技術筆記

關於 設計模式

筆記更新紀錄

|  |  |  |
| --- | --- | --- |
| 日期 yyyy/MM/dd | 更新人員 | 更新備註 |
| 2019/07/20 | Miles Chien | 建立 設計模式- Simple Factory Pattern 內容 |
| 2019/07/21 | Miles Chien | 建立 設計模式- Factory Method Pattern 內容 |
| 2019/07/22 | Miles Chien | 建立 設計模式- Abstract Factory Pattern 內容 |
| 2019/07/23 | Miles Chien | 建立 設計模式- SOLID 內容 |
| 2019/07/25 | Miles Chien | 建立 設計模式- Adapter Pattern 內容 |
| 2019/07/26 | Miles Chien | 更新 設計模式- Adapter Pattern 更多內容 |
| 2019/07/30 | Miles Chien | 建立 設計模式-Builder Pattern 內容 |
| 2020/05/06 | Miles Chien | 建立 設計模式-Command Pattern 內容 |

目錄

[1. SOLID 4](#_Toc39658063)

[2. Simple Factory Pattern/Static SimpleFacotry Pattern（創建型模式） 5](#_Toc39658064)

[3. Factory Method Pattern（創建型模式） 9](#_Toc39658065)

[4. Abstract Factory Pattern（創建型模式） 12](#_Toc39658066)

[5. Adapter Pattern(Object Adapter/Class Adapter)（結構型模式） 16](#_Toc39658067)

[6. Bridge Pattern（結構型模式） 19](#_Toc39658068)

[7. Builder Pattern（創建型模式） 20](#_Toc39658069)

[8. Chain of Responsibility Pattern (行為模式) 23](#_Toc39658070)

[9. Command Pattern(行為模式) 26](#_Toc39658071)

[模板 26](#_Toc39658072)

## SOLID

|  |
| --- |
| 簡介與說明 |
| 1. S = Single Responsibility Principle(SRP) 單一職責(單一責任原則)   指一個類別只負責一件事情，從這樣的講解當中，我們要注意不能將功能切得太細導致過度設計(over design)的情況。  例如一個方法(method)就只做一件事情，一個類別(class)就只做一個功能該做的事情，多於的變數(value)、方法(method)就不需要出現。   1. O = Open/Close Principle(OCP) 開放/封閉原則   物件導向設計當中最重要的開放(擴充)與封閉(修改)原則。一套軟體必須保留高延展性，方便日後的更新與修改，必須減少過度的耦合(Coupling)發生。而延展模組就是物件導向中的繼承  ，而延展的方法相當多，例如直接繼承舊有的模組並且覆蓋新功能上去(Override)、繼承抽象類別、透過抽象類別的規範來實現新功能、透過多型(Overload)來實現新功能。   1. L = Liskov Substitution Principle(LSP) Liskov替換   在一個系統中，子類別應該要可以替換掉父類別而不會受到影響程式架構。  子類別應該要可以執行父類別想做的事情。   1. I = Interface Segregation Principle(ISP) 介面隔離   把不同功能的功能從介面中分離出來。   1. D = Dependency Inversion Principle(DIP) 依賴凡轉   高階模組不應依賴低階模組，兩個都應該依賴在抽象概念上;抽象概念不依賴細節，而是細節依賴在抽象概念。意思就是說: 話不能說的太死，盡量講一些概念性的東西。  子類別肯定依賴父類別，而父類別不能去依賴子類別實作的功能。  抽象類別不應該有太多細節，而實作類別依賴抽象細節，意思是說依賴應該是單向的。因為程式設計上最好能達到低耦合，可惜光繼承就是一個依賴，因此單向的依賴是最乾淨的，雙向的依賴就違反常理的依賴會導致程式碼不容易追蹤。 |
| 適用時機 |
| 任何時候，基本功 |
| 參考範例 |
| 無 |
| 參考網站 |
|  |

## Simple Factory Pattern/Static SimpleFacotry Pattern（創建型模式）

|  |
| --- |
| 簡介 |
| 1. 稱為Static Simple Factory Pattern 與 Simple Factory Pattern。 2. 對客戶端隱藏產品產生的細節，物件如何生成，生成前是否與其他物件建立依賴關係，客戶端皆不用理會，用以將物件生成方式之變化 與客戶端程式碼隔離。 3. 根據需求，直接由工廠產生該物件出來。 |
| 適用時機 |
| 1. 統一建立物件。 2. 能夠有效率的產生、管理、操作物件。 |
| 參考範例(1) |
| 此範例使用了介面(interface)作為訂定規則的規範，並且建立建立一個獨立的工廠(factory)進行建立物件的依據，工廠負責生產特定需求的物件達到設計需求。   1. 我們設計了一個介面，這個介面專門讀取不同格式的檔案      1. 我們建立了兩個實現這個介面的檔案類型(XML、JSON)    1. JSONLoader =>專門讀取JSON格式      * 1. XMLLoader =>專門讀取XML格式      1. 建立一個Factory      1. 當我們要使用某個類別時，直接使用了工廠來產生這個物件     我們可以從範例中發現，註解的地方為 未使用SimpleFactoryPattern的設計模式，如果我要讀取XML格式內容就必須產生一個新物件。  但是這樣有一個問題就是，如果我今天把XMLLoader類別名稱改成了XALLoader類別名稱的話  那麼範例中的new XMLLoader就會出現錯誤，因為這樣產生了高依賴性。  所以我們統一集中在一個類別(class)當中，並且由這個Factory幫我們產生。  這就是SimpleFactoryPattern的基本範例。  但是以上有一個問題，如果我今天再加入CSVLoader類別(class)的話，我就要去動到Factory當中的getLoader方法(method)，這個行為其實我們違反了SOLID設計原則的OCP原則，所以比較正統的工廠模式有另一種寫法，可以參考 工廠方法模式(Factory Method Pattern)。  我們可以從程式碼中看到  C:\Users\ASUS\Desktop\擷取.JPG  如果今天我們把JSON改成CSV，那麼代表著  C:\Users\ASUS\Desktop\擷取.png  我們必須把LoaderFactory內的getLoader方法進行修改，並且在LoaderType 的enum進行添加CSV相關的程式碼。以上行為就會違法SOLID當中的O(OCP)原則。   1. UML |
| 參考網站 |
|  |

## Factory Method Pattern（創建型模式）

|  |
| --- |
| 簡介 |
| 1. 定義一個建立物件(Object)的介面(inferface)，但讓實現(implements)這個介面的類別(class)來決定實體化哪個類別。(出處:維基百科) 2. 由子類別(subClass)來實現這個方法來建立具體類別的物件。(出處:維基百科) 3. 工廠方法帶來的方便在，無須修改原本的程式碼，在符合OCP原則情況之下，當產品發生改變時，只要在新增功能上去即可。 4. 缺點是只能建立單一類型的功能。 5. 與SimpleFactory不同的是，我們將工廠也一併切分了，而不是一個工廠統一生產多種不同類型物件。 6. 將設計一個產生物件的規範(interface)，讓子類別(sub-class)來實現規範建立物件。 7. 未來進行擴充更容易。(就是在建立一個新的工廠) |
| 適用時機 |
| 1. 建立物件需要大量重複的程式碼。 2. 有效率的產生、管理、操作物件。 |
| 參考範例(1) |
| 1. 我們建立了 TV 的規範，也就是介面(interface)或者抽象(abstract)，範例中使用介面。     這個規範就是，所有的TV都具備開與關的功能。   1. 並且建立了兩個類別來實現這個TV(練習用)。        1. 建立一個工廠的介面(interface)用來規範建立的標準。      1. 並且子類別來實現這個工廠規範。     這裡要特別注意，我們將產生TV的規範切分，不同類型的TV由不同的工廠(Factory)處理。     1. 最後由呼叫端來決定要建立哪一個物件。     範例當中我們決定產生ProductTV\_B 所以我們直接呼叫專門產生ProductTV\_B的工廠(Factory)來進行產生實體動作。   1. UML   C:\Users\ASUS\Desktop\Class Diagram3.jpg |
| 參考網站 |
|  |

## Abstract Factory Pattern（創建型模式）

|  |
| --- |
| 簡介 |
| 1. 提供一個建立一系列相關或相互依賴物件的介面，而無需指定它們具體的類別。 2. 管理有關聯性的物件。 |
| 適用時機 |
|  |
| 參考範例(1) |
| 1. UML     由子類別來產生對應的實體。   1. 假設:Audi和BMW都有生產休旅車，而我的工廠專精生產休旅車，不一定需要會做某品牌全系列的商品。在我想要生產其他車種時只需要抽換具體工廠即可(更換生產的產品)。 2. 定義Audi的介面     定義BMW的介面     1. 定義產品            1. 抽象工廠可以製作某品牌的Jeep或是SUV      1. 不同工廠產生不同廠牌的車種        1. 測試的方式     可以看到，當我們要產生不同的車種，我們可以直接抽換成要產生的工廠，這樣就不會有問題了。 |
| 參考網站 |
| <https://ithelp.ithome.com.tw/articles/10208955> |

## Adapter Pattern(Object Adapter/Class Adapter)（結構型模式）

|  |
| --- |
| 簡介 |
| 1. 又稱之為 轉接器。 2. 將兩個或以上不相容的類別結合在一起使用，屬於結構型模式，需要有Adaptee(被適配者)與Adapter(適配者)兩個身分。 3. Adapter 又區分為Object Adapter 、Class Adapter與Default Adapter三種。 4. 其中Object Adapter 為組合轉接器。靈活性高、耦合低，使用動態組合，但使用複雜、需要實體對象(可以參考範例二使用setter或者建構子)。 5. 其中 Class Adapter 為繼承轉接器。使用方便、代碼簡單，但是高耦合、沒有靈活性低。 6. Adapter Pattern 需要Target 也就是參與者(對象)、Client參與者、Adaptee(被動符合)參與者、Adapter參與者 四種角色。 7. 更好的複用性:系統需要使用到現有的類別，而此類別的接口不符合系統的需求。那麼通過轉接器模式(AdapterPattern)就可以讓這些功能得到更好的複用。 8. 透明、簡單:客戶端可以調用同一個接口，因而對客戶來說是透明的。 9. 更好的擴展性。 10. 解耦性高。 11. 符合開放關閉原則(OCP)。 12. 缺點是 太多的轉接器會造成系統非常混亂，容易搞混。 |
| 適用時機 |
| 1. 當有需要將兩個沒有任何關係的類別組合使用。 2. 系統需要複用某個類別(功能)，但是該類別(功能)不符合現在系統的需求，可以使用。 3. 多個組件功能類似，但是接口不統一。 4. 多用Object Adapter少用Class Adapter，因為…以上已經說過了。 |
| 參考範例(1) Class Adapter(繼承) |
| 1. UML:我們有Adaptee 與 Target，其中Adaptee有SpecificRequest()方法，介面Target有Request()方法。兩者有不相容的方法。   所以今天如果要將兩者做一個結合，就必須使用Adapter Pattern設計模式，建立一個Adapter類別繼承Adaptee與實作Target介面，並且將Target介面中的方法Request()實作呼叫父類別Adaptee的方法。     1. Adaptee 類別如下      1. Target 介面資訊如下:      1. 今天要將以上兩結合再一起，需要建立一個轉接器(Adapter)      1. Client使用方式 |
| 參考範例(2) Object Adapter(組合) |
| 1. UML      1. 使用委託方式，例如     經由建構參數或者一個方法傳入值來處理這個adaptee的SpecificRequest()方法。  第一個範例與第二個範例不同的地方在於，不需要繼承產生依賴，相同的方式建立一個Adapter類別做一個轉接器，並且經由委託方式呼叫Adaptee的SpecificRequest()方法。 |
| 參考網站 |
| <https://blog.csdn.net/carson_ho/article/details/54910430> |

## Bridge Pattern（結構型模式）

|  |
| --- |
| 簡介 |
|  |
| 適用時機 |
|  |
| 參考範例(1) |
|  |
| 參考網站 |
|  |

## Builder Pattern（創建型模式）

|  |
| --- |
| 簡介 |
| 1. Builder Pattern 主要著重在物件產生的過程，一個複雜物件產生過程中，將過程隱藏起來。 2. 與Abstract Factory Pattern的差異在於 Abstract Factory Pattern主要是把重點放在物件建立的結果並且集中管理，而Builder Pattern 主要把重點放在建立物件的過程。 3. 將一個複雜對象的建構和表現分離，使得同樣的建構過程可以產生出不同的表現。 |
| 適用時機 |
| 1. 一個物件建立所需要的參數不清楚或者傳入參數太多導致多個Overload時，可以使用。 2. 簡單而且容易了解，根據需求放入需要的參數，並且成功建立物件。 |
| 參考範例(1) |
| 1. 參考UML:      1. 建立一個JavaBean(圖片中省略了get/set)。      1. 我們建立了一系列的規範，讓用戶產生這個User物件時可以根據自己的需求寫入要寫入的參數。      1. 建立一個中介者(指揮者Director)，並且由中介者來執行產生User物件的過程。       中介者實作了介面的規範。   1. 當我們要建立物件的時候，可以寫入自己需要的參數，而不用依賴建構子。      1. 以上範例重點在於UserBuilder 這個Dicetor這個角色。 |
| 參考網站 |
| <http://corrupt003-design-pattern.blogspot.com/2017/01/builder-pattern.html> |

## Chain of Responsibility Pattern (行為模式)

|  |
| --- |
| 簡介 |
| 1. 有幾個物件都能處理某種請求，但能夠處理的範圍不同，當這個物件沒有處理的權限時，能夠將這個請求，傳遞給下一個物件處理。 2. 效能上會有些許差異(較慢)。 3. 我們可以稱之為 責任鏈 |
| 適用時機 |
| 1.暫時不知道~ |
| 參考範例(1) |
| 1. 我們要將一個字元根據不同的形態(英文字/數字)來進行判斷後做一些事情，那麼一般的做法會是這樣     如果字元為英文字就做特定事物，再判斷如果是數字就做某件事，但如果有一大堆東西要判斷，那麼這個if..else..會判斷不完，但是這樣叫做集中管理分配   1. 我們可以使用Chain Of Responsibility Pattern 設計模式將程式設計成     這是一個父類別，目的就是將這個東西不斷往下丟。   1. 可以建立一個類別專門處理符號相關資訊(這是一個比喻)     然後將這個變數繼續往下丟   1. 我們建立一個類別專門處理字符相關資訊     然後將這個變數繼續往下丟   1. 我們建立一個類別專門處理數字字符相關資訊     也可以繼續往下丟(可以一直寫下一層下一層下一層…)   1. 並且建立由誰先檢查     範例中，我們檢查的順序如上圖，即使更改了DigitHandler順序(與CaracterHandler 交換或者與SymbolHandler交換)也只是順序上的改變而以。  以上的設計模式可能在效能上會有比較慢的情況，但是卻提高了判斷的自由性，加上如果要再增加判斷依據也不需要修改程式，減少耦合發生。 |
| 參考網站 |
| <http://twmht.github.io/blog/posts/design-pattern/chainOfResponsibility.html> |

## Command Pattern(行為模式)

|  |
| --- |
| 定義/簡介 |
| 將請求封裝為物件，使你藉由不同的請求，對客戶端請求參數化，並支援可取消的功能。  將引發命令的物件與實際執行操作的物件隔離開來。 |

|  |
| --- |
| 角色與定位Clinet |
| 負責建立具體命令並組裝接收者。  建立 具體的命令物件 (ConcreteCommand)，並設定其接收者 (Receiver)，此處的 Client 是站在『命令模式』的立場，而非泛指的『客戶』！ |

|  |
| --- |
| 角色與定位Command |
| 負責制定命令使用介面。  『至少』會含有一個 Execute() 的抽象操作 (方法) (abstract operation) 。 |

|  |
| --- |
| 角色與定位ConcrateCommand |
| 具體的命令類別，通常持有 Receiver 物件。 |

|  |
| --- |
| 角色與定位Receiver |
| 負責執行命令的內容，任何能實現命令請求的類別，都有可能當作 Receiver。 |

|  |
| --- |
| 角色與定位Invoker |
| 負責儲存與呼叫命令。  儲存 具體的命令物件 (ConcreteCommand)，並負責呼叫該命令 —— ConcreteCommand.Execute()，若該 Command 有實作 『復原』功能，則在執行之前，先儲存其狀態。 |

|  |
| --- |
| 適用時機 |
|  |

|  |
| --- |
| 參考範例 |
|  |

|  |
| --- |
| 參考文獻 |
| 命令模式(Command Pattern): <https://notfalse.net/4/command-pattern> |

## Command Pattern(行為模式)

|  |
| --- |
| 簡介 |
|  |

|  |
| --- |
| 適用時機 |
|  |

|  |
| --- |
| 參考範例 |
|  |

|  |
| --- |
| 參考文獻 |
|  |