->_sendMail();
    }

    protected function _saveData() { echo "會員資料寫入資料庫\n"; }
    protected function _giveCoupon() { echo "贈送會員折價券\n"; }
    protected function _sendMail() { echo "寄發會員通知信\n"; }
}

$member = new Member();
$member->save();
```

會員註冊時，通常需要再執行很多動作

這樣寫的缺點

- 客戶經常會改變這裡的動作。
- 像是在某段期間註冊完成後，透過 **API** 更新客戶的 **ERP** 系統。
- 有什麼方法可以不必更動 **save** 方法，而可以擴充它？

會員註冊

```
interface Observer
```

```
{
```

```
    public function update(Subject $subject);
```

```
}
```

```
interface Subject
```

```
{
```

```
    public function register($name, Observer $observer);
```

```
    public function unregister($name);
```

```
    public function notify();
```

```
}
```

觀察者只需要一個 update 方法

主題則需要知道自己有哪些觀察者，
並且也需要通知的方式

會員註冊

class Member implements Subject

```
{  
    protected $_observerList = array();
```

實作 Subject 介面的三個方法

```
public function register($name, Observer $observer)  
{  
    $this->_observerList[$name] = $observer;  
}
```

```
public function unregister($name)  
{  
    unset($this->_observerList[$name]);  
}
```

```
public function notify()  
{  
    foreach ($this->_observerList as $observer) {  
        $observer->update($this);  
    }  
}
```

Observer 的重點所在

// 接下頁

會員註冊

// 接上頁

```
protected $_data = array();
```

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

```
public function save()  
{  
    $this->_saveData();  
    $this->notify();  
}
```

通知要動作的 Observer

```
protected function _saveData() { echo "會員資料寫入資料庫\n"; }
```

```
}
```


會員註冊

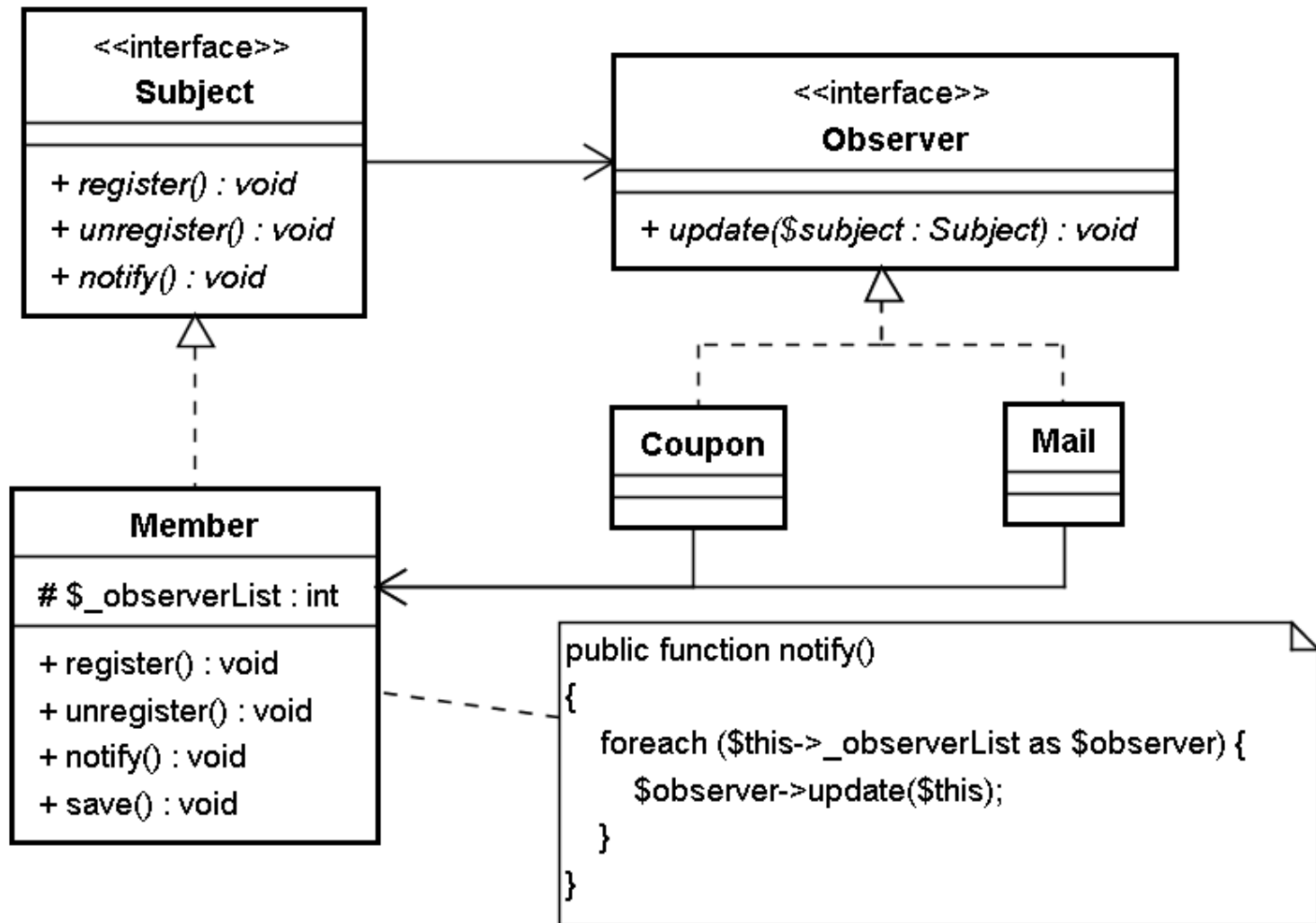
```
class Coupon implements Observer
{
    public function update(Subject $subject)
    {
        echo "贈送會員折價券\n";
    }
}
```

```
class Mail implements Observer
{
    public function update(Subject $subject)
    {
        echo "寄發會員通知信\n";
    }
}
```

```
$member = new Member();
$member->register('coupon', new Coupon());
$member->register('mail', new Mail());
$member->save();
```

Observer 獨立出來，這樣要增刪功能就容易得多

會員註冊



Observer Pattern 小結

- **Observer Pattern** 很適合在物件狀態更新時，需要同時執行其他功能的狀況。
- **Observer Pattern** 提供程式可以動態加入或移除物件對它的監聽。
- 我們也可以讓 **Observer** 只監聽有興趣的事件。
- 注意過於頻繁地呼叫 **notify** 方法，造成重複 **update**。

Part 1 小結

模式是一種框架嗎？

- 模式不是一種框架，但框架常會用到模式。
- 框架裡常會包含許多以模式存在的功能機制。
- 像 Zend Framework 裡的 `Zend_Db_Adapter` 就是把 `PDO`, `MySQLi` 等性質相近，但介面不同的套件轉介給自己的 `Zend_Db` 使用。

如何把模式導入設計？

- Design for Change.
- 如果目的明確，一開始可以考慮用模式設計。
- 比較好的方式是讓程式透過重構漸漸來接近模式。

下次見