兩大原則

Tuesday, November 1, 2016 11:42

Program to an interface, not an implementation.

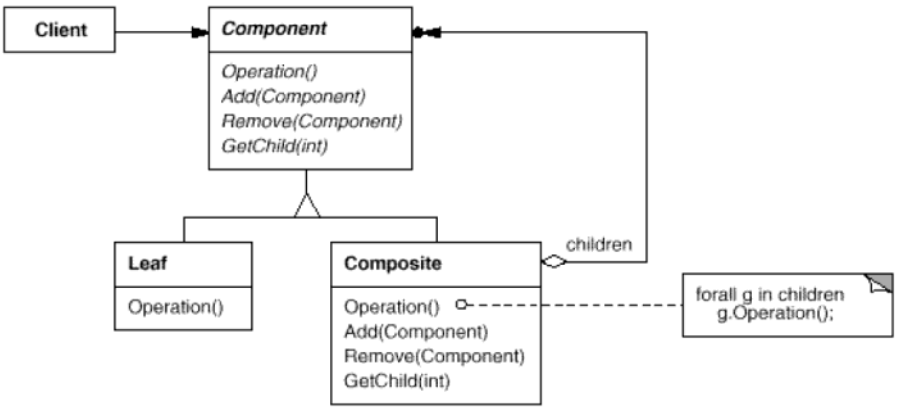
使用者不需知道所用物件的型別，只需要知道物件符合預期的介面即可
使用者不需知道所用物件的類別，只需要知道介面是由哪個抽象類別定義
表示程式不需要在 compile 的時候決定物件的型態,根據子類別回傳的物件再決定型態

Favor object composition over class inheritance

繼承是 OOP 裡最強烈的耦合(Coupling)關係,當你動到父類別時,將影響底下所有的子類別。
繼承的關係越長、越複雜時,所造成的影響就會越大、越難維護。

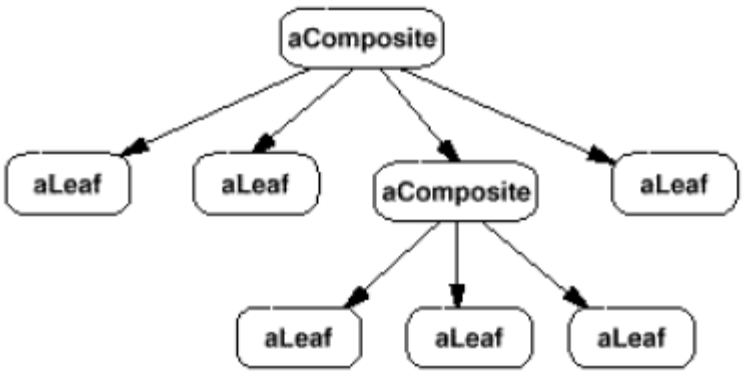
Composite

Thursday, November 10, 2016 17:22



- 表達『部分-全體』關係，讓外界以一致性的方式對待個別物件和整體物件。
- 使用遞迴複合技巧使client不必區分這些物件。
- Composite關鍵在於用一個抽象類別同時代表個別物件與物件容器。
- 無需考慮基本物件與複合物件的差異，以一致的方式處理複合結構裡的物件。
- 容易新增新的composite或leaf類別而不需修改client程式。(Open-Closed)
- Component介面極大化，為了不讓client知道面對的是leaf或composite，Component會盡可能涵蓋
  兩個都可能有的操作。
- Add/Remove是否該擺在component上？
  - 若選擇在 Component 中,對所有的 Component(包含 leaf node 和 composite)可以一視同仁提升通透性,但是卻犧牲了安全性,因為對於 leaf node 而言無法避免被惡意撰寫「新增、刪除」的惡搞程式，所以會在leaf上讓它們的預設版本失敗或者拋出例外。
  - 若選擇在 Composite 中,可以在編譯時期找出「新增、刪除」leaf node 程式的錯誤,但卻失去通透性。

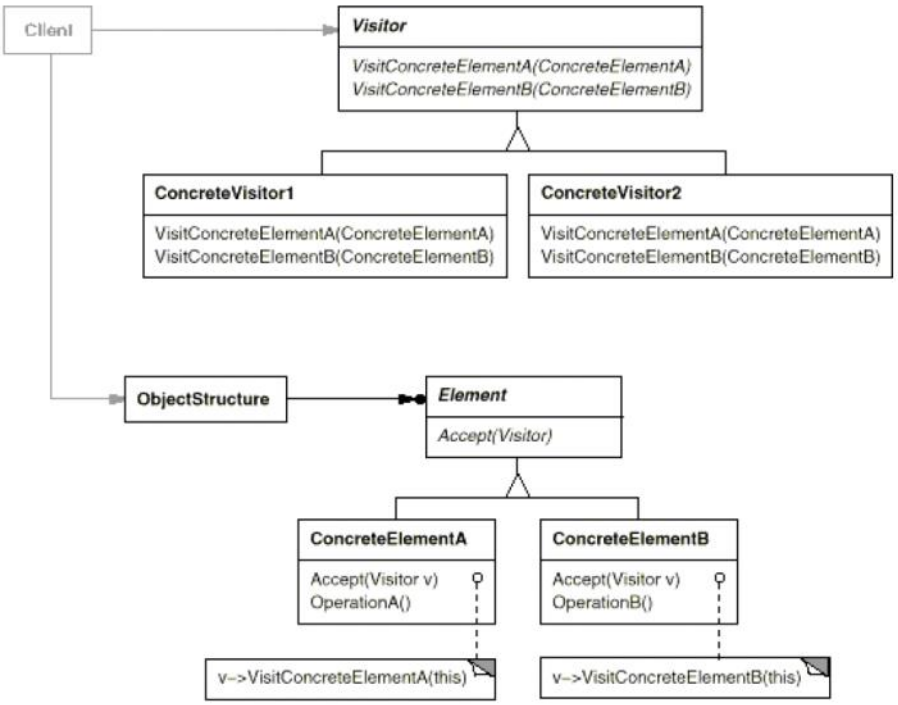
複合物件結構圖



http://www.cnblogs.com/zhenyulu/articles/41829.html

Visitor

Thursday, November 10, 2016 17:35

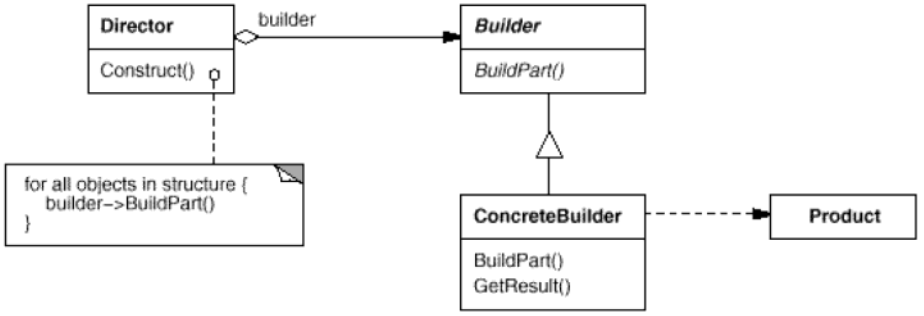


- 不必修改作用對象的類別介面,就能定義新的操作。
- Visitor作法是將Node系列類別的操作全部抽取出來，擺到Visitor物件上，節點在接受這個visitor的時候，會送一個訊息(節點類別資訊和節點本身)給visitor，visitor收到訊息後，便對節點實施操作。
- 當一個物件中的元素幾乎不會異動,但是這些物件的行為常會增減,那麼就適合使用Visitor pattern。
- Visitor可將操作集中在另一個類別上，即使同一份物件結構被許多程式共用，也不會彼此干擾。
- 容易增加新操作，只要新增一個visitor即可輕易增添一項元素相關的操作。
- 難以增添新的ConcreteElement，每增加一個新的，則需要再替Visitor增加一抽象操作，ConcreteVisitor也都要實作。
- 累積狀態，可在拜訪物件結構各個元素時順便累積狀態(Ex: 周長、面積加總)
- 破壞封裝，會迫使公開一些可存取Element內部狀態

http://www.cnblogs.com/zhenyulu/articles/79719.html

Builder

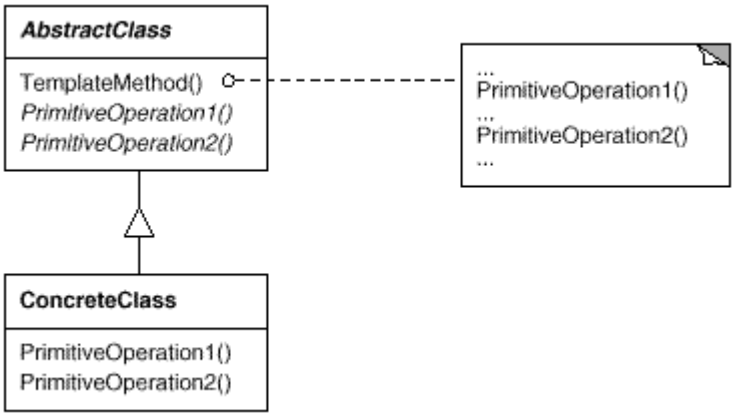
Thursday, November 10, 2016 17:55



- 從複雜物件中取出生成程序,以便在同一個生成程序製造各種不同的物件。
- 若要將『建造複雜物件的演算法』和物件零件與組裝方式保持獨立時使用Builder。
- 若要讓同一個物件生成程序能產生數種不同形式物件時使用Builder。
- 將生成程序與內部布局的程式碼隔離，builder將物件生成及佈局方式封裝起來，提高模組化程度。
- 對生成程序的掌控更細膩，成品物件是在Director的監控下一步步建出來，等到完全建好後，Director才向builder索取成品物件，與其他一個步驟建出成品的生成樣式不一樣。
- 預設情況下，Builder操作只是空殼子，不將builder系列操作宣告成pure virtual，只定義成空殼子操作，只需複寫需要的操作即可，無須全部複寫。

Template Method

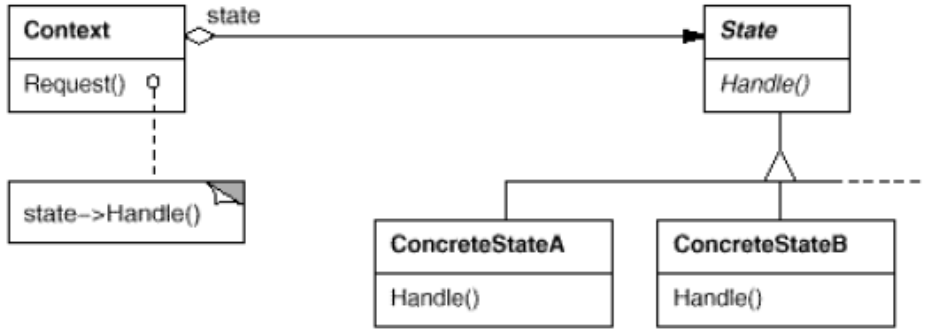
Thursday, November 10, 2016 17:57



- 只定義演算法流程，某些步驟留給子類別做，以便在不改變演算法整體架構。
- TemplateMethod為non-virtual function、PrimitiveOP1、PrimitiveOP2為virtual function。
- Non-Virtual該不該不Override?不該!若被Override則無法確保流程一致，Virtual function 被Override是為了Polymorphism。

State

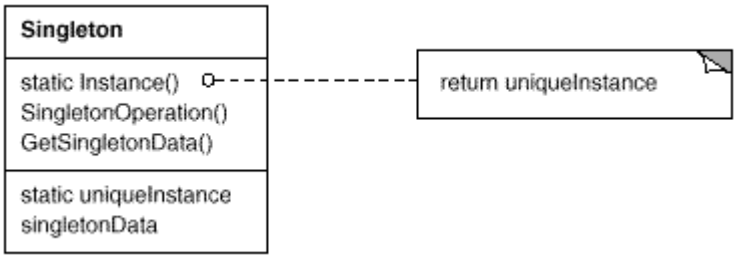
Friday, November 18, 2016 19:29



- 讓物件的外顯行為隨內部狀態的改變而改變，如同類別也改變了。
- 當物件行為取決於它的狀態，連執行期行為也隨狀態改變時使用。
- 集中處理與狀態相依的行為並切割。
- 凸顯狀態轉移邏輯。
- 狀態物件可以共用。

Singleton

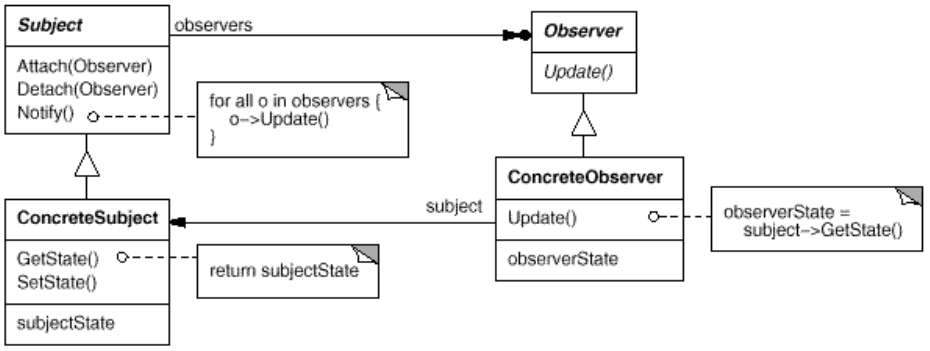
Sunday, December 11, 2016 19:41



- 確保類別只會有一個物件實體存在，並提供單一存取窗口。
- 定義Instance()操作讓外界存取唯一的物件個體。
- 掌握此為一個體所有存取動作，Singleton 將此唯一的個體封裝起來。
- Static變數只有在第一次使用時初始化，之後在使用時只會直接使用，故確保只產生一個物件。

Observer

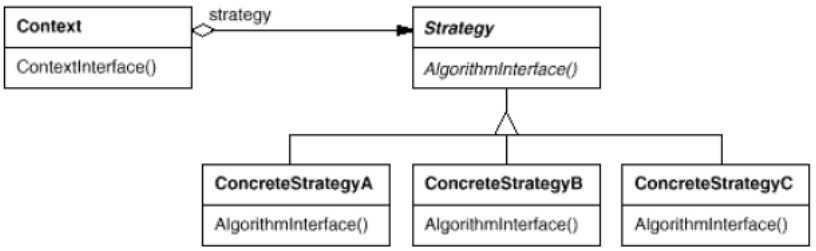
Sunday, December 11, 2016 20:23



- 定義一對多的物件依存關係，讓物件狀態有變動時，就會自動通知其他相依物件做更新動作。
- 誰負責觸發Update()?
  a. 設定在Subject狀態改變後呼叫Notify()，優點是Client不需自己去呼叫Notify，但如果一連串的狀態改變，則會引發一連串的Notify()，非常沒有效率。
  b. 讓Client在適當時機呼叫Notify()，優點是可以等到一連串的狀態改變完成後再呼叫Notify()，但Client得需負責觸發Update程序，很容易忘記去呼叫Notify()。
- 使用時機：當某個主題 class 變動時需要通知其它觀察者 class，而且有可能需要動態增減通知的對象。
- 可降低耦合度，但代價是Message length變長。

Strategy

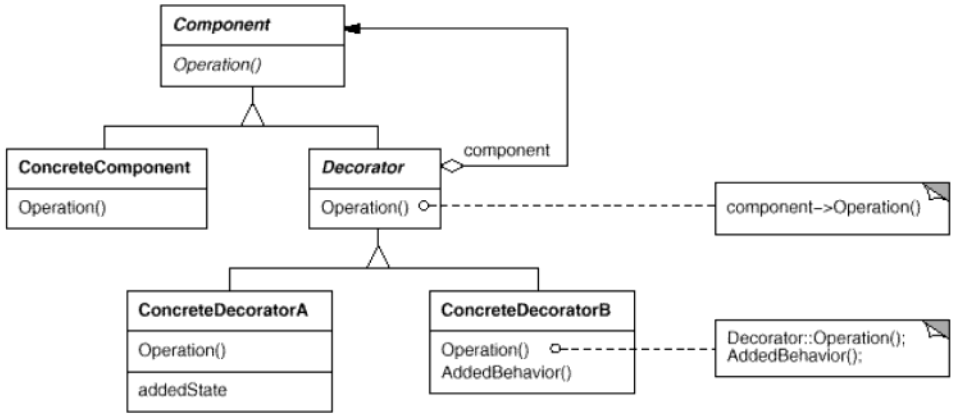
Friday, November 25, 2016 11:04



- 可以定義一組相關演算法，將每個演算法封裝起來，可互換使用，可在不影響外界的情況下個別抽換所用的演算法。
- 使用時機：不同的演算法各有不同的適用時機時。(例如：文章分段演算法，用類別來封裝各樣的分行演算法，每個被封裝的演算法都是個Strategy)
- 使用時機：如果一個類別裡定義多種行為，且這些行為都是以條件判斷指令來切換時，最好把每條分支搬到個別的Strategy類別身上。
- Strategy會增加程式的物件數量，如果把Strategy做成不含狀態的物件，就可給Context使用，至於狀態資訊則由Context維護，必要時才當參數交給Strategy物件操作，可共用的Strategy不能含有狀態資訊。
- Strategy與Context之間通訊負擔，不管Concrete Strategy類別實作的演算法是簡單還是複雜，它們都共用一個Strategy介面，因此有些操作對Concrete Strategy可能是多餘的。
- Strategy可針對同一行為提供多種不同實作。

Decorator

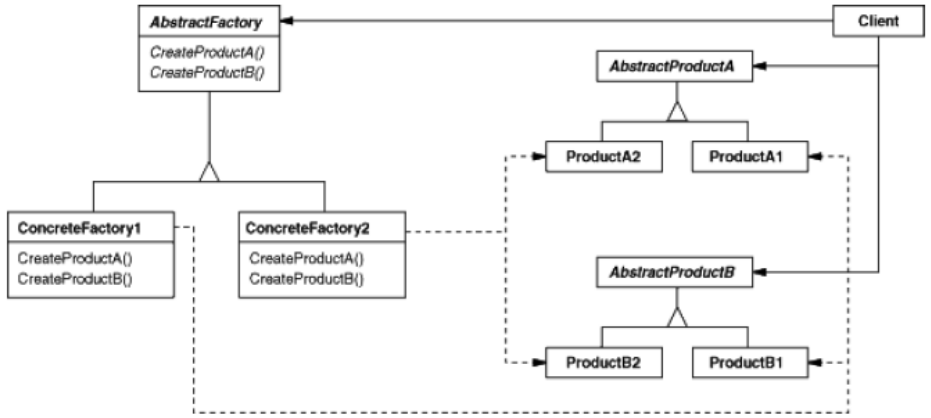
Tuesday, January 3, 2017 17:07



- 將額外權責附加於物件上，不必衍生子類別即可彈性動態擴增功能。
- 使用時機：想動態且透明添加額外權責到個別物件身上，且不影響其他物件。
- Decorator比靜態(多重)繼承更有彈性，只要隨時把Decorator掛上去或拿掉，就在執行階段動態增刪權責。
- 修飾類必須和原來的類有相同的介面。
- 再不更改程式的狀態下，擴充功能。
- 若Component類別已經太過臃腫，導致Decorator負擔過重，改用Strategy會比較好，Strategy中Component會將部分行為丟給另一個獨立的Strategy物件負責，只需替換這個Strategy就可以改變或擴充這個Component功能。
- 優點
  - 提供比繼承更多的靈活性。
  - 使用不同的具體物件，及裝飾物件的排列組合，可以創造出許多不同的行為組合。
- 缺點
  - 進行系統設計的時候會產生很多小對象。
  - 靈活的特性，也代表比繼承容易出錯，除錯也比較困難。

Abstract Factory

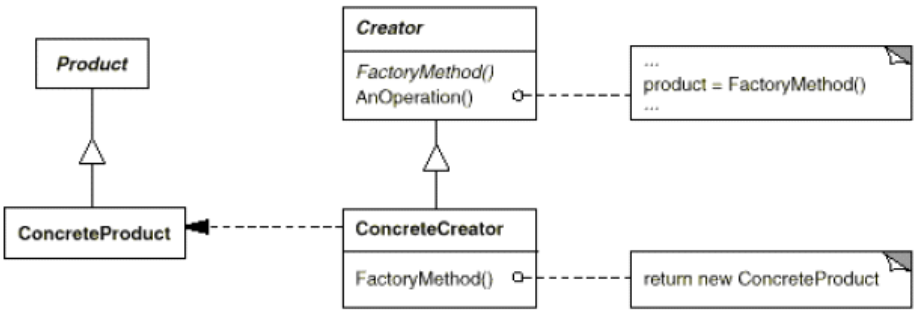
Tuesday, December 20, 2016 19:20



- 以同一個介面來建立一整族相關或相依的物件，不須點名各物件真正所屬的具象類別。
- 使用時機：當系統必須要和最終成品的生成、組合、表達方式保持獨立時。
- 與具象類別隔離開，Abstract Factory讓你掌握程式所產生的物件類別，因為Factory將建立成品物件的程序與責任封裝起來，Client隔絕在真正的實作類別之後，只能透過抽象類別來操作物件。
- 增進成品物件的一致性，確保程式同一時間只會使用某一物件陣營。
- 難以提供新的成品物件種類，讓Abstract Factory能產生新的Product類型並非容易的事，因為它的介面將能產生哪幾種Product寫死，想要擴充就必須修改Abstract Factory類別及所有子類別。
- 除了產生物件外還需要限制物件間關係(要同類型)。
- Abstract Factory在修改時，要付出的代價極高，若新增Subclass時，也須修改Parent Class(同Visitor) 。

Factory Method

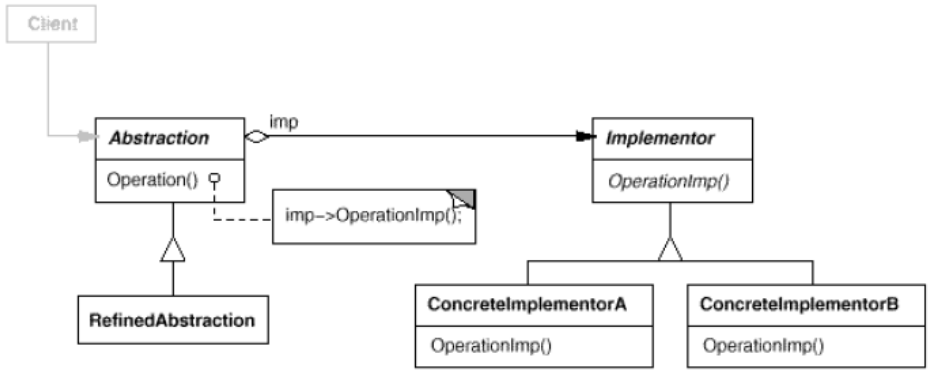
Tuesday, December 20, 2016 21:00



- 定義可生成物件的介面，但讓子類別去決定該具現出哪一種類型的物件。此模式讓類別將具現化程序交付給子類別去處置。
- 使用時機：當類別希望讓子類別去指定欲生成的物件類型時。

Bridge

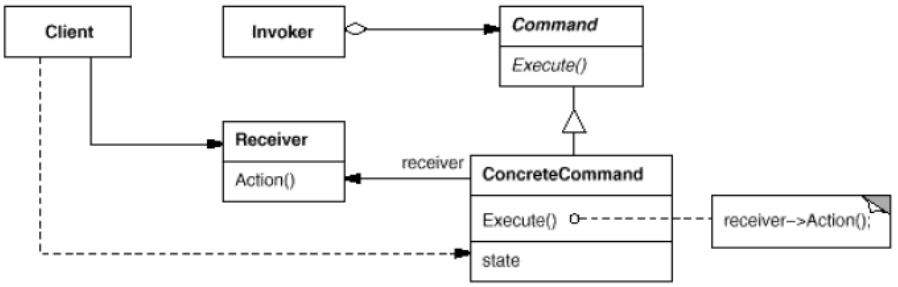
Friday, December 2, 2016 14:47



- 將實作體系與抽象體系分離開來，讓兩者能各自更動各自擴充。
- 使用時機：避免將抽象體與實作體綁死再一起、希望能用子類別去擴充抽象體與實作體。
- 將抽象體的實作方式完全隱藏起來不讓外界知道，外界只看到類別的介面。

Command

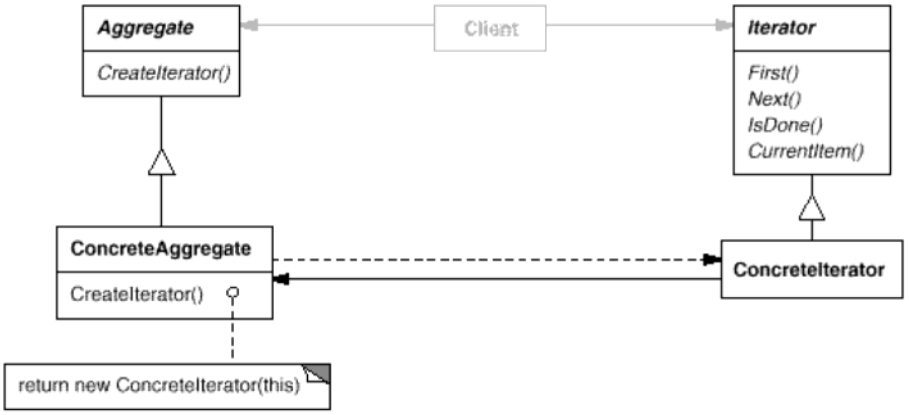
Friday, December 2, 2016 14:48



- 將訊息封裝成物件，以便能用各種不同訊息、記錄復原等方式加以參數化處理
- 使用時機：支援復原功能，Command的Execute可儲存狀態以備復原，也可透過Unexecute抵銷前一次的效果。
- 避免復原過程中不斷累積錯誤，命令一在執行、取消、再執行、再取消，若有小錯誤會逐漸累積成大錯誤，因此有必要再Command裡多存一點資訊，確保物件能回到最初的狀態。
- 優點
  - 降低對象之間的耦合度。
  - 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。
  - 調用同一方法實現不同的功能
- 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。因為針對每一個命令都需要設計一個具體命令類，因此某些系統可能需要大量具體命令類，這將影響命令模式的使用。

Iterator

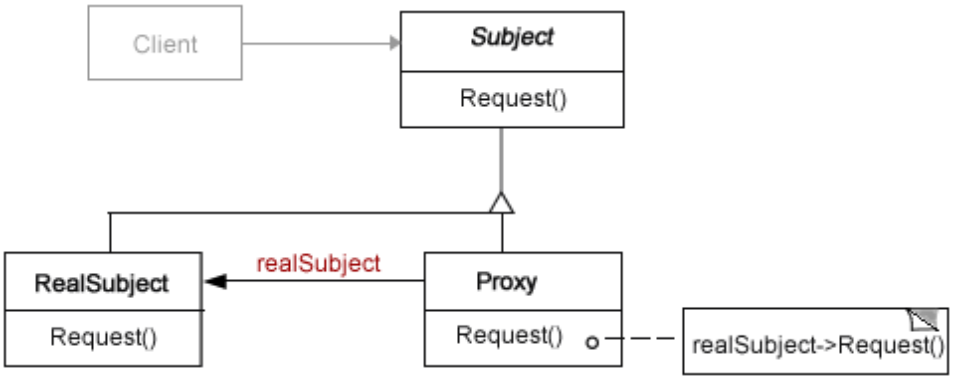
Tuesday, December 20, 2016 19:21



- 不須知道Composite物件的內部細節，即可依序存取內部的每一個元素。
- 可自行定義尋訪樹狀結構的操作(前序、中序等等)。
- 使用時機：想用多種方式尋訪Composite物件。

Proxy

Tuesday, January 3, 2017 17:07

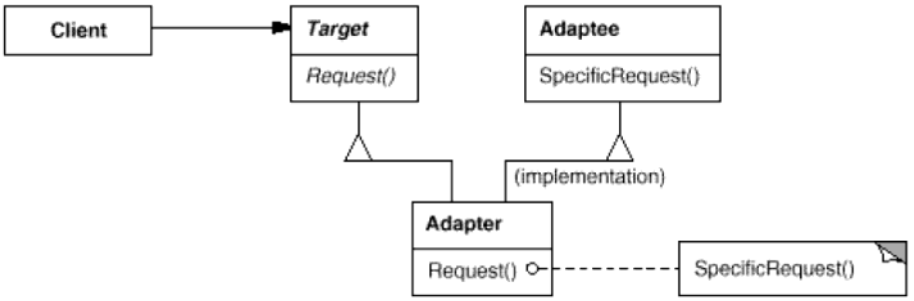


- 替其他物件預留代理者空位，藉此控制存取其他物件。
- 延緩生成及初始化物件所需付出的成本，等到真正**需要時**才將耗資源的物件建立起來。(Virtual Proxy)

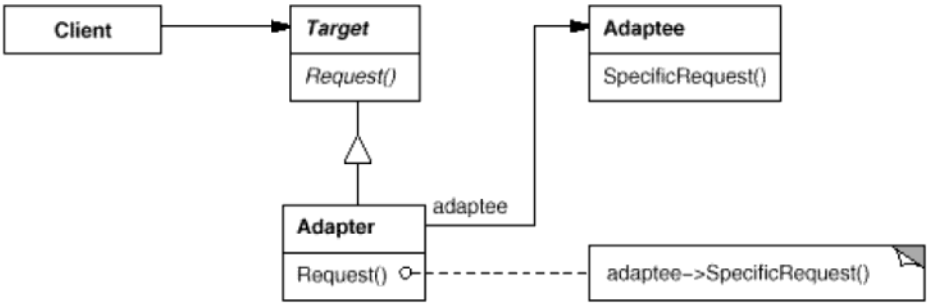
Adapter

Tuesday, January 3, 2017 17:06

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



An object adapter relies on object composition:



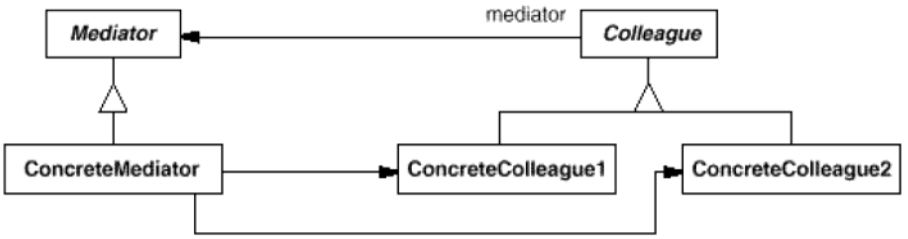
- 將類別的介面轉換成外界所預期的另一種介面，讓原先介面不相容問題而無法使用的類別可以合作使用。
- 使用時機：想利用現有的類別，但它的介面與你所需不符。
- Class Adapter
  - 優點：
    - 容易override Adaptee原本的行為。
      因為Class Adapter繼承了Adaptee，所以可以輕易的override Adaptee。
    - 代碼較精簡。
  - 缺點：
    - 只能轉換單一Adaptee。
    - 無法動態改變欲轉換的Adaptee。
      因為使用繼承技術，在compile-time已經決定了要繼承的Adaptee，所以無法動態改變Adaptee。
- Object Adapter
  - 優點：
    - 可轉換多個Adaptee。
      因為使用了組合技術，配合polymorphism(多型/多態)，所以能轉換class和其derived class。
    - 可動態改變欲轉換的Adaptee。
      因為使用了組合技術，可以在run-time的改變欲轉換的Adaptee。
  - 缺點：

        - 難override Adaptee原本的行為。

- Class Adapter和Object Adapter優缺點剛好互補，可依實際需求決定之，大體上而言，Object Adapter優於Class Adapter，因為彈性較大，且可面對將來未知的class，也應證了那句『**Favor object composition over class inheritance**』的Design Pattern真言。

Mediator

Saturday, January 7, 2017 14:45



- 定義可將一群物件互動方式封裝起來的物件，因為物件彼此不互相指涉，所以耦合性低，容易逐一變更互動關係。

S.O.D.

Thursday, January 5, 2017 17:59

SRP	The Single Responsibility Principle	A class should have one, and only one, reason to change. 一個class只有一個責任/原因。 If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the ability of the class to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed. (不會把Model的code寫在View，否則View會有兩種修正的原因；例如把操作DB的Code寫在View時，View會有變更介面及修改DB操作的兩種作用力。)
OCP	The Open Closed Principle	You should be able to extend a classes behavior, without modifying it. (藉由「新增程式碼」來擴充系統功能,而不是「藉由修改原本已經存在的程式碼」來擴充每個功能。) Software entities (class, modules, functions, etc.) should be open for extension, but closed for modification. 一個軟體個體應該要夠開放使得它可以被擴充,但也要夠封閉以避免不必要的修改。希望達到藉由「新增程式碼」來擴充系統功能,而不是「藉由修改原本已經存在的程式碼」來擴充每個功能。
DIP	The Dependency Inversion Principle	Depend on abstractions, not on concretions.(Like OCP) 1. 高層模組不應該依賴低層模組，兩個都應該依賴抽象。 2. 抽象不應該依賴細節，細節應該依賴抽象。 針對介面程式設計，而不要對實現程式設計。

Other

Tuesday, January 10, 2017 13:09

Pointer被打包起來稱Iterator
Algorithm被打包起來稱Strategy
Constructor被打包起來稱Abstract Factory、Builder
Request被打包起來稱Command

Decorator改變外部
Strategy改變內部
Decorator介面不變想加行為
Adapter行為不變想改介面

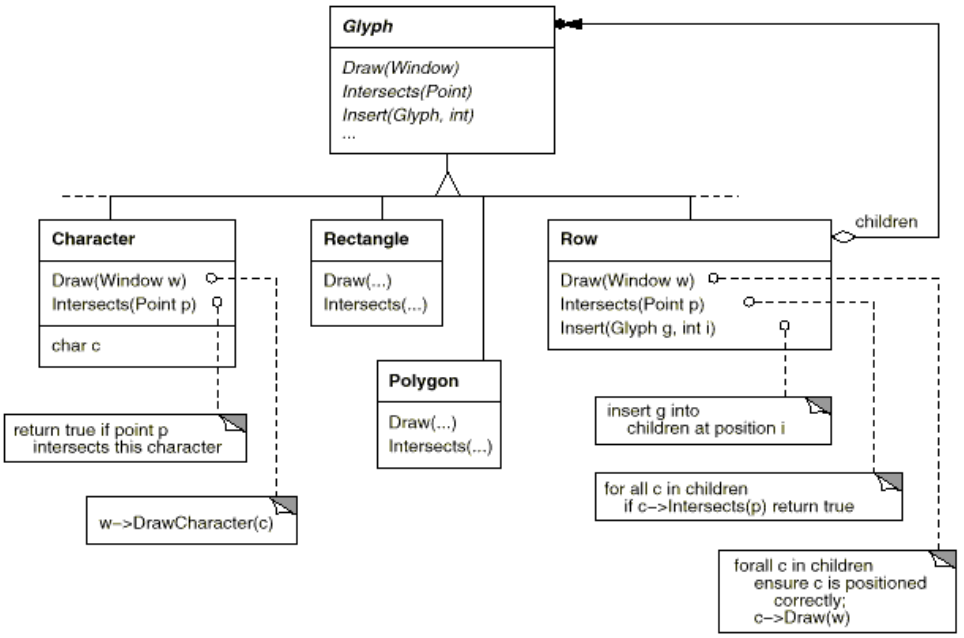
解構元加Virtual原因是要delete子類別時也要將介面一起delete掉

Class Circle: Public Shape (完整繼承介面)
Class Circle: Private(Protected) Shape (繼承者知道，但外界看不到Method)

104_Final

Thursday, January 12, 2017 15:12

1. 沒教
2. (我是覺得不會考,期中範圍)Pattern name： Composite
   Class diagram：

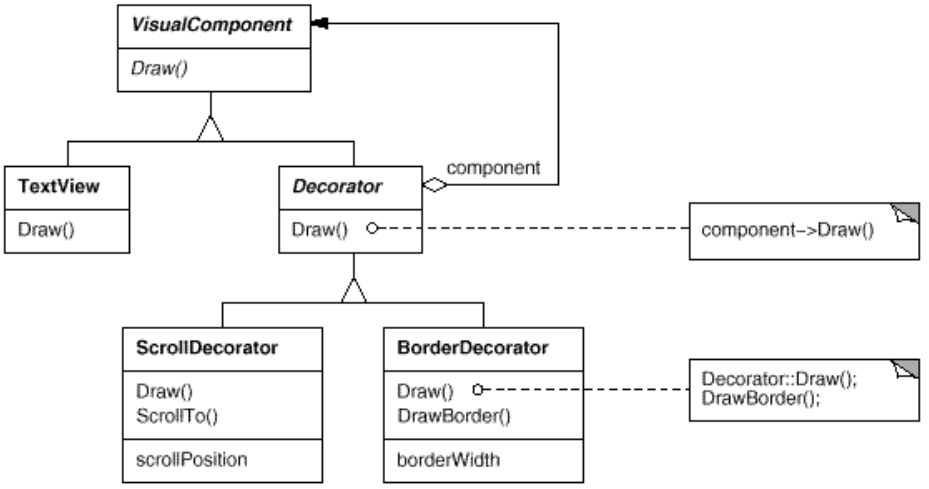


3. 與Composite的Add一樣的意思，參考作業中的測試即可

```
void Insert ( Glyph* g, int i ) {
    glyphs.insert ( glyphs.begin() + i, g )
    // glyphs is a vector<Glyph*>
}

Glyph* g = new Character ();
Insert ( g, 3 );
CHECK ( g->child(3) != NULL )
```

4. Pattern name： Decorator
   Class diagram：

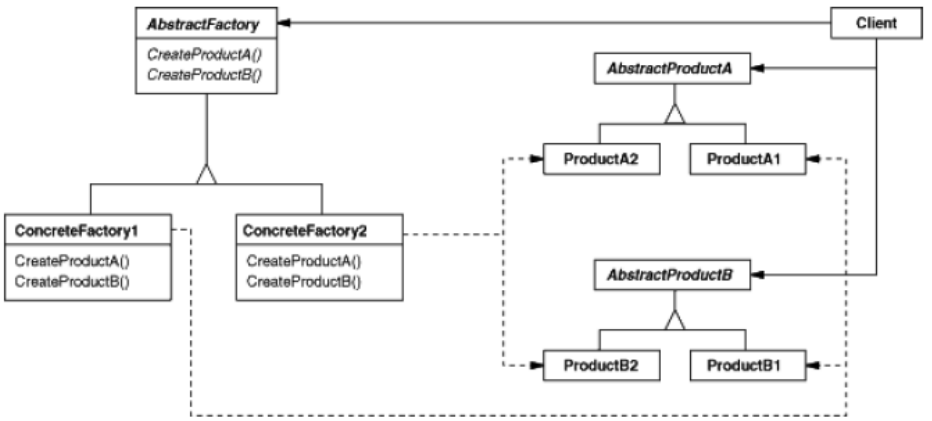


5. 期中範圍

(b) 如果想要新增新的 visit 動作，只需要再加一個 visitor 子類別即可 (open for extension)
不用動到任何 Glyph 已經寫好的 class (closed for modification)

6. Border、Scroller不應該被拜訪，拋例外。
7. 期中範圍
8.

a.



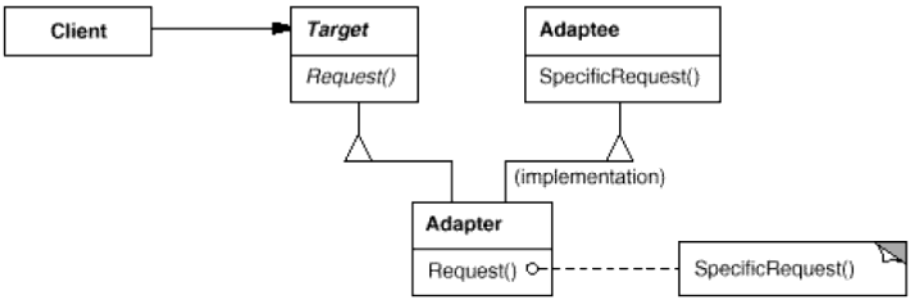
以介面來建立一整族相關或相依的物件，不須點名各物件真正所屬的實作類別。

DIP: 1. High-level modules should not depend on low-level modules. Both should depend on abstractions
2. Abstraction should not depend on details. details should depend on Abstraction

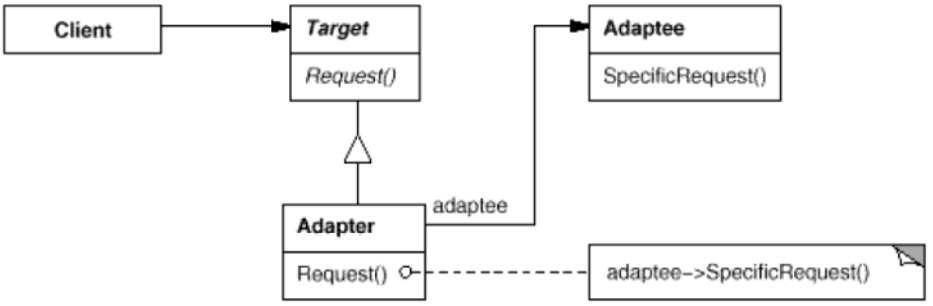
b.

把 client 端與實際產生的物件隔離，都透過 AbstractFactory 去操做，以降低偶合

9. Class Adapter



Object Adapter



Object adapter比較好， **Favor object composition over class inheritance**

繼承是 OOP 裡最強烈的耦合(Coupling)關係,當你動到父類別時,將影響底下所有的子類別。繼承的關係越長、越複雜時,所造成的影響就會越大、越難維護。