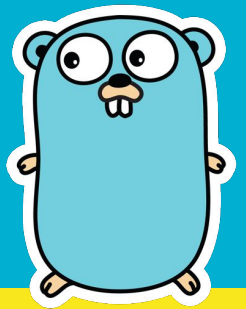
Go 程式設計課 02

—

Go 與設計模式

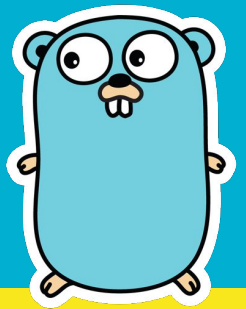
大綱

- 完成 Go 之旅的
  - interface
  - 泛型
- 介紹物件導向
- 介紹函數導向
- 介紹設計模式
- UML 圖繪製
  - 物件圖



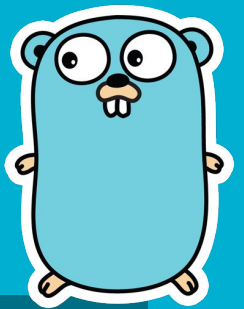
大綱

- 設計模式
  - 工廠模式
  - 策略模式
  - 觀察者模式
- 設計模式介紹與實戰
  - 工廠模式
  - 策略模式
  - 觀察者模式



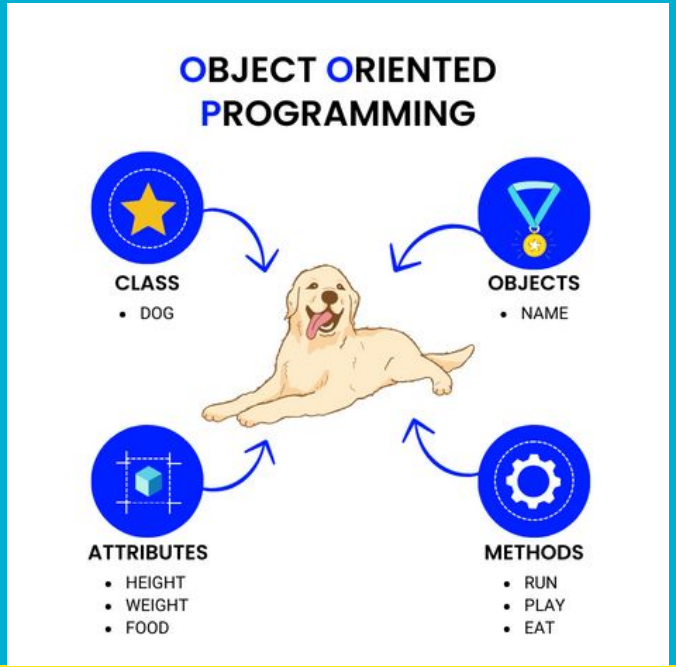
完成Go之旅的-Interface(介面)

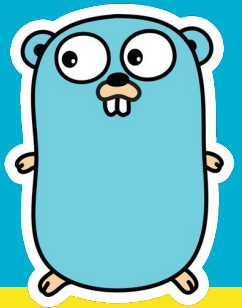
完成Go之旅的-Generics(泛型)



物件導向(Object-Oriented Programming, OOP)

- 物件導向設計(Object-Oriented Programming)，簡稱OOP或OO
- 特色：
  - 物件 (Object)
  - 封裝 (Encapsulation)
  - 繼承(Inheritance)
  - 多型(Polymorphism)





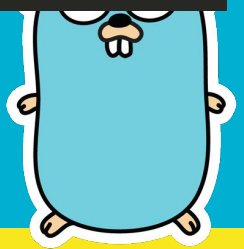
OO的優缺點

- 優點:
  - 模組化 (Modularity):程式碼可以分成多個獨立的物件，易於理解和維護
  - 重用性 (Reusability):可以通過繼承和多型機制重用現有的程式碼
  - 可擴展性 (Scalability):容易擴展和修改，容易添加新的功能
  - 可讀性 (Readability):使用OOP可以提高程式碼的可讀性和理解度
- 缺點:
  - ~~執行效率 (Execution Efficiency):OOP可能會比較佔用記憶體和計算資源，導致執行效率較低~~
  - ~~設計複雜性 (Design Complexity):如果設計不良，可能導致過度複雜的層次結構~~

OO與Go

- OO 常使用 reference 來傳遞物件，以確保現在操作的是同一個 物件。
- 至於 Go 算不算物件導向的語言呢？
  - 簡單講，因為Go缺乏繼承，只能稱之為物件導向風格(OO style)

```
1   type Human struct {
2       Height  int
3       Weight  int
4       Age     int
5   }
6
7   func (h *Human) HappyBirthday() {
8       h.Age += 1
9   }
10
11  func main() {
12      me := Human{ Height: 175, Weight: 70, Age: 22 }
13      me.HappyBirthday()
14      fmt.Println(me.Age)
15  }
16
17  // OUTPUT
18  > 23
```



OO與Go - 指標 Pointer

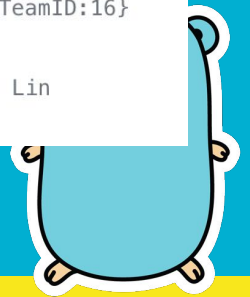
- 在 Go中，可以用**指標(pointer)**來傳遞 struct address，達到傳 reference 的效果。
- 右圖範例用結構在宣告時就用 &，表示回傳一個位址，用這個位址傳入函式操作。

```go
package main
import "fmt"

type Person struct {
        Name string
}

type Member struct {
        // Person   *Person // struct in struct
        *Person  // 不需要欄位名稱
        TeamName string
        TeamID   int
}

func main() {
        var haren = &Member{&Person{"Haren Lin"}, "ERS", 16}
        fmt.Printf("Member = %+v\n", *haren)
        // Member = {Person:0xc000010240 TeamName:ERS TeamID:16}
        fmt.Printf("%s\n", haren.Name) // Haren Lin
        fmt.Printf("%s\n", haren.Person.Name) // Haren Lin
}
```



OO與Go - 內嵌 Embedding

OOP 常會用 Inheritance 來共享父類別程式碼。

然而，Go 沒有繼承的特性，但能用「組合 Compisition」的方式來共享程式碼。不僅如此，Go 還提供一種優於組合的語法特性，稱作**內嵌 (Embedding)**

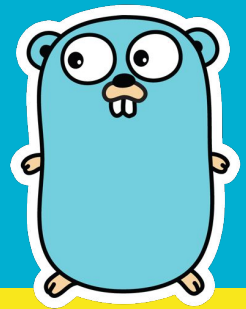
```go
package main

import "fmt"

type Person struct {
    Name string
}

type Member struct {
    Person   *Person // struct in struct
    TeamName string
    TeamID   int
}

func main() {
    var haren = &Member{&Person{"Haren Lin"}, "ERS", 16}
    fmt.Printf("Member = %+v\n", *haren)          // Member = {Person:0xc000096220 TeamName:ERS TeamID:16}
    fmt.Printf("Person = %+v\n", *(haren.Person)) // Person = {Name:Haren Lin}
}
```

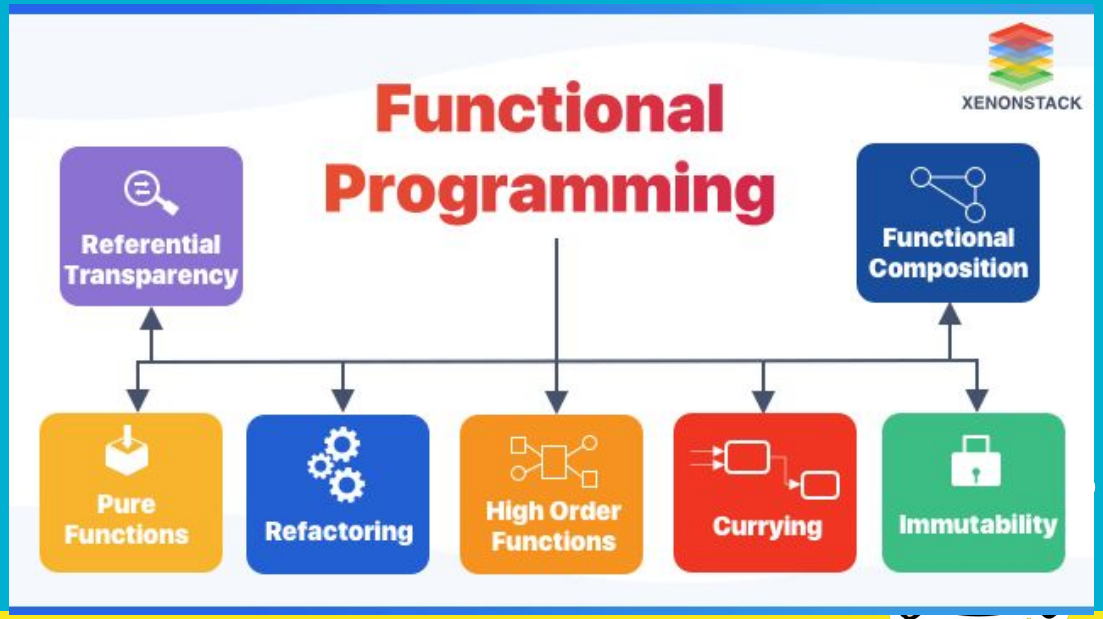


函數導向(Functional-Oriented Programming)

- 函式語言程式設計（functional programming）或稱函式程式設計、泛函編程，是一種編程範式，它將電腦運算視為函式運算，並且避免使用程式狀態以及易變物件。其中，**λ演算（lambda calculus）**為該語言最重要的基礎。而且，λ演算的函式可以接受函式當作輸入（引數）和輸出（傳出值）。
- 比起指令式編程，函數式編程更加強調程式執行的結果而非執行的過程，倡導利用若干簡單的執行單元讓計算結果不斷漸進，逐層推導複雜的運算，而不是設計一個複雜的執行過程。
- 在函式語言程式設計中，函式是第一類物件，意思是說一個函式，既可以作為其它函式的參數（輸入值），也可以從函式中返回（輸入值），被修改或者被分配給一個變數。

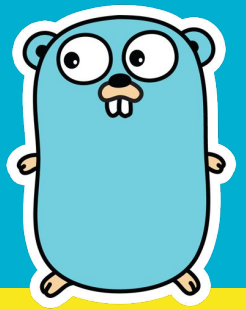
FP的特色

- 無狀態(stateless)
- 宣告式(declarative)思考，將程式步驟拆成更小、更明確運算單元，讓程式更好閱讀(Pure functions)
- 不可變性 (Immutability)



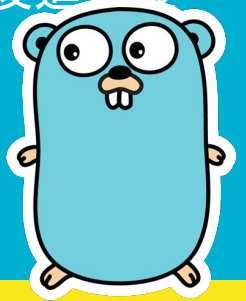
FP的優缺點

- 優點：
  - 簡潔性 (Conciseness)：FP程式碼通常比較簡潔，易於理解。
  - 並行性 (Concurrency)：由於函式的無狀態和不可變性，易於實現並行計算。
  - 穩定性 (Stability)：FP的行為通常更加穩定和可靠。
  - 測試性 (Testability)：由於函式不依賴於外部狀態，易於測試。
- 缺點：
  - 效能問題 (Performance Concerns)：某些情況下，函式導向的程式碼可能會因為大量的函式調用而導致效能下降（可能因為追求pure而有無意義複製）
  - ~~不適合所有情況 (Not Suitable for All Situations)~~

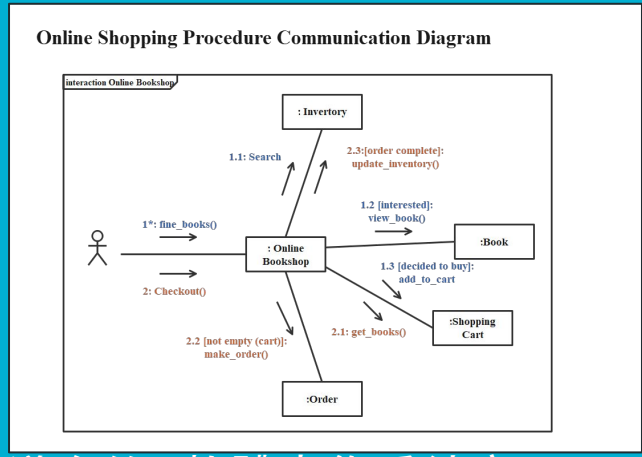


FP的Currying

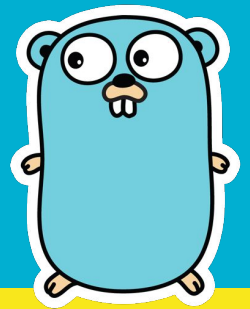
- Curry Function
  - function 本身可以當作另一個 function 的參數傳入，同時也能夠回傳一個新的 function
  - 透過currying，我們可以把一些參數設定先拆出來，但不處理資料運算；而傳入不同的設定參數後取得不同目標的 function，把 function 功能拆分得更小，也讓程式的重用性更高。
  - 可以輕易地把「設定」和「資料」進行隔離
- 惰性求值(Lazy Evaluation)，或稱為延遲評估，其中重點在於「盡可能延後進行複雜運算的行為」



介紹UML圖



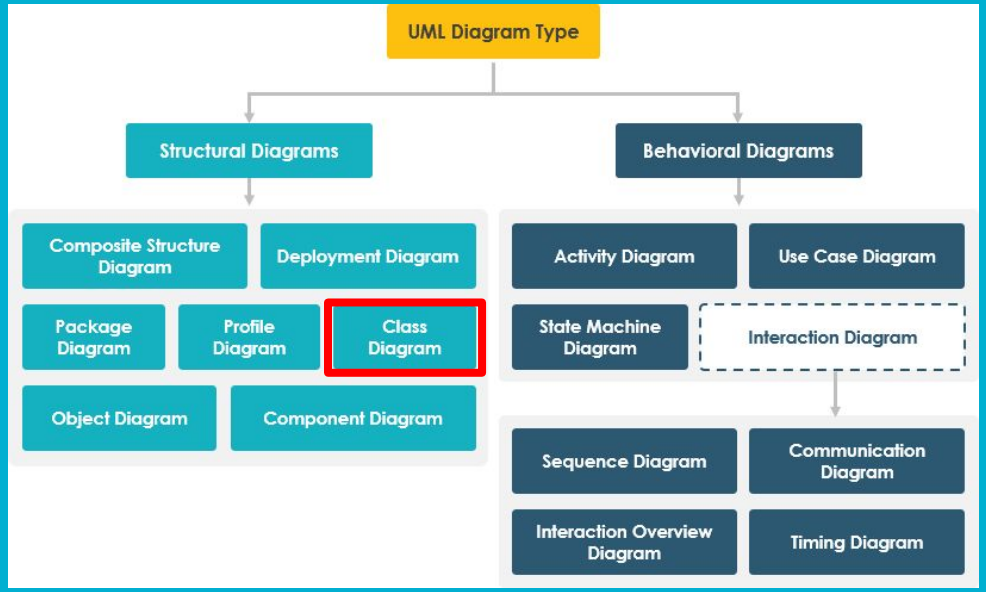
- UML是一種標準化的通用建模語言
- 由Object Management Group(簡稱OMG)制定
- 用於說明、視覺化、構建和編寫一個正在開發的、物件導向的、軟體密集系統產品的開放方法（by 維基百科）
- 為一種圖形表示法(graphic notation)，其模型必須依據相關的圖示標記與規範來加以繪製
- UML是視覺化的塑模語言(visual modeling language)，故UML模型又被稱為視覺化模型(visual model)

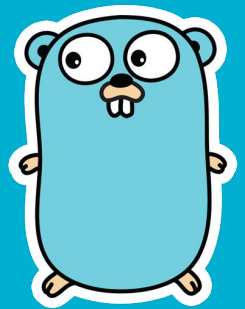


UML圖的分類

在UML 2.2中，共定義了14種圖示：

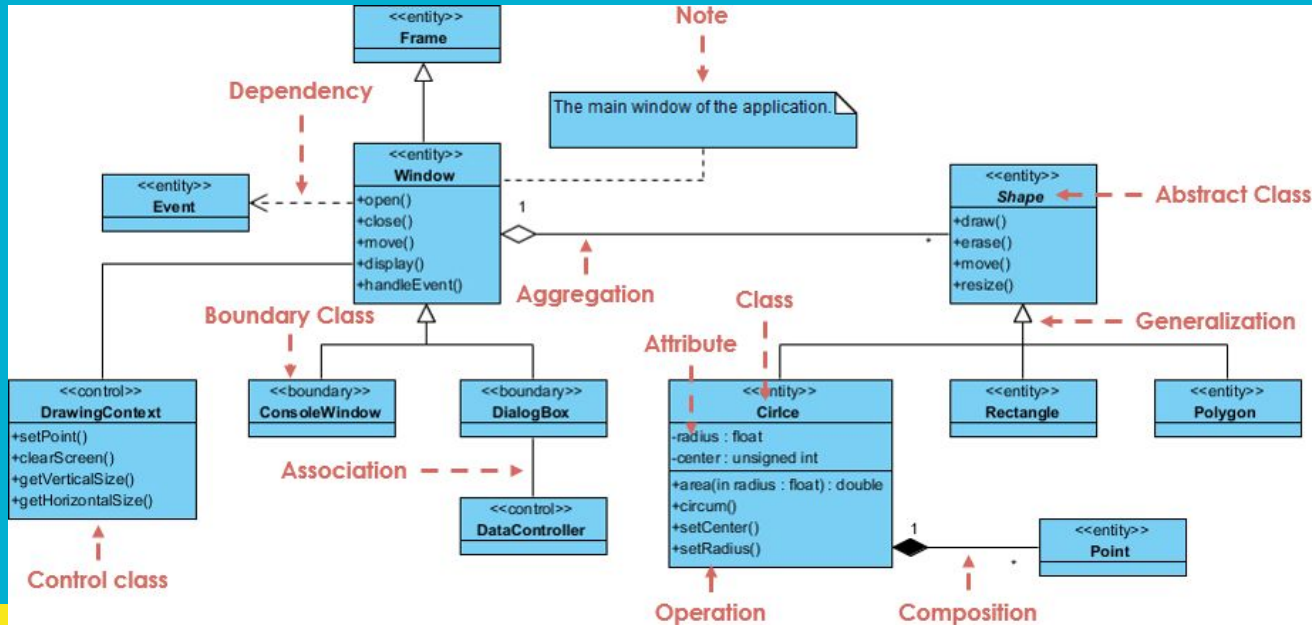
- 行為式圖形(Behavioral Diagrams)
- 結構性圖形(Structure Diagrams)
- 互動性圖形(Interaction Diagrams)





介紹類別圖(Class Diagram)

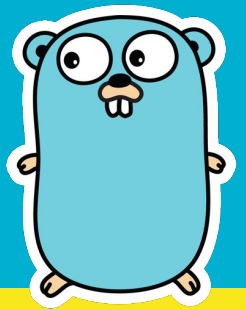
類別圖(Class Diagram)是一種靜態圖示，它透過顯示系統的類別及其屬性和操作以及物件之間的關係來描述系統的結構。



介紹Class Diagram - 使用Mermaid

1. 定義類別 (Defining Classes)：

- 語法：**class 類別名稱**
- 意義：定義一個新類別。類別(class)代表了一群具有相同屬性和方法的物件
- 舉例：class Student 定義了一個名為 "Student" 的類別

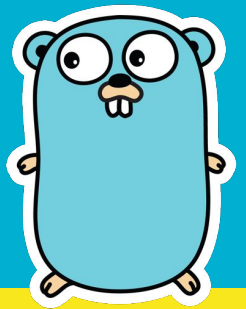


介紹Class Diagram - 使用Mermaid

2. 定義類別成員 (Defining Class Members):

- 語法:**類別名稱 : 成員名稱**
- 意義:用來定義類別的屬性或方法
  - 屬性:代表類別的資料
  - 方法:代表類別可以執行的行為
- 舉例:Student : -int studentID

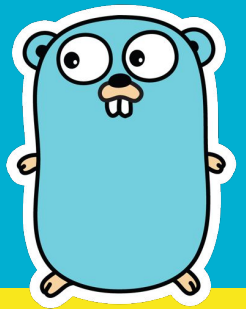
  定義了 Student 類別中一個名為 studentID 的整數型別屬性



介紹Class Diagram - 使用Mermaid

3. 類別之間的關係 (Relationships):

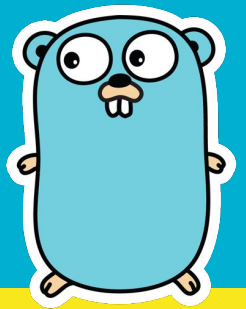
- 繼承 (Inheritance): **類別A <|-- 類別B**
  - 意義: 表示類別B繼承自類別A,類別B會擁有類別A的所有屬性和方法。
- 組合 (Composition): **類別A *-- 類別B**
  - 意義: 表示類別A包含類別B,類別B的生命週期依賴於類別A。
- 聚合 (Aggregation): **類別A o-- 類別B**
  - 意義: 表示類別A使用類別B,但類別B的生命週期不依賴於類別A。
- 關聯 (Association): **類別A --> 類別B**
  - 意義: 表示類別A和類別B有關聯,一個類別的物件可以與另一個類別的物件互動。



介紹Class Diagram - 使用Mermaid

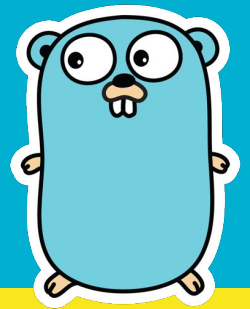
4. 類別關係上的標籤 (Labels on Relations)：

- 語法：**類別A "標籤" 關係符號 類別B**
- 意義：可以在類別之間的關係上添加描述性的標籤，說明關係的性質
- 舉例：Customer "1" --> "*" Order 表示一個客戶可以下多個訂單(1對多關係)



繪製UML的步驟

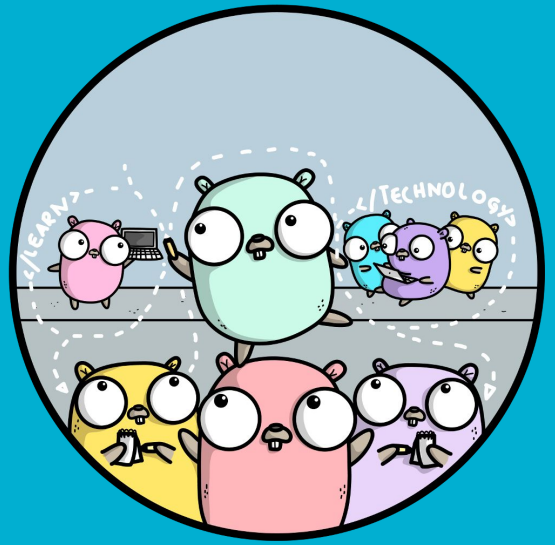
- 確定目的和範圍
- 選擇適當的 UML 圖類型
- 識別元素
- 定義關係
- 開始繪製UML圖
- 檢視和改進，確保其能準確代表正在建立的系統/流程
- 製作相關說明文件
- 保持更新
- 儲存、共享及匯出



小試身手

1) code : https://github.com/leon123858/go-tutorial/tree/main/design-pattern
2) online - editor
   https://mermaid.js.org/syntax/classDiagram.html
1) online - edteor - GUI
   https://app.diagrams.net/

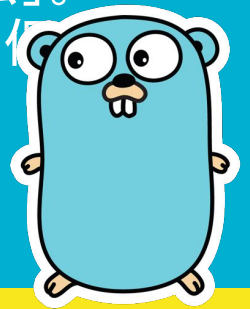
按照 Lab 中的 SPEC 繪製物件圖吧！



設計模式

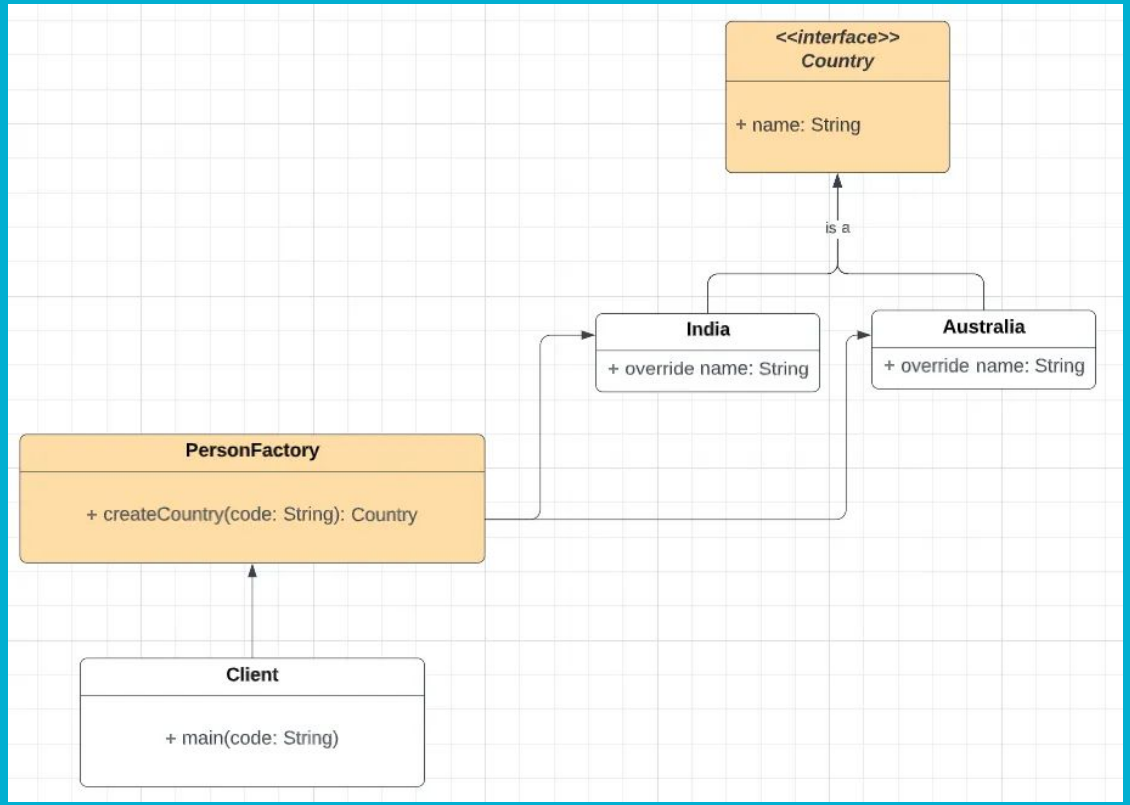
工廠模式(Factory Method Pattern)

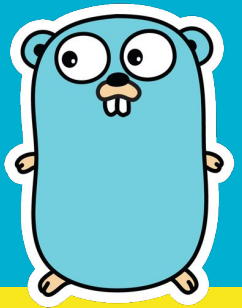
- 概念：工廠(Factory)模式定義了一個建立物件的介面，但由次類別決定要實體化的類別為何者。工廠方法讓類別把實體化的動作，交由次類別進行
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- 工廠方法模式可以封裝「具象型態的實體化」。用類別圖來看，抽象的 Creator 提供一個方法能建立物件，這樣的方法就叫做「工廠方法」。
- 在 Creator 中的其他方法都能使用工廠方法所製造出來的產品，但次類別真正實作這個方法，產生出物件。



工廠模式(Factory Method Pattern)

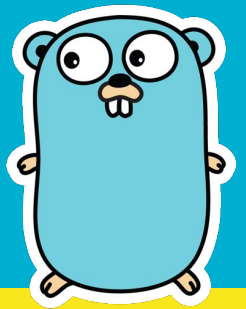
❖ 示意圖:





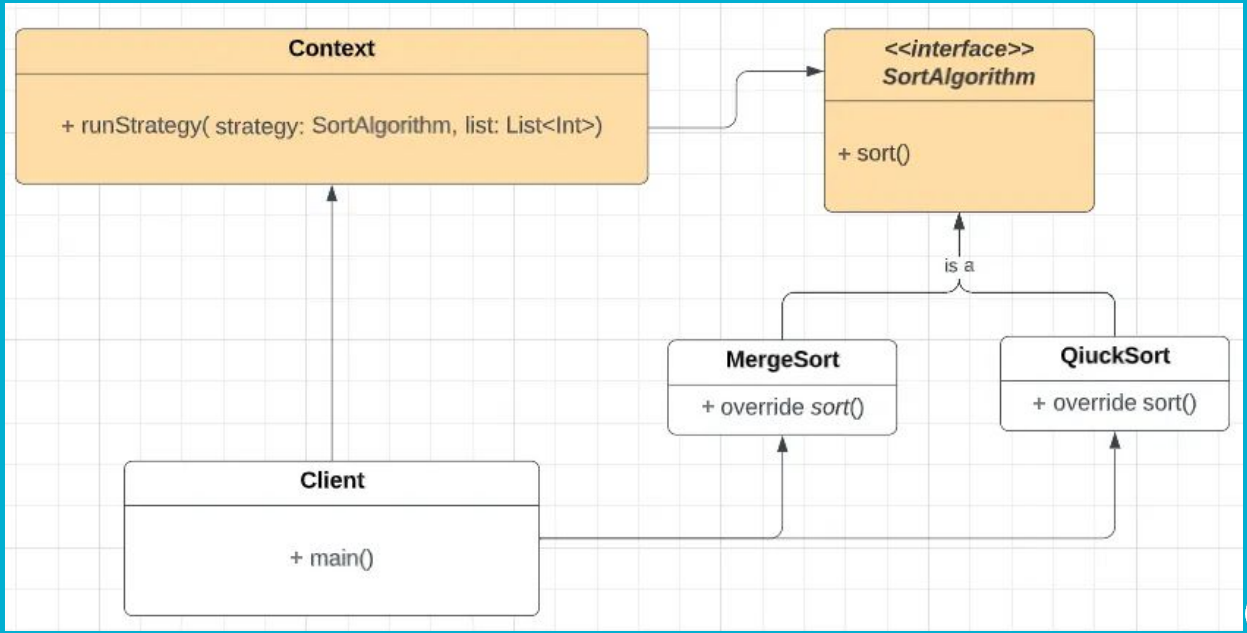
策略模式(Strategy Method Pattern)

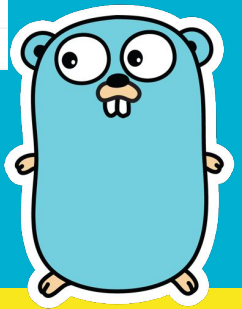
- 概念：定義一系列的演算法，並且把這些算法，用介面封裝到有公共介面的策略(strategy)類中，使他們能互相替換。
- 用策略的介面來替換在某個實體中的方法，可以經由替換不同的策略使得物件擁有不同的行為。
- 經過策略的組合，我們得以獲得行為不同的物件。



策略模式(Strategy Method Pattern)

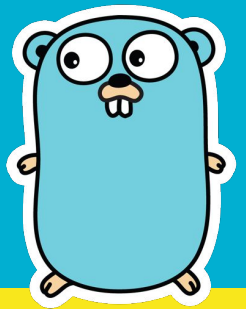
❖ 示意圖:





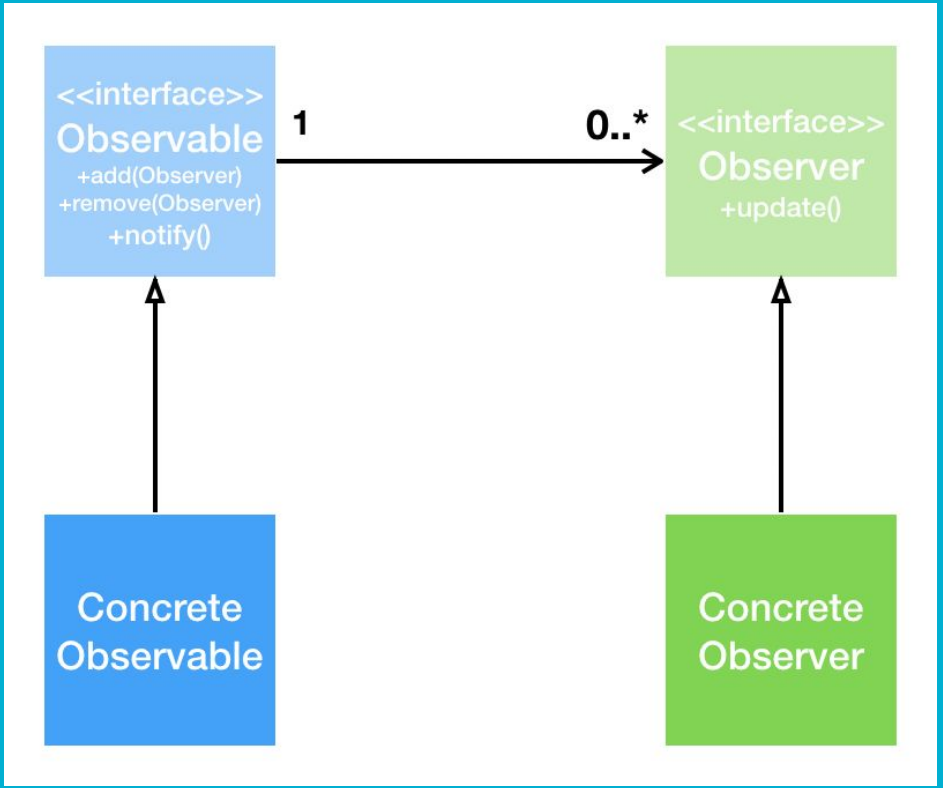
觀察者模式(Observer Method Pattern)

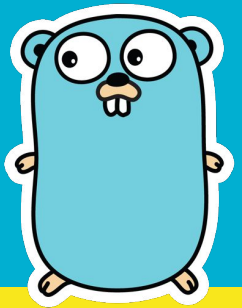
- 適用時機：當多個 Class 都需要接收同一種資料的變化時，就適合使用 Observer Pattern
  - 多個Class=>觀察者
  - 同一種資料=>主題
- 實作原理：
  - 把獲取資料的部分抽離出來，並在資料改變時，同步送給所有的觀察者。
  - 且觀察者可以在任何時候決定是否要繼續接收資料。



觀察者模式(Observer Method Pattern)

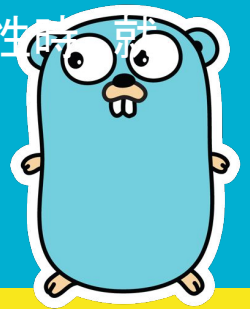
- 示意圖





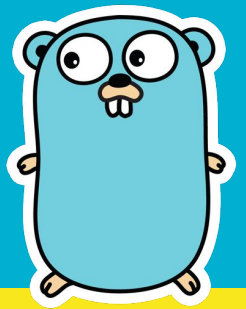
Design Pattern的哲學

- **保持簡單 (Keep It Simple)：**儘可能用簡單的方式解決問題。不要認為沒用設計模式就不是好設計，而是有時為了設計簡單而有彈性，需要使用設計模式。
- **設計模式非萬靈丹：**需要考慮使用設計模式時，對其他部份造成的影響。
- **要知道何時需要設計模式：**通常設計中需要改變的地方就需要採用設計模式。但加入模式是為了實際的改變，不是假定的改變。
- **重構的時間，就是模式的時間：**重構 (Refactoring) 是個好時機，可以檢視你的設計，是否能利用設計模式讓它有更好的結構。
- **拿掉不需要的設計模式：**當你的系統變得非常複雜，且不用預留任何彈性時，就可以拿掉模式。換句話說，有簡單的方案時，就可以不用設計模式。



Design Pattern的哲學

- 封裝(encapsulation)：物件要盡可能封裝複雜的邏輯
- 抽象(abstract)：實踐出所有方法
- 委派(delegate)：盡量委派任務給他人完成(不需要知道他是怎麼做的)



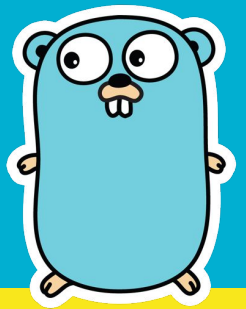
Lab

設計模式介紹與實戰

Go Lab-工廠模式

SPEC

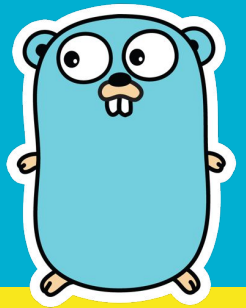
- 這是一款射擊遊戲
- 有多種槍枝, 且未來會不斷新增
- 槍枝的基本功能是名稱和威力
- 槍枝的特殊之處是聲音, 要針對每一種槍有不同的播音邏輯



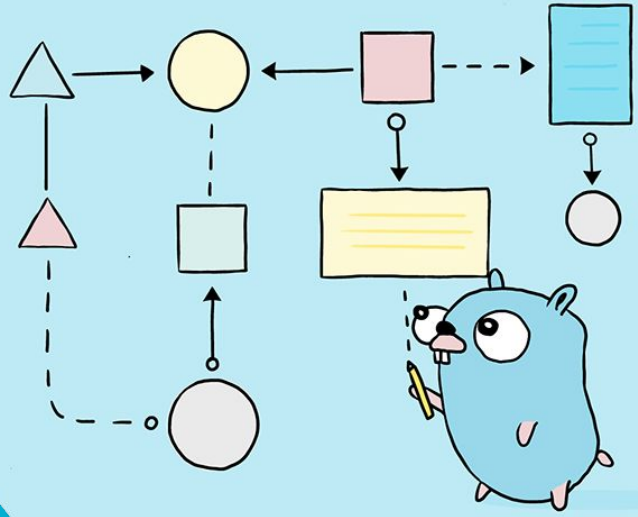
Go Lab-策略模式

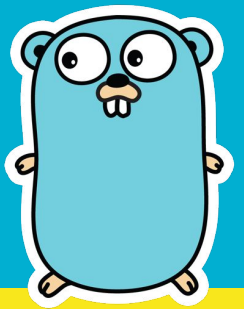
SPEC

- 這是 web server 的 in memory cache
- cache 每次塞滿都要挑選數據刪除
- 挑選的策略大幅度攸關於用戶的使用者體驗, 需要針對不同場景設計
- 未來也會持續增加與更新不同的策略
- cache 只需要具備新增與讀取數據的功能即可
- 本次只需實作 FIFO 和 LIFO



Q&A 時間





作業: 設計模式實作

1. 工廠模式
2. 策略模式

Link: https://github.com/leon123858/go-tutorial/tree/main/design-pattern