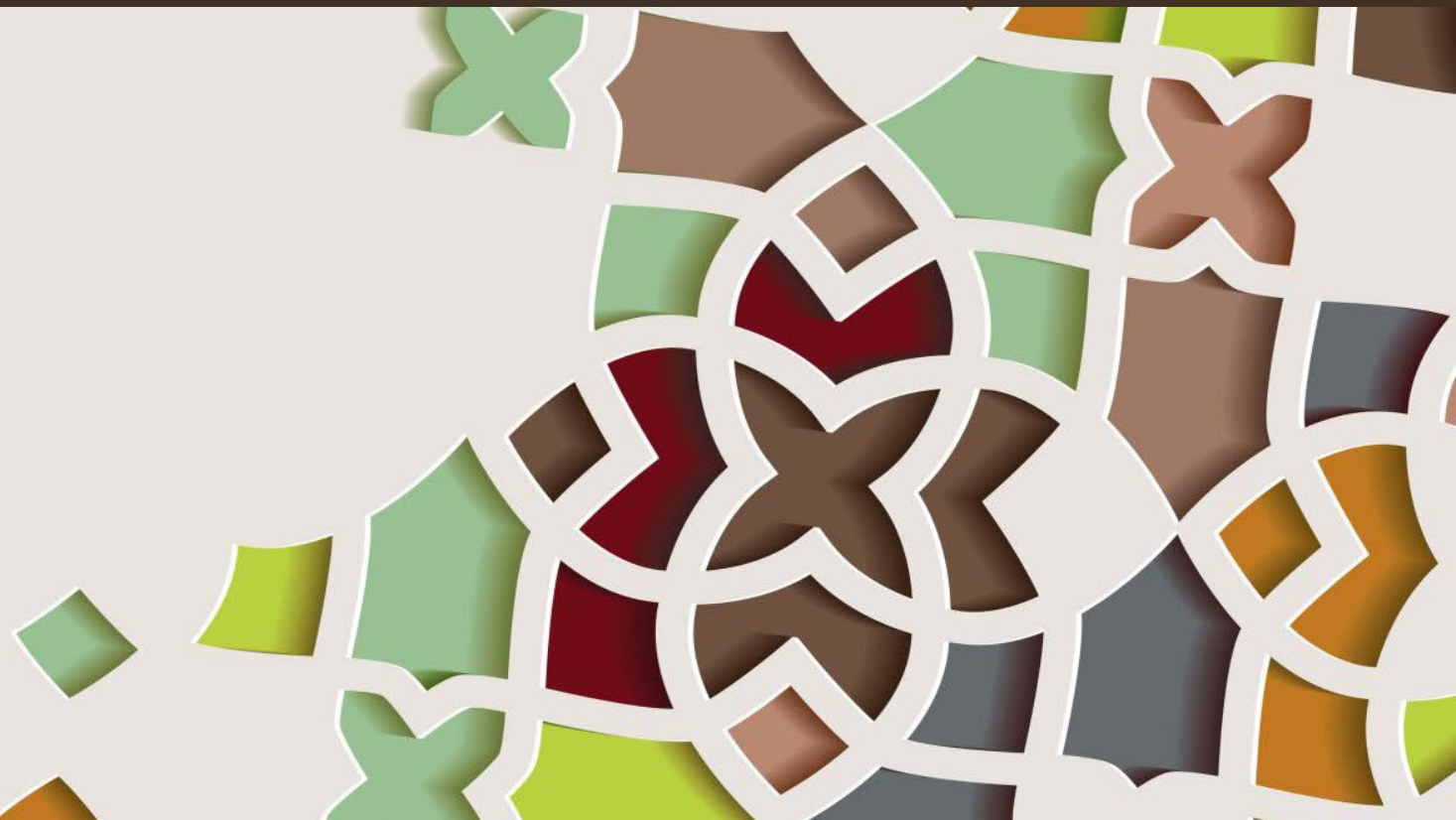
Design Pattern

Lecture 05



回顧

- 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

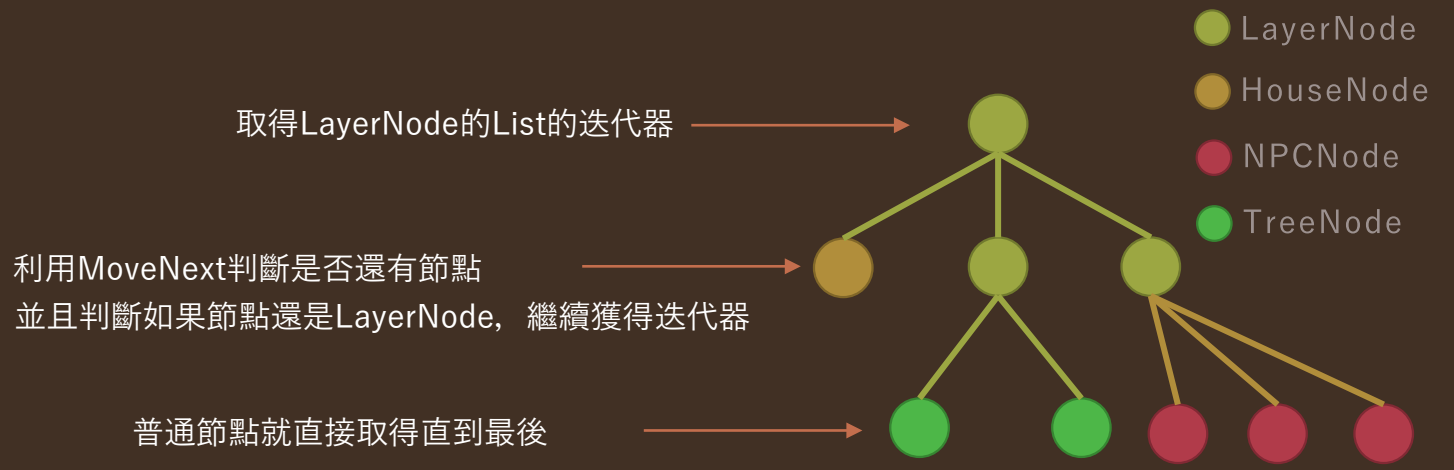
本回

- → 迭代器模式 & 組合模式
- → 5種模式
  - → 狀態模式
  - → 解釋器模式
  - → 蠅量/享元模式
  - → 責任/職責鏈模式
  - → 建造者/生成器模式

迭代器模式 & 組合模式

- 迭代器模式
  - 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。
- 組合模式
  - 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。

迭代器模式 & 組合模式


取得LayerNode的List的迭代器
利用MoveNext判斷是否還有節點
並且判斷如果節點還是LayerNode，繼續獲得迭代器
普通節點就直接取得直到最後
LayerNode
HouseNode
NPCNode
TreeNode

Q&A

模式

- → 5種模式
  - → 狀態模式
  - → 解釋器模式
  - → 蠅量/享元模式
  - → 責任/職責鏈模式
  - → 建造者/生成器模式

狀態模式（State）

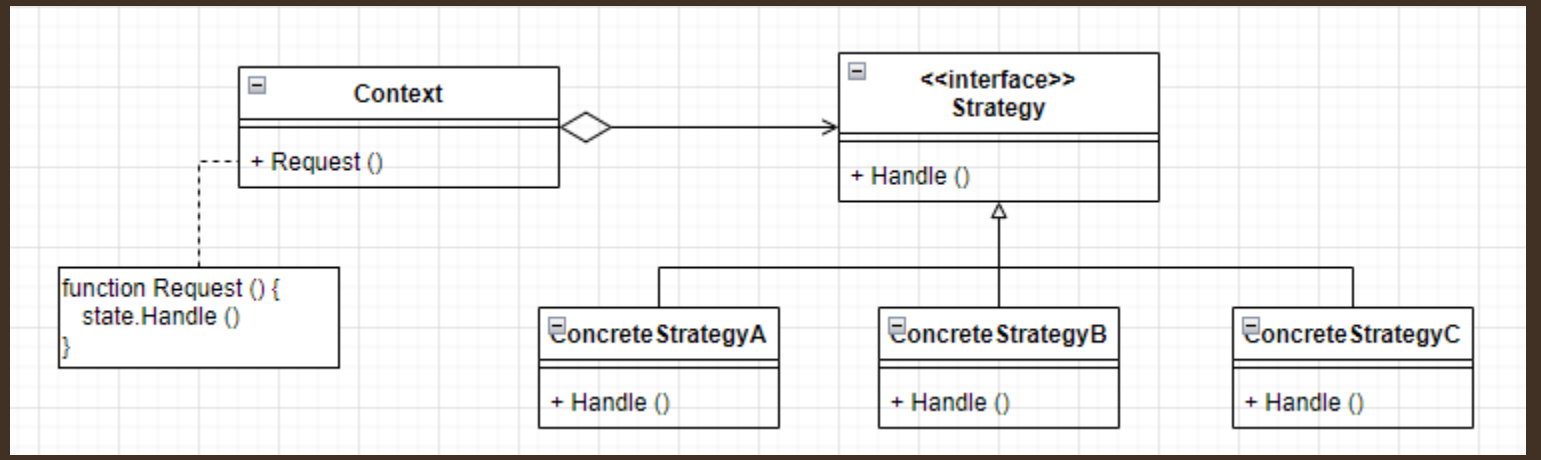
→ 允許對象在內部狀態改變時改變它的行為，對象看起來好像修改了它的類。

→ 將對象的狀態封裝成新的類族，使改變從使用者上分離(解耦)，並根據條件自動切換狀態。

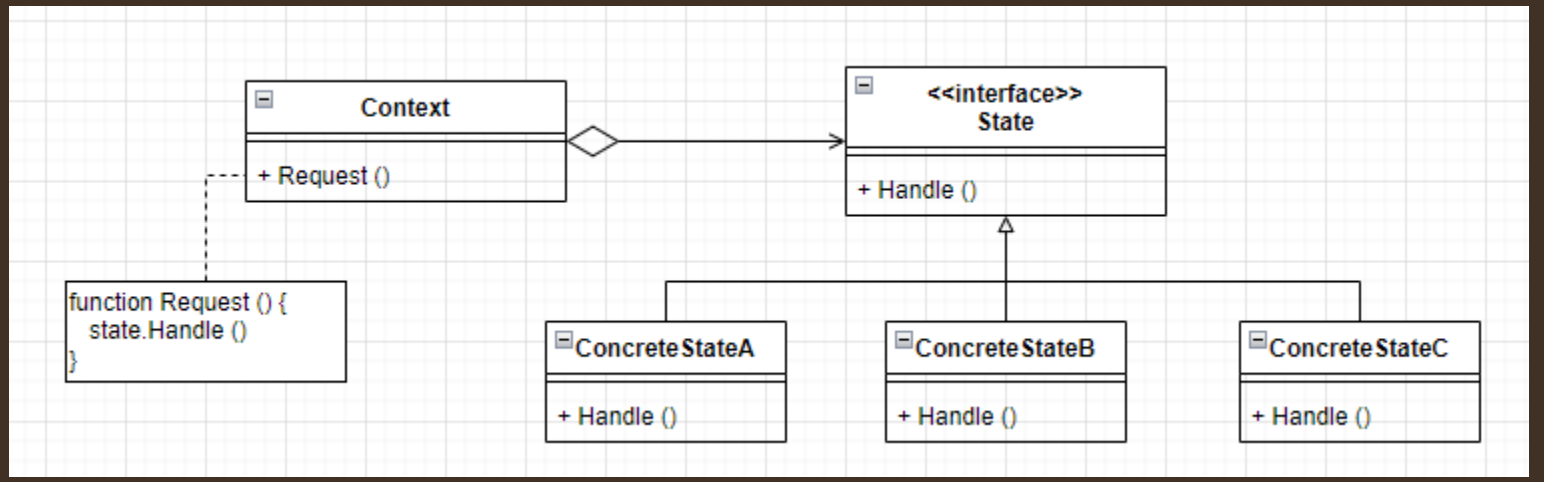
狀態模式（State）

- → 與策略模式相似，主要區別在於狀態模式是由內部自動地切換成不同的狀態，而外部無需關心它的切換條件。而策略模式是有意圖地改變其行為策略，而非自動。
- → 其它: Finite State Machine（有限狀態機）
  - → Unity - Animation
- → 優缺點
  - → 減少switch判斷當前狀態，並且減少判斷遺漏錯誤
  - → 分離責任，使代碼更清晰
  - → 若狀態過多，會產生過多的狀態類

狀態模式 vs. 策略模式

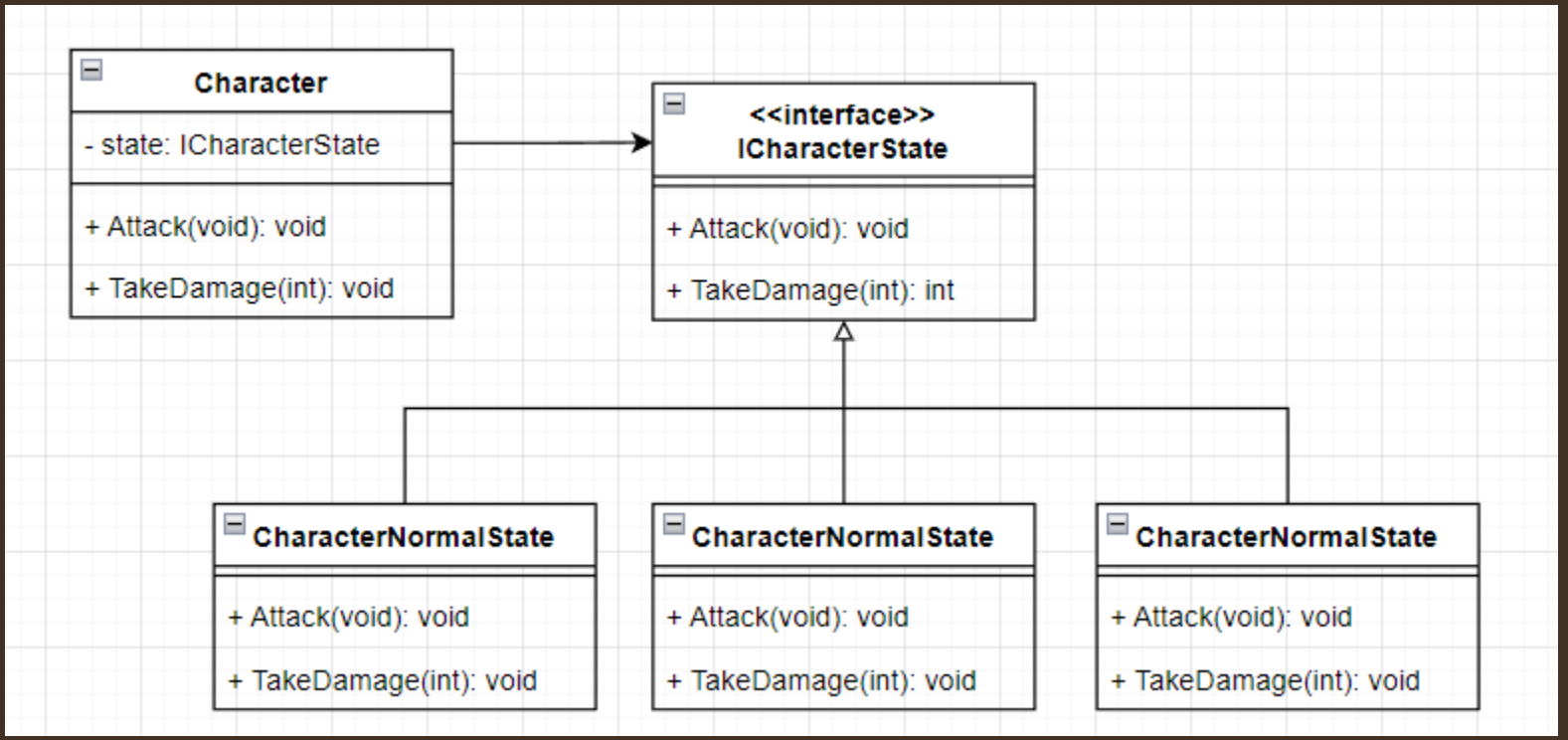


策略模式



狀態模式

狀態模式（State）


Character
- state: ICharacterState
+ Attack(void): void
+ TakeDamage(int): void
<<interface>>
ICharacterState
+ Attack(void): void
+ TakeDamage(int): int
CharacterNormalState
+ Attack(void): void
+ TakeDamage(int): void
CharacterNormalState
+ Attack(void): void
+ TakeDamage(int): void
CharacterNormalState
+ Attack(void): void
+ TakeDamage(int): void

Q&A

模式

- 5種模式
  - 狀態模式
  - 解釋器模式
  - 蠅量/享元模式
  - 責任/職責鏈模式
  - 建造者/生成器模式

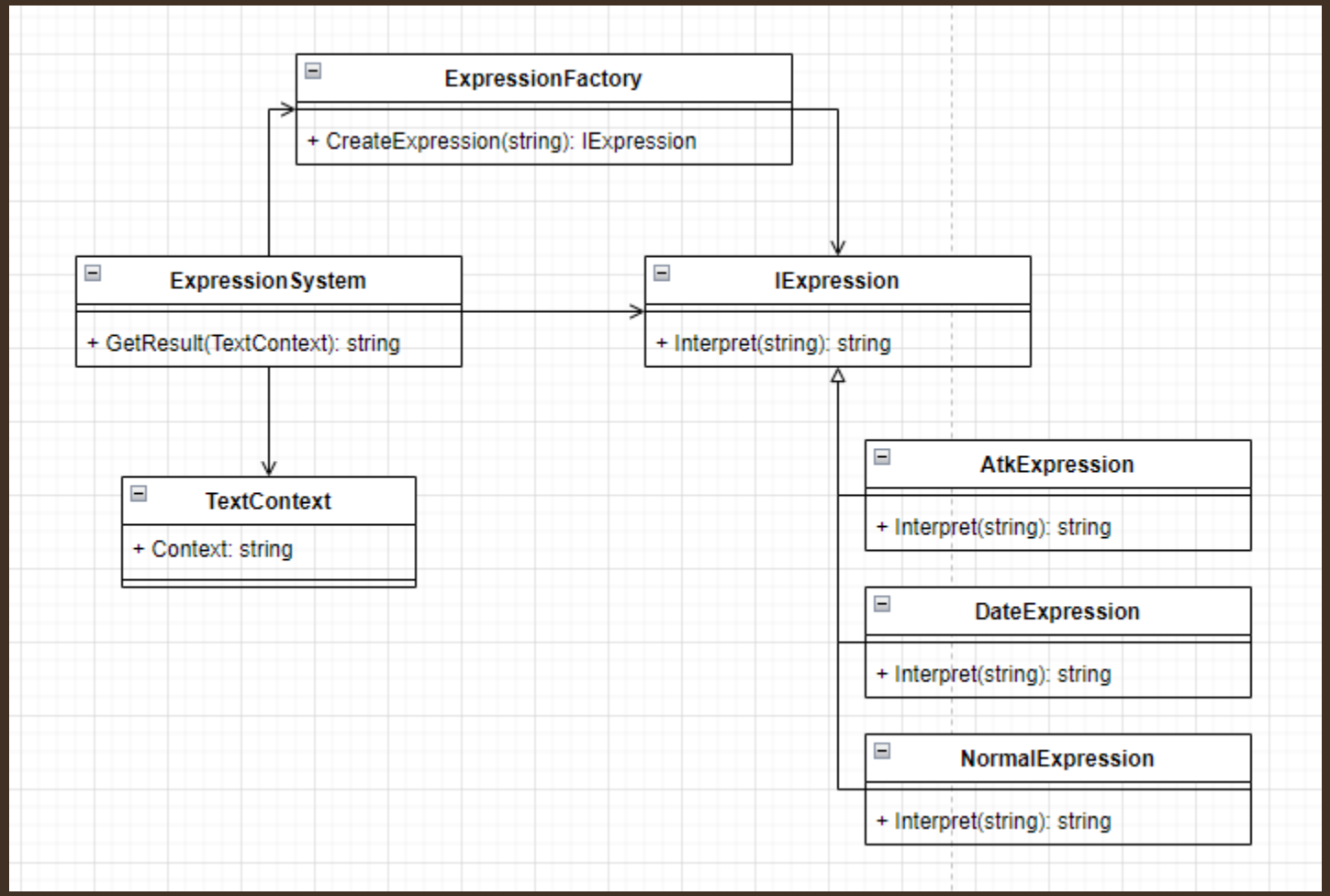
解釋器模式（Interpreter）

→ 給定一個語言定義它的文法的一種表示，並定義一個解釋器，這個解釋器使用該表示來解釋語言中句子。

→ 類似於編譯器。

→ 更簡單地來理解就是游戲的文本經常會出現的<SkillName>、<Atk>。這些名字都會被程序解釋成對應的值。

解釋器模式（Interpreter）



Q&A

模式

- → 5種模式
  - → 狀態模式
  - → 解釋器模式
  - → 蠅量/享元模式
  - → 責任/職責鏈模式
  - → 建造者/生成器模式

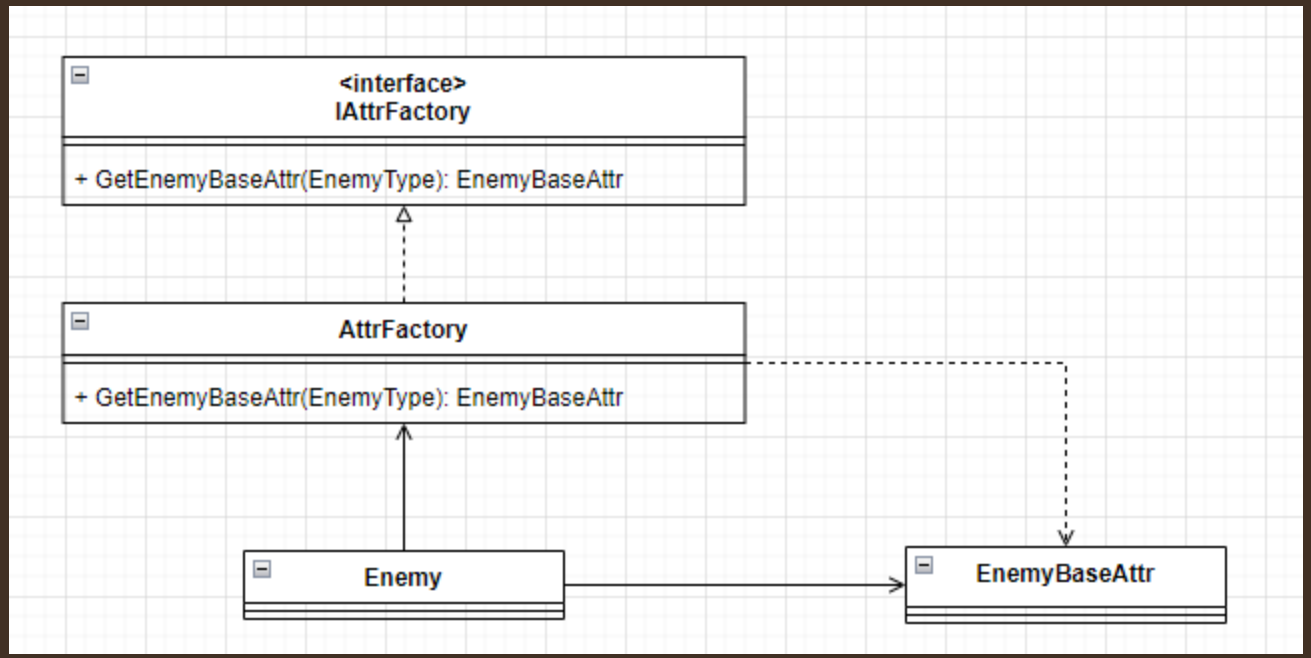
蠅量/享元模式（Flyweight）

- 運用共享技術有效地支持大量細粒度的對象。

- 將一些相同且不變的屬性提取出來變成公共使用的屬性，減少內存消耗。
  - 角色的血量、攻擊力、移動速度...
- 能有效地降低內存消耗，但是在取得屬性時也會增加了CPU的消耗(消耗不大)

- 與對象池(pool)的區別
  - 對象池中的對象是提供給單個對象使用的。
  - 享元中的對象是共享給多個對象使用的，而且不變的。

蠅量/享元模式（Flyweight）



模式

- → 5種模式
  - → 狀態模式
  - → 解釋器模式
  - → 蠅量/享元模式
  - → 責任/職責鏈模式
  - → 建造者/生成器模式

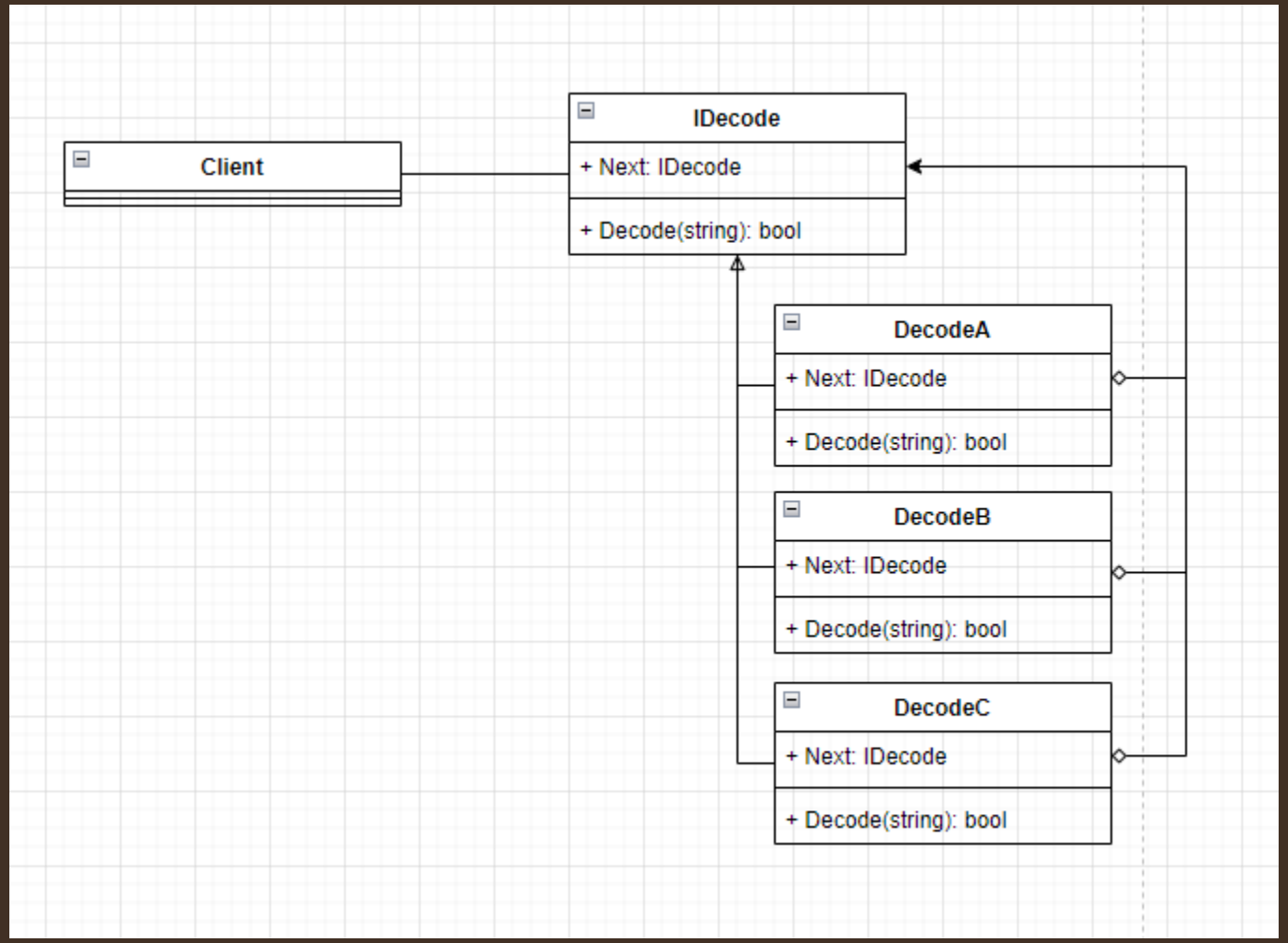
責任/職責鏈模式（Chain of Responsibility）

- → 使多個對象都有機會處理請求，從而避免請求的發送者和接受者之間的耦合關係，將這個對象連成一條鏈，並沿着這條鏈傳遞請求，直到有一個對象處理它為止。

- → 員工向上司提出加薪時(加薪是請求)，會一層層地往上提交，直到請求被解決。
  - → 組長->經理->總經理->CEO
  - → 上面就會形成一條單向的鏈，負責處理加薪請求

- → 需要注意的是，如果對象有能力處理請求，那麼就不需要再將請交向上提交。

責任/職責鏈模式（Chain of Responsibility）



Q&A

模式

- → 5種模式
  - → 狀態模式
  - → 解釋器模式
  - → 蠅量/享元模式
  - → 責任/職責鏈模式
  - → 建造者/生成器模式

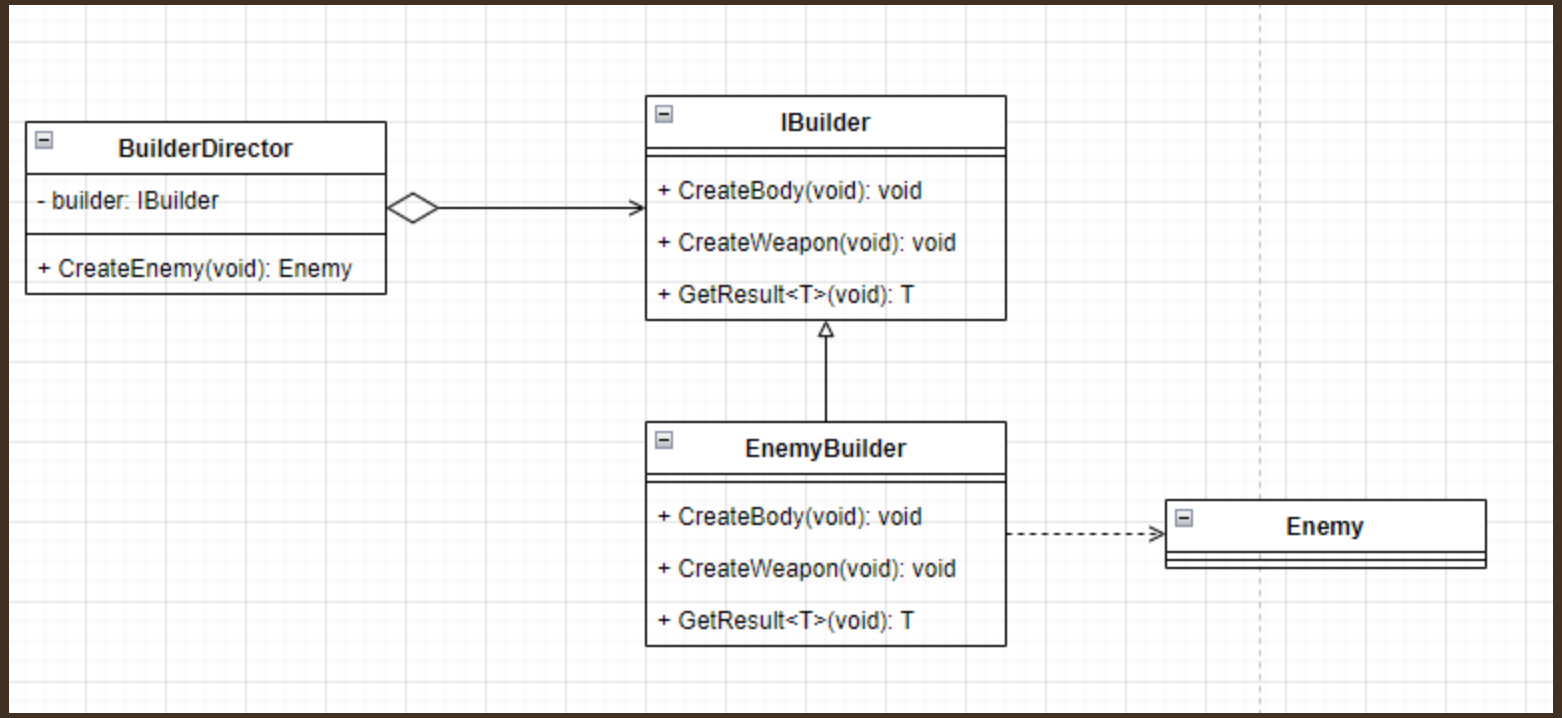
建造者/生成器（Builder）

→ 將一個複雜對象的構建與它的表示分離，使得同樣過程可以創建不同的表示。

→ 與工廠模式類似。但是建造者模式會關心對象的每一個部件(part)的生產。每一個部件的生產也可以是利用工廠模式。

→ 當不需要關心部件(part)的生產時，可以使用工廠。否則使用建造者。

建造者/生成器（Builder）



Q&A

總結

- → 迭代器模式 & 組合模式
- → 5種模式
  - → 狀態模式
  - → 解釋器模式
  - → 蠅量/享元模式
  - → 責任/職責鏈模式
  - → 建造者/生成器模式