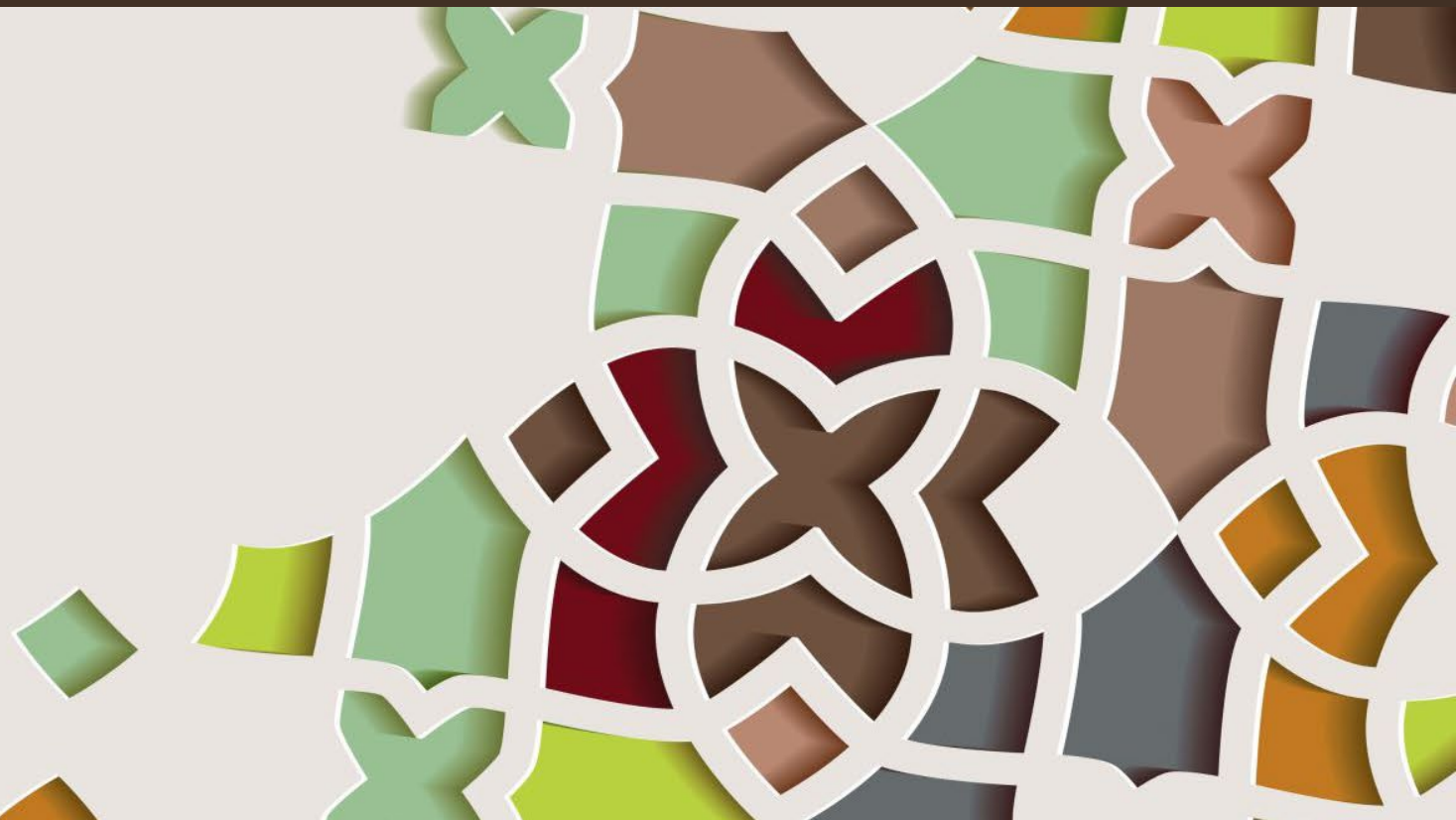
Design Pattern

Lecture 04



回顧

- Singleton的5種實現
  - 內部靜態類
- 3種原則
  - 開放-關閉原則
  - 最少知識原則/迪米特原則
  - 好萊塢原則
- 6種模式
  - 模板模式
  - 外觀模式
  - 適配器模式
  - 命令模式
  - 代理模式
  - 橋接模式

本回

- 2種原則
  - 接口隔離原則
  - 里氏替換原則
- 其它
  - C# 值類型 vs 引用類型
    - string是引用類型?
- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式

原則

- 接口隔離原則 (Interface Segregation Principle, ISP)
  - 要求儘量將臃腫龐大的接口拆分成更小的和更具體的接口，讓接口中只包含客戶感興趣的方法。
- 里氏替換原則 (Liskov Substitution Principle, LSP)
  - 假設q(x)是關於類型為T的對象x的可證明性質，那麼q(y)對於類型為S的對象應該成立，其中S是T的子類型。
  - 子型態必須遵從父型態的行為進行設計。

里氏替換原則 (Liskov Substitution Principle)

- 由Barbara Liskov在1987年在一次會議(資料的抽象與層次)的演說出首先提出。
  - Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T. – **Barbara Liskov**
  - 衍生類別(子類)物件可以在程序中代替其基础類別(基類)物件 **- Robert Martin**
  - 主要說明了有關繼承的一些原則，也就是甚麼時候應該使用繼承，甚麼時候不應該使用繼承。
- 假設有一個模組P，P裏有Car類的物件。當利用新的類(例: Benz)代替Car，而P的功能不會被影響的話，那麼就說Benz就是Car的子類。
- 也就是說，只要S跟T替換後，整個P的行為都沒有差別，那麼S就是T的子型態。從類上來看，S完全可以繼承T，成為T的子類。

里氏替換原則（Liskov Substitution Principle）

- → **通俗來講:** 子類可以擴展父類的功能，但不能改變父類原有的功能。也就是說，子類繼承父類時，除添加新的方法完成新增功能外，儘量不要覆蓋父類的方法。
  1. 子類可以實現父類的抽象方法，但不能覆蓋父類的非抽象方法。
  2. 子類可以擴展自己的方法。
  3. 子類的前置條件不能被加強: 參數不能超過父類的可控範圍
  4. 子類的後置條件不能被削弱: 輸出範圍不能小於父類的輸出
- → **結論:** 保證了父類的複用性，同時也能夠降低系統出錯誤故障，防止誤操作，同時也不會破壞繼承的機制。
- → 繼承不要隨意使用。因為繼承是依賴性超強的一個特性。

Q&A

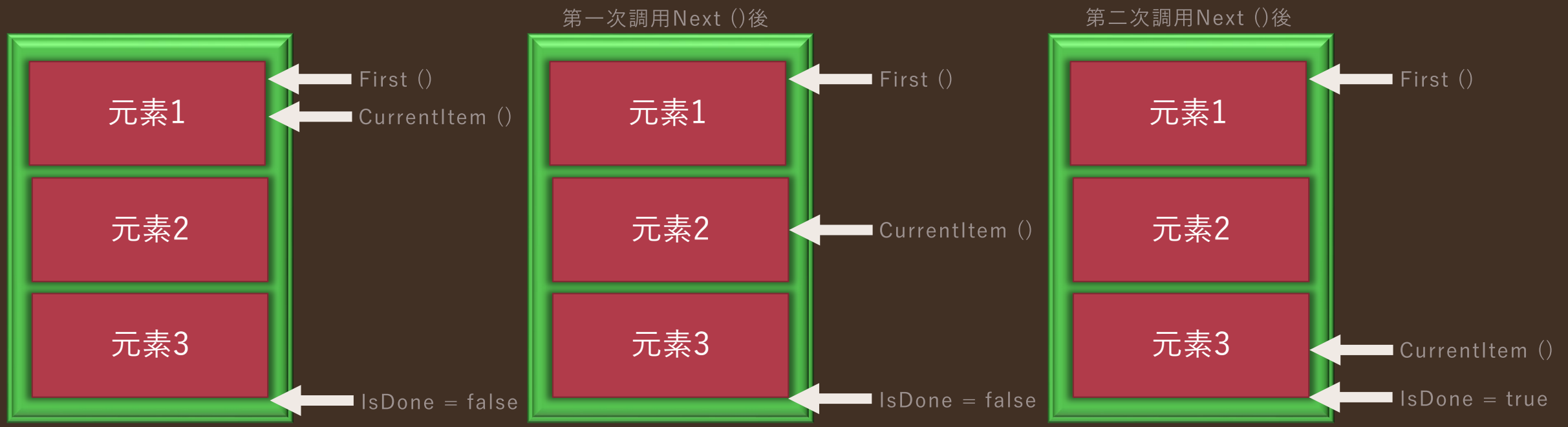
模式

- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式

迭代器

- 可以遍歷訪問容器中的物件。
- 首先會有一個數組(Aggregate)，可以利用迭代器(Iterator)，遍歷訪問這個數組中的元素。
- 迭代器一般會有
  - First ()：獲得首元素
  - Next ()：移動到下一個元素
  - IsDone ()：是否已經到達最後的元素
  - CurrentItem ()：取得當前所指向的元素

迭代器


元素1
元素2
元素3
First ()
CurrentItem ()
IsDone = false
第一次調用Next ()後
元素1
元素2
元素3
First ()
CurrentItem ()
IsDone = false
第二次調用Next ()後
元素1
元素2
元素3
First ()
CurrentItem ()
IsDone = true

迭代器模式（Iterator）

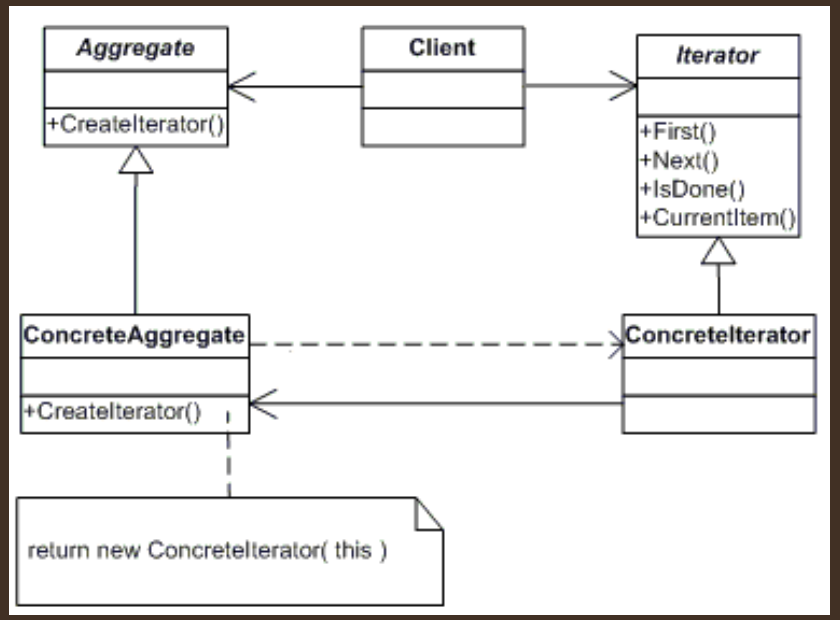
→ 提供一種方法順序訪問一個聚合對象中的各個元素，而不暴露其內部的表示。

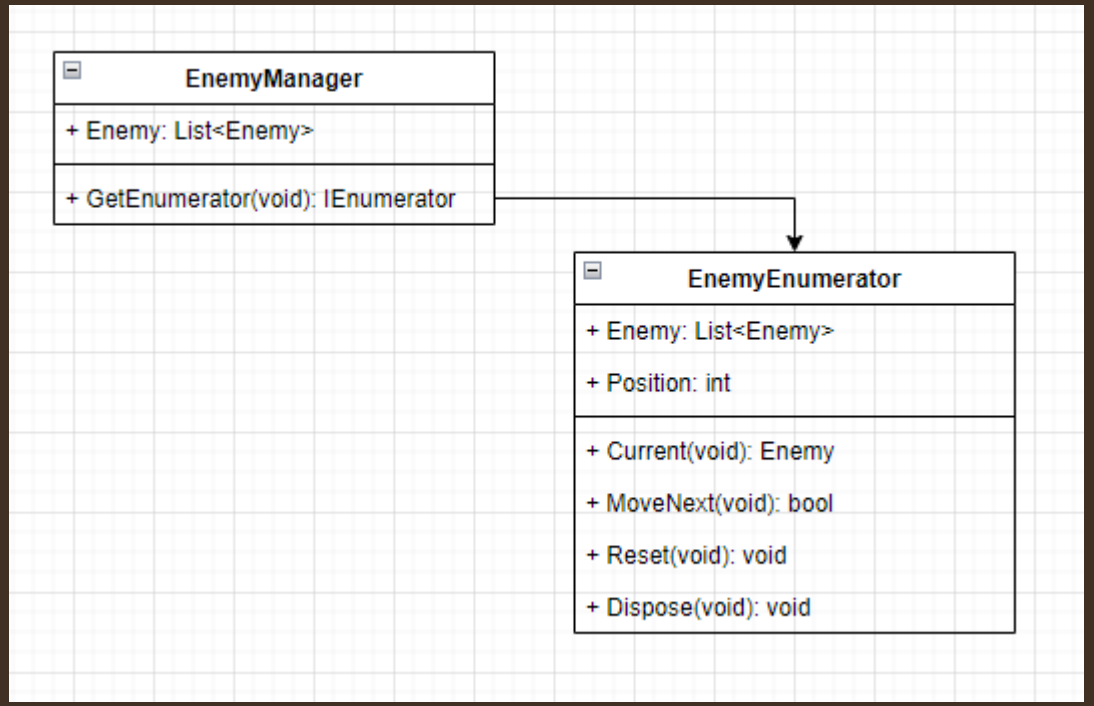
→ C#中提供了IEnumerator和IEnumerable。

→ 在C#2.0以上的版本中，在IEnumerable中提供了yield return的關鍵字，會自動創建IEnumerator而不需要自己實現IEnumerator。

→ 在C#中，只有實現了IEnumerable的類才能被foreach使用。

迭代器模式（Iterator）



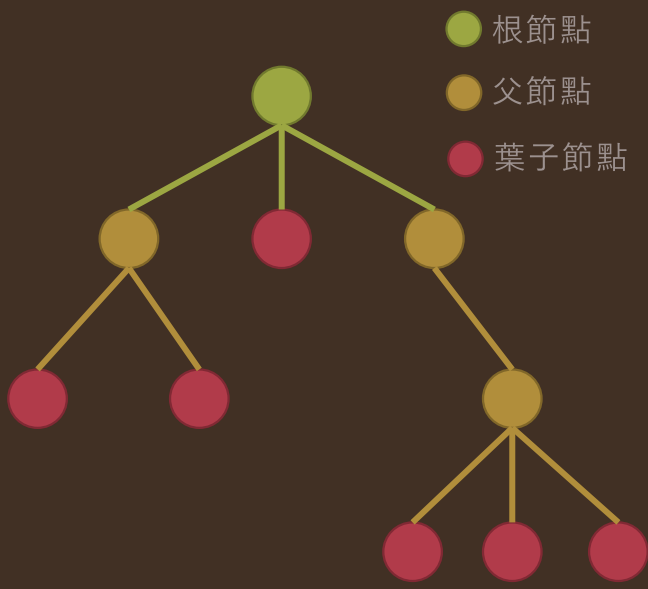


模式

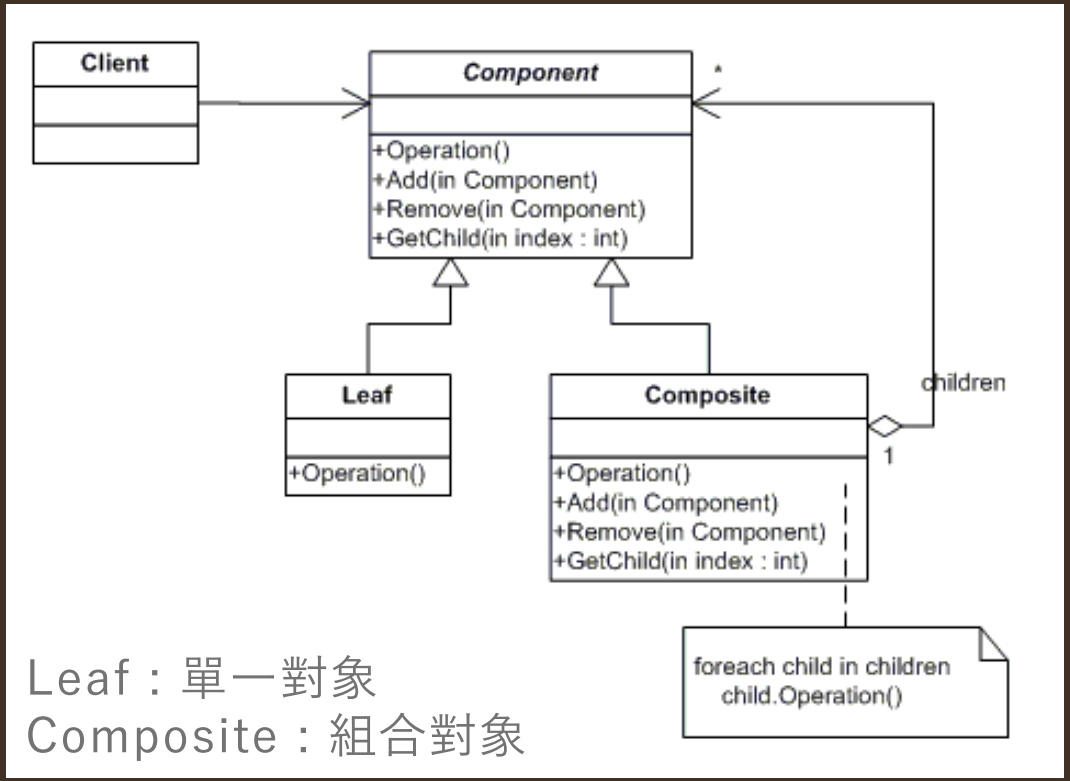
- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式

組合模式（Composite）

- → 允許你將對象組合成樹形結構來表現”整體/部分”層次結構。組合能讓客戶以一致的方式處理個別對象以及對象組合。
- → 在樹中的節點，可以是其它節點的父節點(Node)，也可以是葉子節點(Leaf)。在組合模式中，葉子和父節點都視為同一種「實例」。這樣一來，用戶在處理節點時，就不需要判斷當前節點是父節點還是葉子節點。
- → 組合模式經常與迭代器模式一起使用。利用迭代器來遍歷樹。



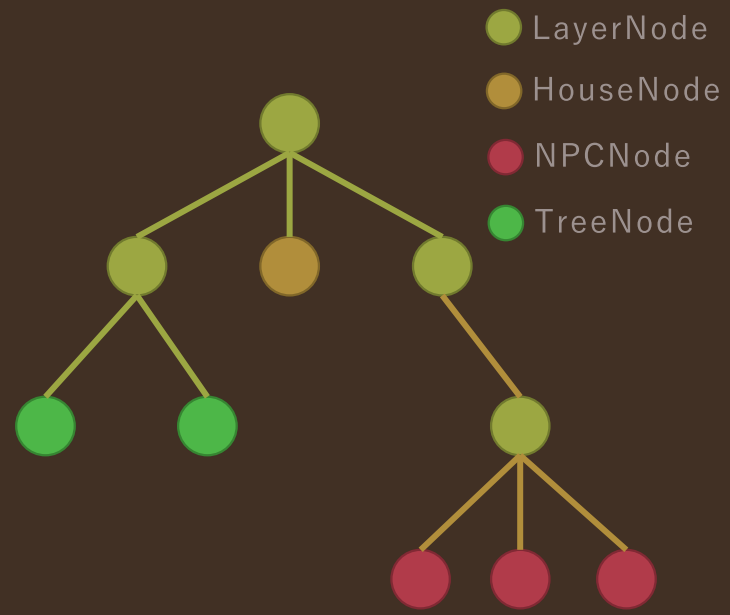
組合模式（Composite）



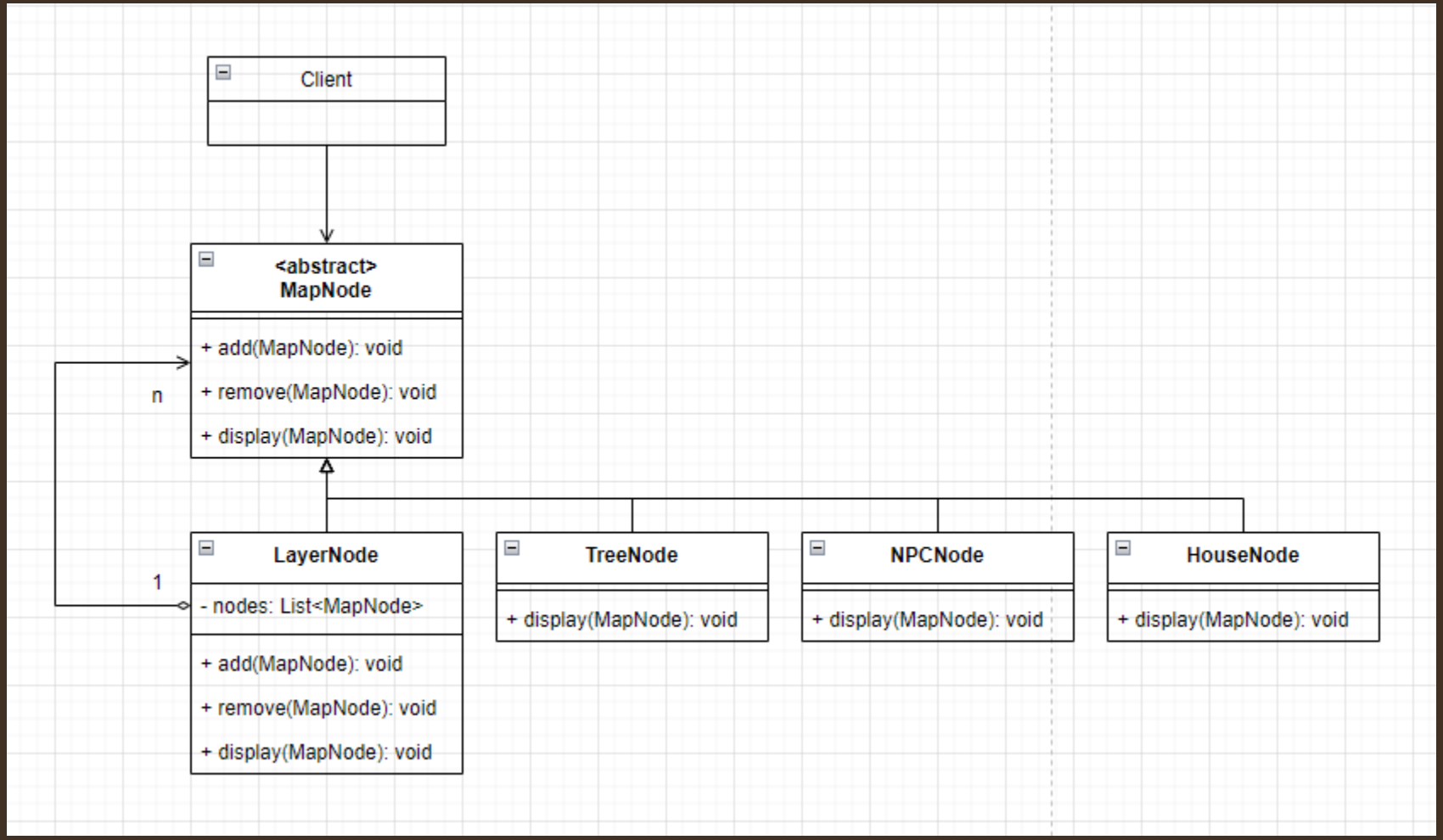
在用戶看來，操作一個Leaf和一個Composite是沒有分別的，都是調用Operation ()。但是在Composite中的內部實現，實際上會調用他的所有子節點的Operation ()。

組合模式（Composite）

- 例子（MapNode）
  - Leaf: TreeNode, NPCNode, HouseNode
  - Composite: LayerNode
  - 有3個NPCNode, 2個TreeNode, 1個HouseNode, 4個LayerNode（包含Root）



組合模式（Composite）



組合模式（Composite）

- 透明方法 vs 安全方法
  - 因為在Leaf中，add和remove都是沒有用的。但為了客戶的使用一致性，導致Leaf擁有add和remove的方法 – **透明方法**。
  - **透明方法**的好處在於，Leaf和Composite的操作都是一樣的，不需要用戶判斷當前操作的對象是哪一種，從而再選擇要執行的方法。
- 實際上破壞了單一責任原則(SRP)。

*組合模式 & 迭代器模式

→ 只要與遍歷有關，首先就可以想到迭代器。

Q&A

模式

- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式

中介者/調停者模式（Mediator）

→ 用一個中介對象來封裝一系列的對象交互。中介者使各對象不需要顯示地相互引用，從而使其耦合松散，而且可以獨立地改變他們之間的交互。

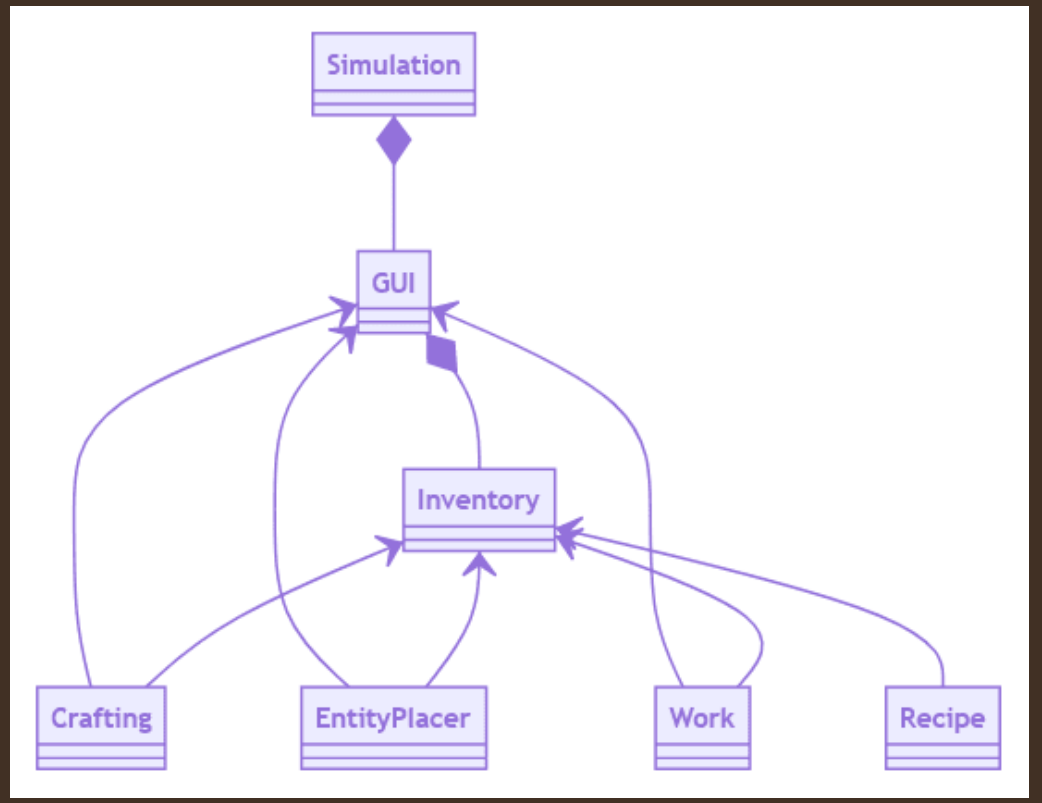
→ 當對象之間有很複雜的關系時，可以用Mediator封裝這些關係。但是Mediator容易變得複雜。

中介者/調停者模式（Mediator）

→ 參考資料:
https://www.gdquest.com/tutorial/godot/design-patterns/mediator/

→ 問題

在玩家背包裏會與entity placement system, crafting system, work system, recipe system交互。每一個系統都可以任意訪問並編輯背包中的插槽的屬性。當你一個人進行開放時，可能沒有問題，但當是一群人一起工作時，這樣就容易出現錯誤。因為系統與背包與GUI的耦合度太高。

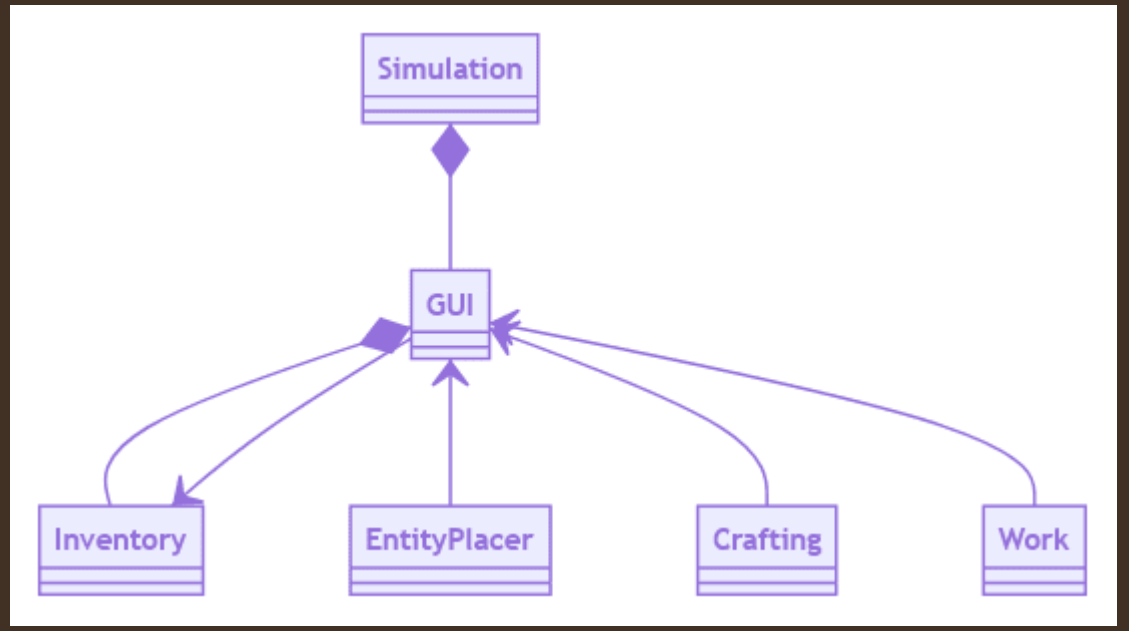


中介者/調停者模式（Mediator）

→ 解決方案

可以讓一個對象作為訪問背包的”中間人”，並逼使其它系統通過”中間人”，間接與背包溝通。

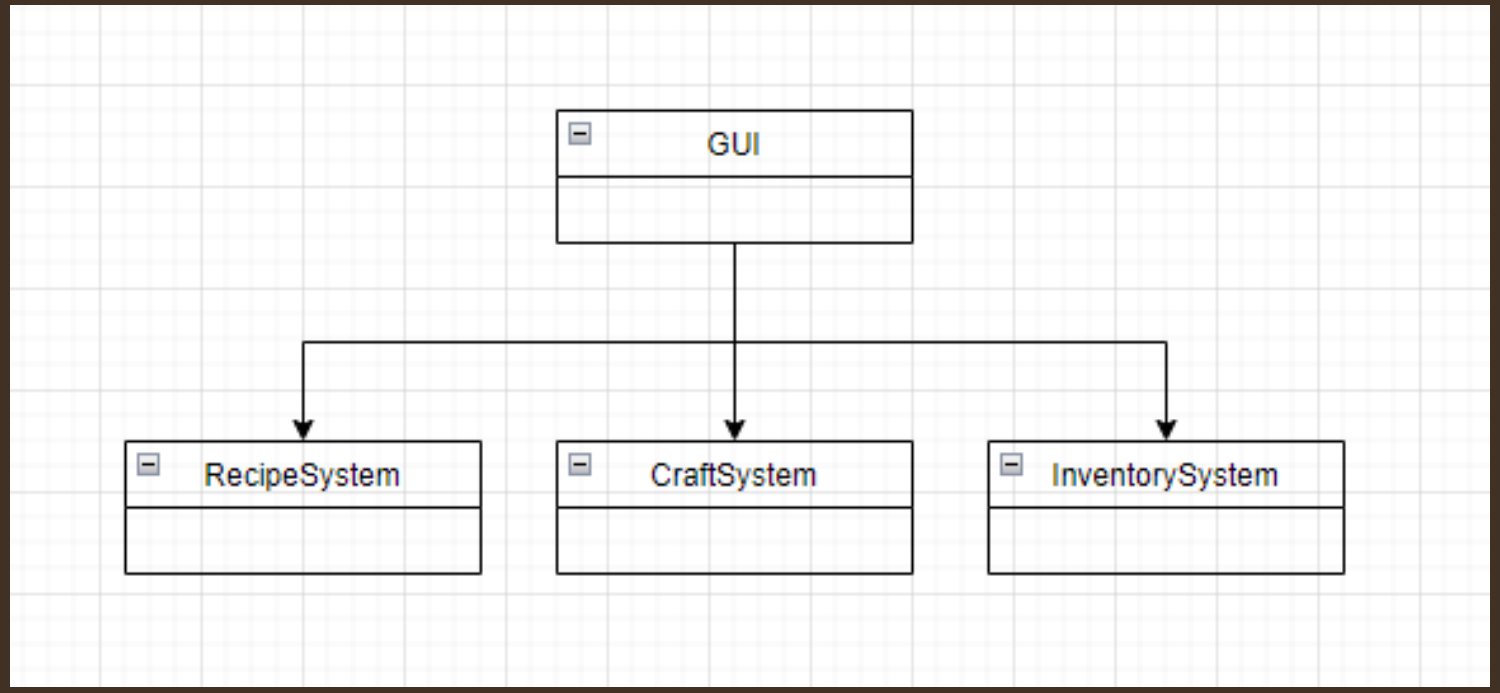
如右圖所示，GUI作為”中間人”，令其它系統與背包解耦。在這種情況下，系統不能直接與背包交互。只能通過GUI來訪問背包。



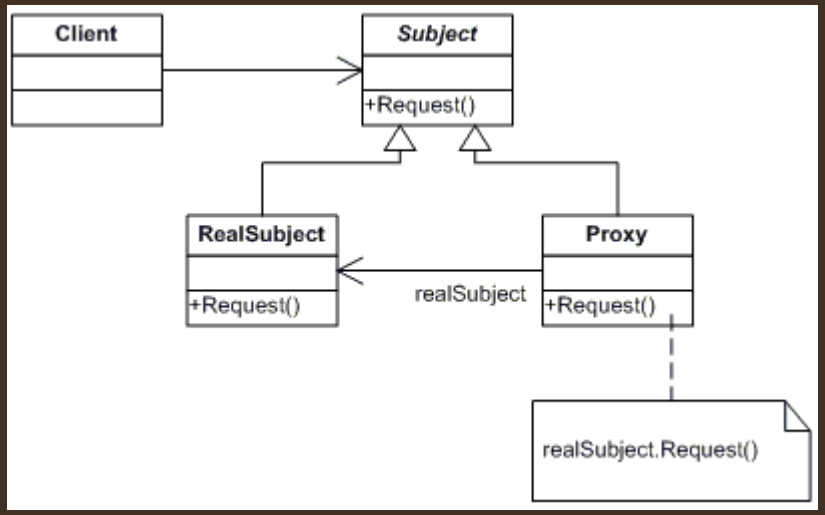
→ 分析

因為在GUI中，只有能夠制作的道具能按下Craft鍵，是只有能夠制作的道具會顯示出來，所以這一步驟就包含了判斷「是否能夠制作」的過程。所以在按下Craft按鍵後，直接與Inventory交互，並更新GUI便可。

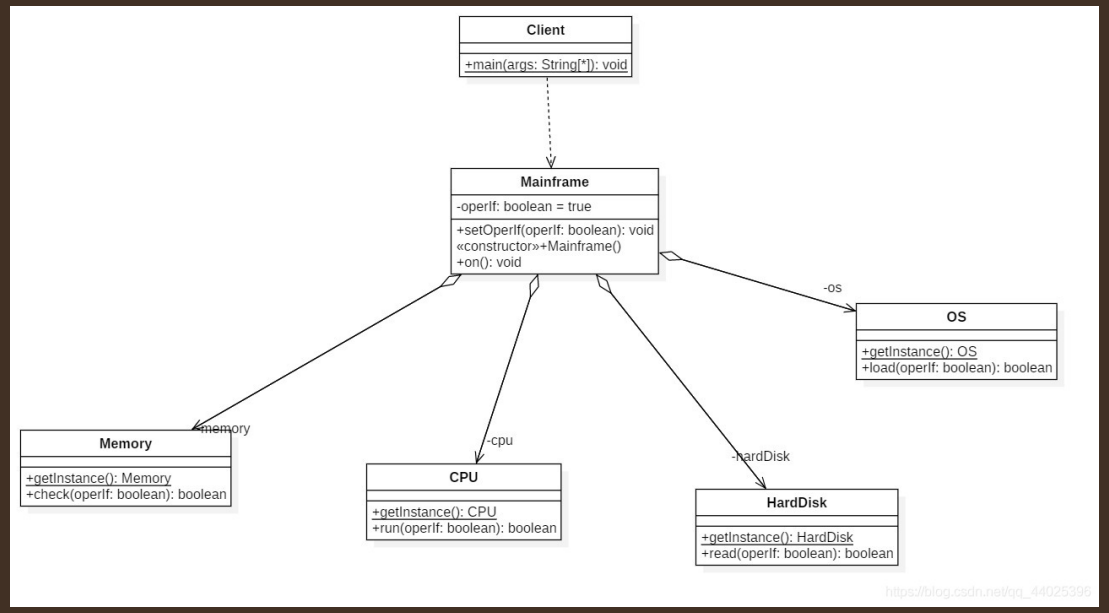
中介者/調停者模式（Mediator）



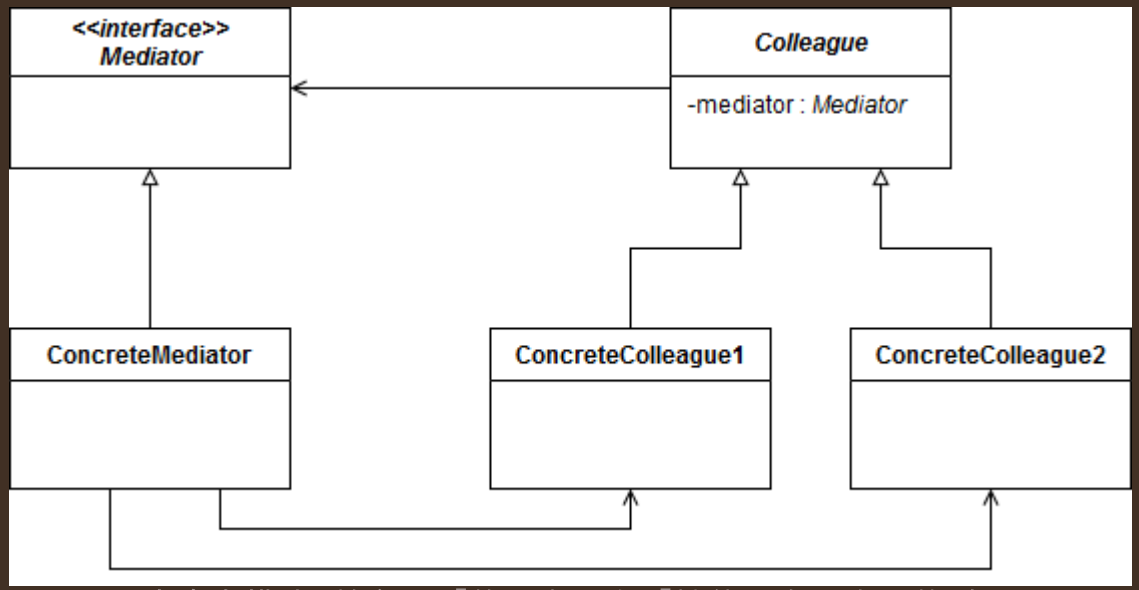
Mediator vs Proxy vs Facade



代理模式: 代理類與要被代理的類是同源的。



外觀模式: 各種子系統接口的集合



中介者模式: 抽象了「使用者」與「被使用者」之間的引用

Q&A

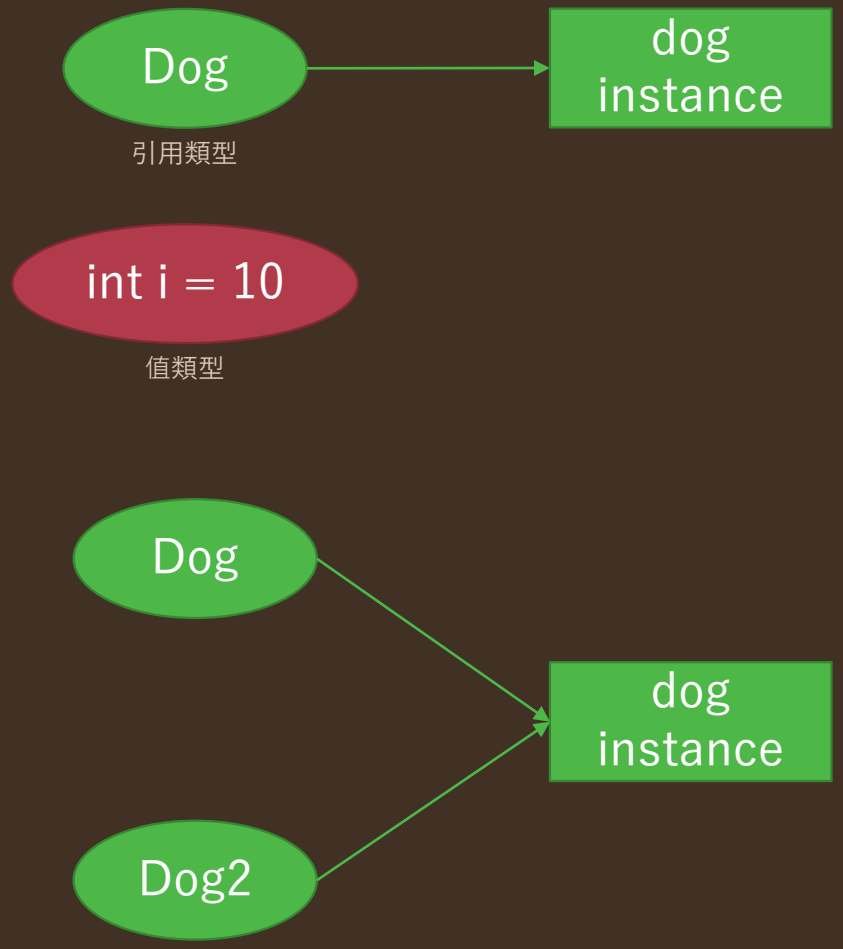
模式

- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式

引用類型 vs 值類型

→ 引用類型(指針): 變量存儲對其數據(對象)的引用。

→ 值類型: 變量直接包含其數據。



C# string引用類型

→ string是引用類型，但用起來卻是”值類型”。

如圖1，如果修改了string，正常來說，string2也會被修改。但實際上會變成圖2那樣，string和string2分別指向了不同的地址。

→ string對象是不可變的(readonly)，一旦創建了對象，就不能修改對象的值。string在修改的時候，實際是上建立了一個新的string對象，變量會指向這個新的對象。這也是string效率低下的原因。如果經常改變string的值，應該使用StringBuilder而不使用string。

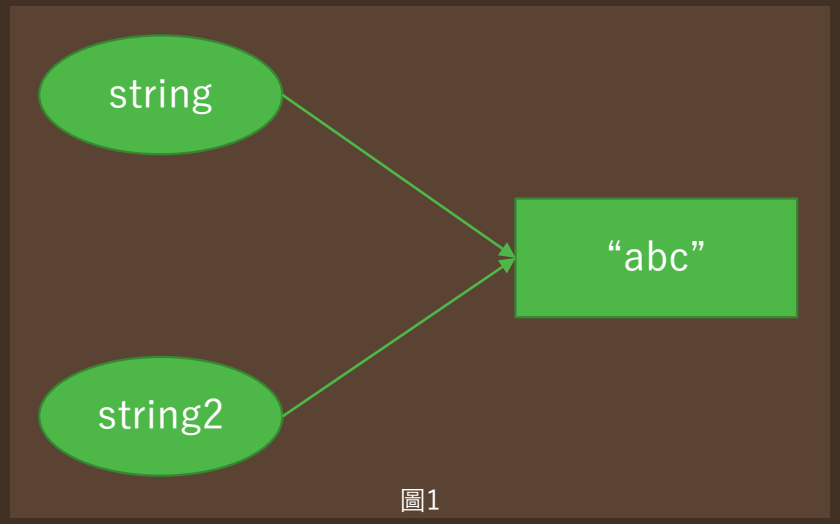


圖1

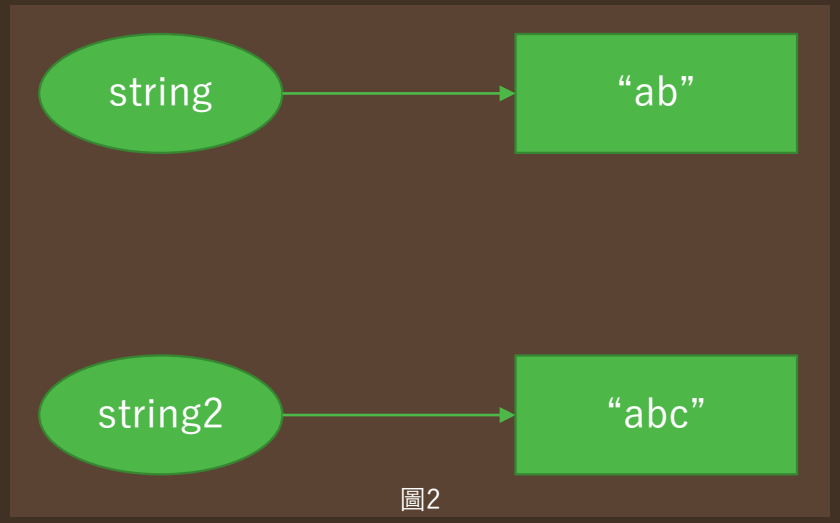


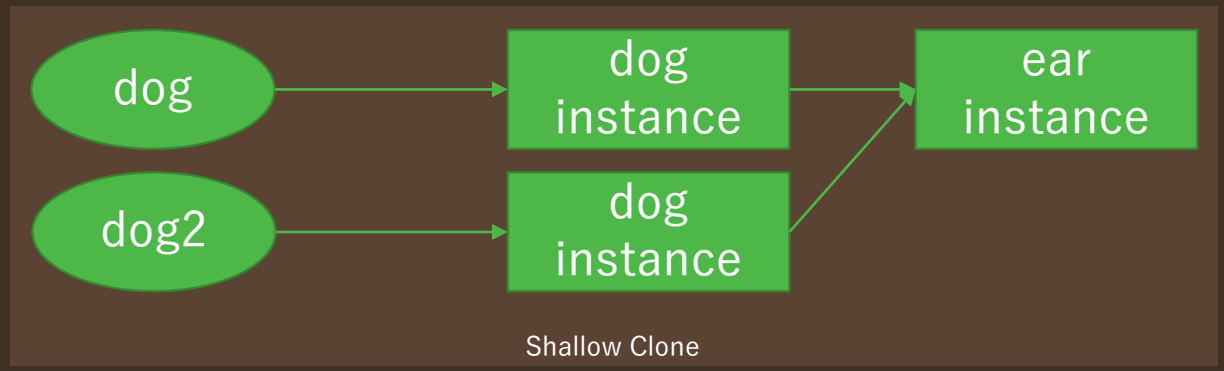
圖2

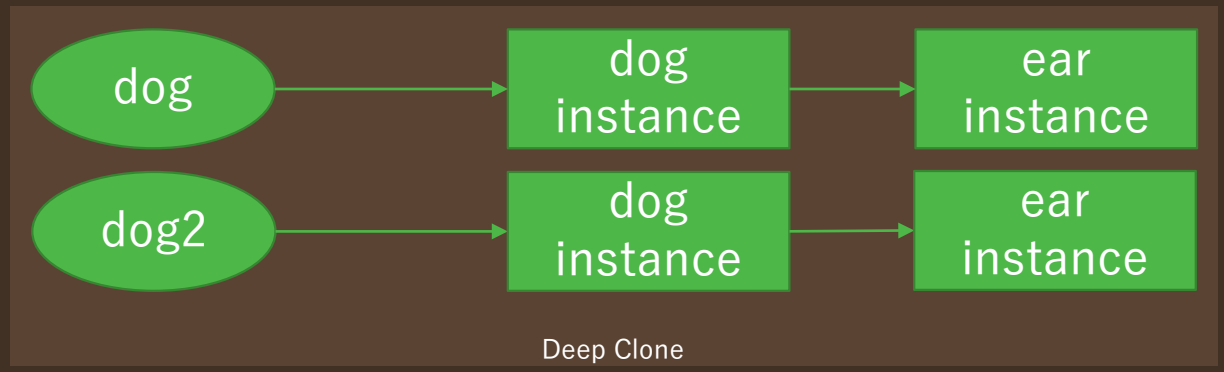
原型模式（Prototype）

- 用原型實例指定創建對象的種類，並且通過拷貝這些原型創建新的對象。

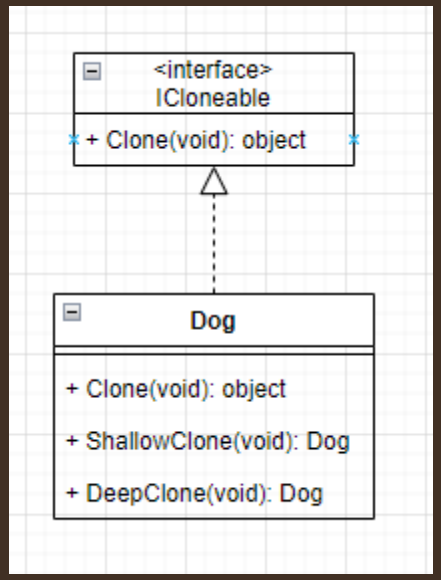
- 簡單來說，就是利用對象複製對象。
- Clone種類
  - Deep Clone
  - Shallow Clone
- 減少new和constructor的調用
- C#中提供了ICloneable接口，並提供了(protected)MemberwiseClone ()作為Shallow Clone方法。

Clone圖示


dog
dog2
dog instance
dog instance
ear instance
Shallow Clone


dog
dog2
dog instance
dog instance
ear instance
ear instance
Deep Clone

原型模式（Prototype）



原型模式（Prototype）

- Clone
  - 在C#的對象中，如果使用Clone()來創建對象，創建時不會調用new()和類的constructor。因此如果初始化比較繁瑣時，可以用Clone()來取代new ()，達到提升速度。
  - 但是實際上，正常情況下，Clone的速度並沒有new的快，除非constructor異常地耗時，否則Clone速度更慢。

原型模式（Prototype）

正常的constructor

Count	New (ms)	Clone (ms)
10000	6.4	7.8
100000	52.8	57.8
1000000	551.4	775.2
10000000	5873.8	6642.4

複雜的constructor

Count	new (ms)	Clone (ms)
10000	330.6	5.2
100000	3279.6	66.4
200000	6538.2	109.6
500000	16939	279.2

模式

- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式

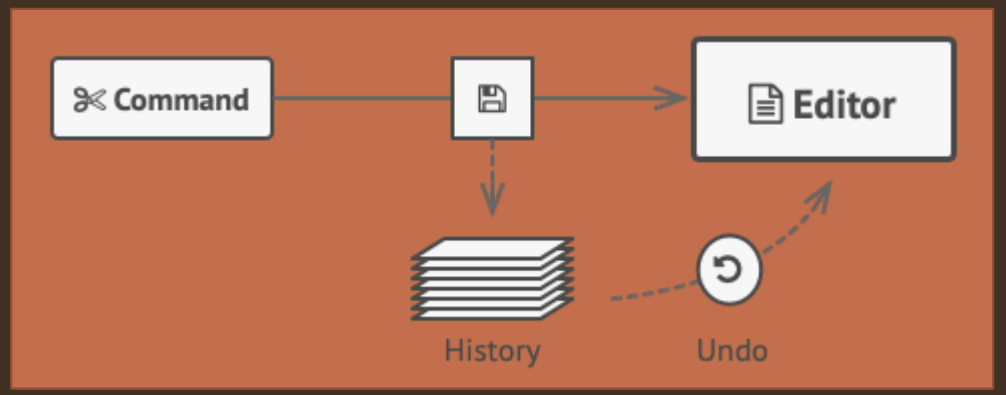
備忘錄模式（Memento）

→ 在不破壞封裝性的前提下，捕獲一個對象的內部狀態，並在該對象之外保存這狀態。這樣以後就可將該對象恢復到原先保存的狀態。

→ Photoshop中的撤銷功能，可以通過備忘錄模式實現。

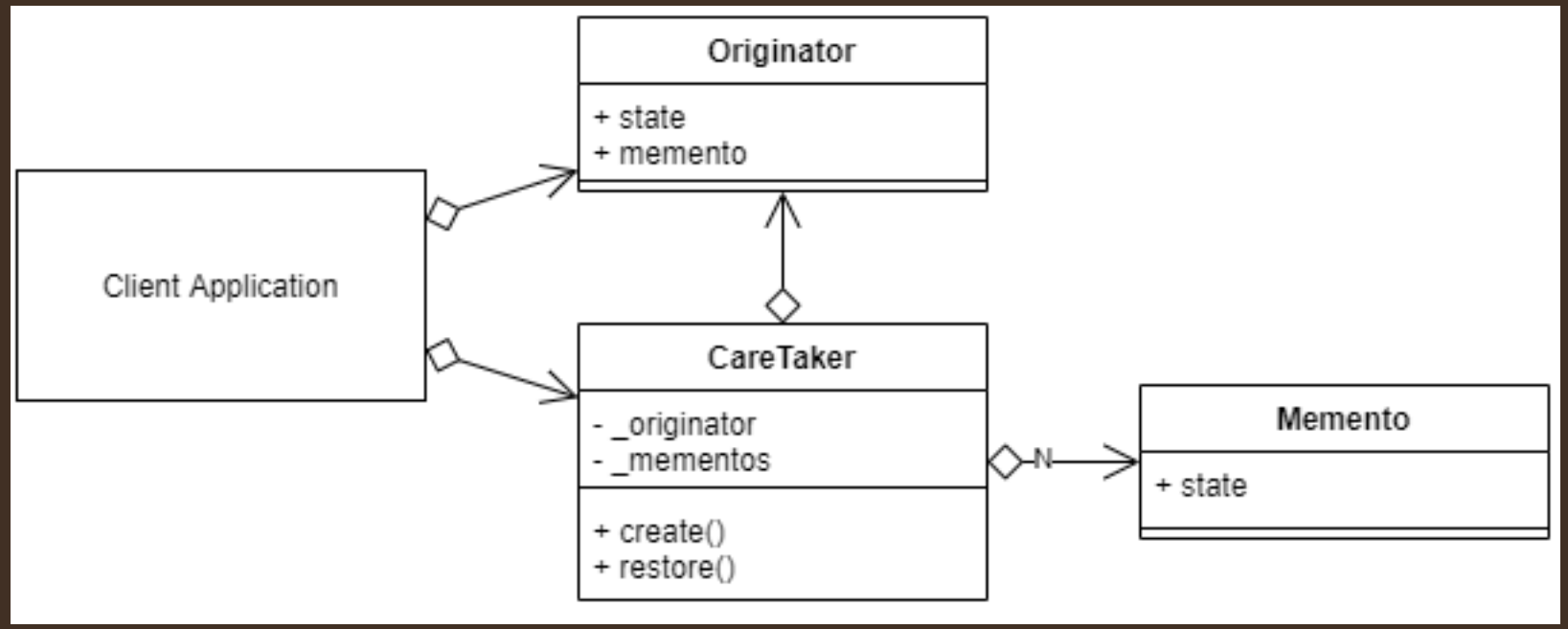
→ 將每一動作作為畫布的「備份」存儲起來。

→ 將功能從對象中分離出來，從而實現遵守單一責任原則(SRP)。

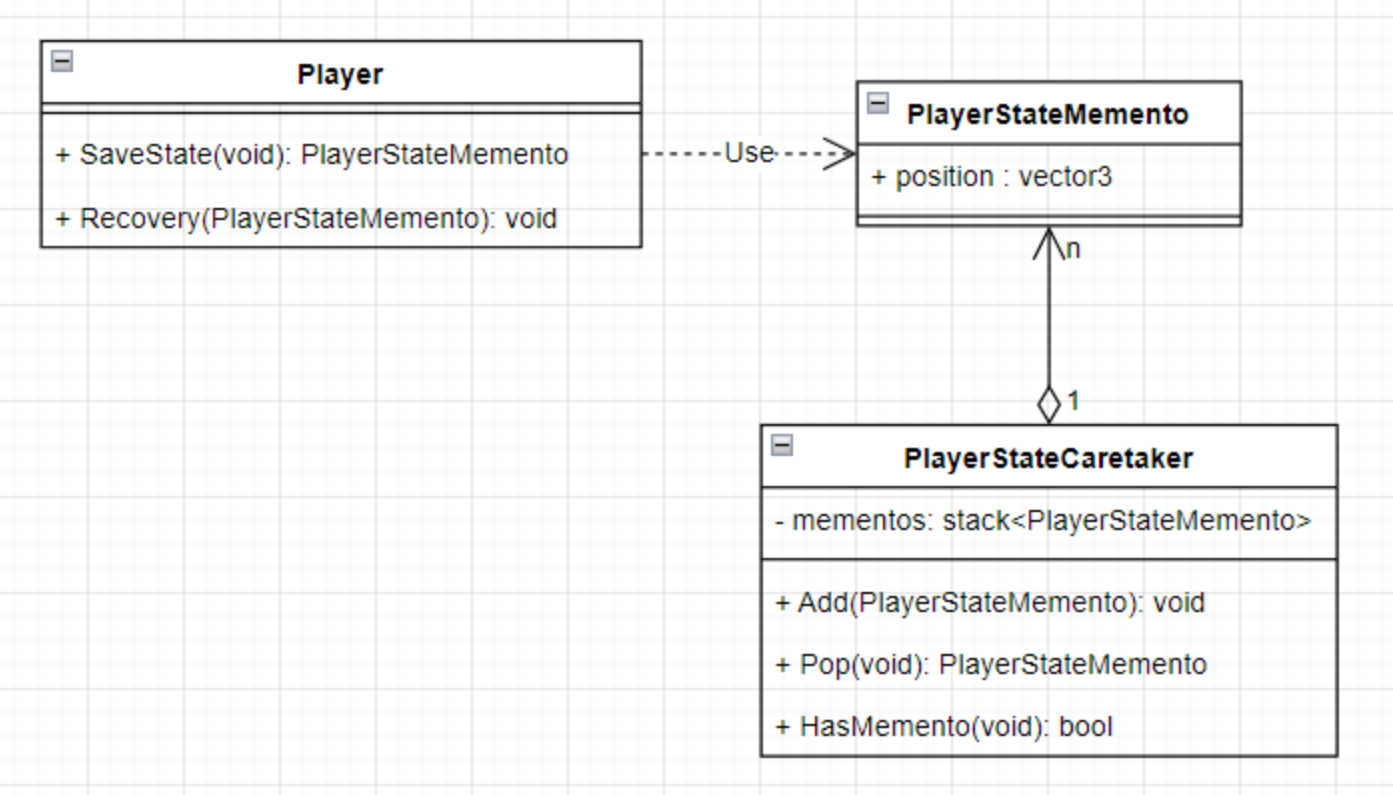


https://refactoring.guru/design-patterns/memento

備忘録模式（Memento）



備忘錄模式（Memento）



Q&A

總結

- 2種原則
  - 接口隔離原則
  - 里氏替換原則
- 其它
  - C# 值類型 vs 引用類型
    - string是引用類型?
- 5種模式
  - 迭代器模式
  - 組合模式
  - 中介者模式
  - 原型模式
  - 備忘錄模式