物件導向系統分析與設計
Object Oriented Analysis and Design

Design Principles

劉儒斌 Paladin R. Liu
paladin@ntub.edu.tw

AGENDA

- 物件導向設計原則
- 類別設計原則
- 類別庫設計原則

物件導向設計原則

物件導向設計原則

- **物件導向利用封裝，將資料及操作結合起來**
  - 讓物件可以自行管理資料及需要提供的功能
- **透過多型、抽象化、介面及繼承等機制運用到系統的設計上**
  - 降低物件彼此之間的耦合度
  - 讓系統在維護及功能擴充上，變的更容易

物件導向設計原則

- **這些機制帶來的好處：**
  - 提升設計的再使用性
  - 提高程式的維護性
- **概念雖然重要，但是並沒有提供設計時所應遵循的指引**
  - 該如何利用它們，進行系統的設計？
  - 有沒有比較好的設計準則？

類別設計原則

類別設計原則

- **建構軟體系統，與建築公司蓋房子一樣…**
  - 如果用了品質很差的磚塊來建造，可以想像得到，建造出來的房子，品質也不會好到哪裡去…
  - 假如採用了很好的材料來蓋房子，但是卻沒有好好的規劃房子的架構，還是有可能把這間房子搞得一團糟…

類別設計原則

- **SOLID 原則**
  - Robert C. Martin, Uncle Bob 提出
  - 協助我們好好的安排及規劃
    - 將類別裡的函數以及資料結構進行妥善的安排
    - 規劃類別之間的溝通、串聯關係
  - 讓軟體代碼的維護及擴充變的可能

類別設計原則

- **SOLID 原則**
  - 單一功能原則； Single Responsibility Principle, SRP
  - 開閉原則； Open-Close Principle, OCP
  - Liskov 替換原則； Liskov Substitution Principle, LSP
  - 介面分離原則； Interface Segmantation Principle, ISP
  - 相依反轉原則； Dependency Inversion Principle, DIP

單一功能原則

單一功能原則

- **對每一個模組、類別或函數，都應該對系統的一部份功能負責…**
  - 這個部份必需被封裝起來
  - 在這個模組、類別或函數中提供的服務，都要跟這個職責緊密的結合
- **單一功能原則與內聚力的概念有關**
  - 最難懂，也最容易誤解的原則

單一功能原則

- **一個模組都可以做一件事，且只能做這件事**
  - 這個原則的確存在
  - 用於把複雜的函數拆分為數個功能更單一的小函數
  - 但它不是 SRP

單一功能原則

- **一個模組都有一個、且唯一個可以修改功能的理由 / 原因**
  - 不應該同時存在有兩個或以上
  - 如果發生，就需要考慮把它們進行拆分
  - 思考的方向：這個程序需要對「**誰**」做出回應？

單一功能原則

- **範例：企業組織**
  - 想像一下，在企業由執行長帶領，執行長之下有…
    - 財務長：負責財務相關的管理
    - 營運長：負責公司的日常營運
    - 技術長：負責公司基礎技術架構
  - 他們各自負責的不同的工作與職責

```
public class Employee {
  // 依照員工的工作合約、狀態、工作時數，
  // 來計算該支付給員工的薪資
  public Money calculatePay();

  // 用來保存員工的相關資料
  public void save();

  // 計算員工的工作時數，返回一個字串並加入到報表，
  // 審查人員會確認員工是否上滿工作時數及領到正確的薪資
  public String reportHours();
}
```

單一功能原則

- **思考： calculatePay()**
  - 哪一位 C 字輩的經理人，需要定義這個功能？
  - 假設這件事搞砸了，誰要被殺頭？

單一功能原則

- **思考： calculatePay()**
  - 這個用來計算員工的薪水
  - 很顯然是 CFO 的責任
  - 如果因為這個功能的定義錯誤，而在同一個月份支付給員工兩份薪水…
    - 那這位 CFO 肯定要被開除的

單一功能原則

- **思考： reportHour()**
  - 哪一位 C 字輩的經理人，需要定義這個功能？
  - 假設這件事搞砸了，誰要被殺頭？

單一功能原則

- **思考： reportHour()**
  - 另一個管理者需要定義回傳字串的格式以及它的內容
  - 這個主管負責了行政審核以及監督的責任，很明顯的是 COO 的工作
  - 如果他把這一份資訊報告搞砸了…
    - 那 CEO 肯定會想把 COO 吊起來打

單一功能原則

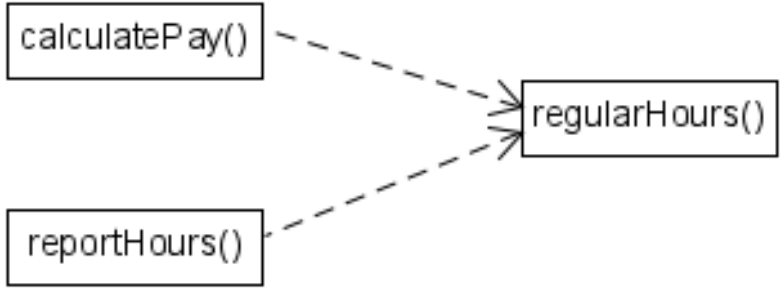
- **思考： save()**
  - 最後一個功能需要維持員工資料的保存及正確性
    - 很明顯的就是 CTO 的工作
  - 假如公司內部的資料庫系統崩潰了，所有的員工資料全部消失，沒有辦法救回…
    - 那麼 CTO 恐怕難辭其咎 (吊起來打可能都還不夠)

單一功能原則

- **讓我們做一下整理…**
  - 對於 calculatePay 方法，對這些更改的請求將來自以 CFO 為首的組織
  - 針對 reportHours 方法，相關的修改需求，將會來自 COO 的組織
  - 而與 save 相關的修改需求方法，則會來自 CTO 所管轄的組織
  - 因此，這個原則基本上與「人」有關
    - 更精確一點，一個角色

單一功能原則

- 假如：開發人員在功能上做了這樣的設計…
  - 在計算標準工時的功能上，共用了一個「 regularHours 」函數



單一功能原則

- **有一天，會計單位前來要求對標準工時的計算方式進行修改…**
  - 開發人員很快就找到，修改 regularHours()，就能完成
    - 但是他沒發現 reportHours() 也用了這個函數

單一功能原則

- **但 regularHours 原本的計算方式，對 COO 下屬的 HR 有特別意義…**
  - 所以他們不想改
  - 但他們不知道將會被調整…

單一功能原則

- 開發人員很有效率的完成了開發…
  - 功能也送給了會計單位驗證沒有問題
  - 敏捷的他們提交了這份修正到系統，同時部署到正式環境中
- 然後呢…？？

單一功能原則

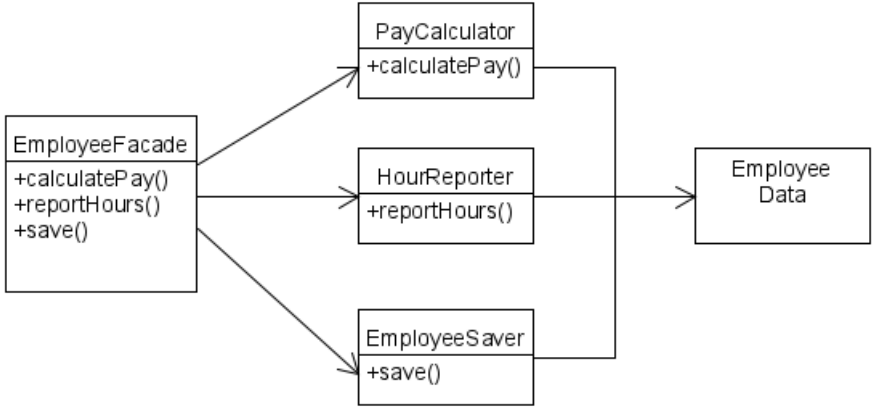
- **沒有人會想要自已負責事情，沒有獲得授權的情況下被更動**
  - COO 不會希望自己因為 CFO 的需求調整而被開除
  - 一旦發生過，未來 COO 可能會要求不能修改 reportHours 的相關功能

單一功能原則

- 再另一個例子：送車到修車廠修冷氣
  - 冷氣是修好了，但車也發不動了
  - 你會有什麼感覺？

單一功能原則

- **解決方案**
  - 獨立出三個彼此無關聯的類別
    - 分別負責不同部門的工作
    - 缺點是要透過不同的類才能完成原本的功能呼叫
  - 由一個 EmployeeFacade 類來統整相關功能



單一功能原則

把受**相同原因**發生變化的事物**收聚在一起**

試著將那些受**不同原因**而改變的事物**分開**

開閉原則

開閉原則

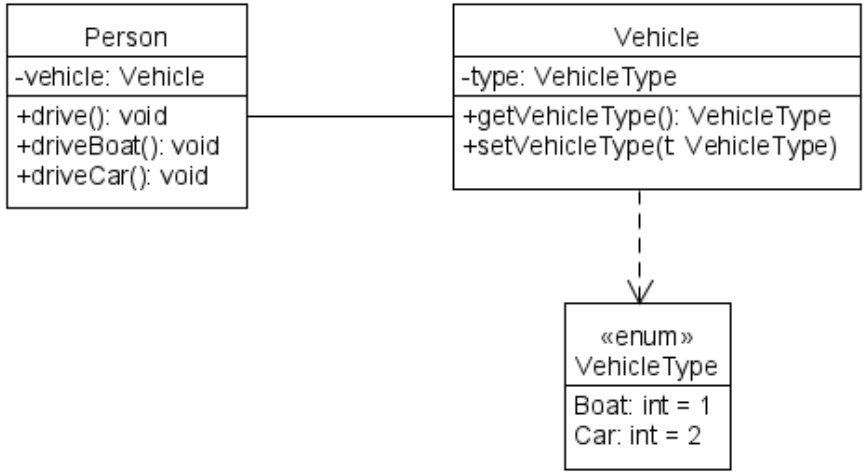
- **軟體中的模組和功能…**
  - 對擴充應該是開放的
  - 但是對於本身的功能修改則是封閉的
- **意謂著，在不改動原有的程式碼為前題下，增加系統的行為**

開閉原則

- 要如何實現這個原則呢？
  - 使用**介面**來**抽象化**相關的實作
  - 開發功能的過程裡，只需要**實作**這些**介面**
  - 假設功能需要進一步擴充
    - 透過**繼承**的方式擴充現有的**介面**，而不是直接修改它
  - 新的功能擴充，至少必須要實作原有的介面

開閉原則

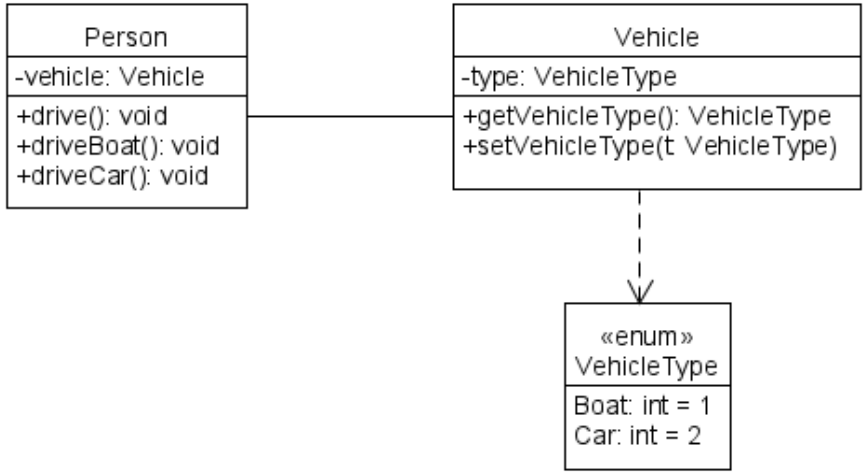
- 範例：出門開 XX 去
  - 到達目的地有很多方法
  - 可以開車
  - 也可能開船



參考課本 p.16-3 程式碼

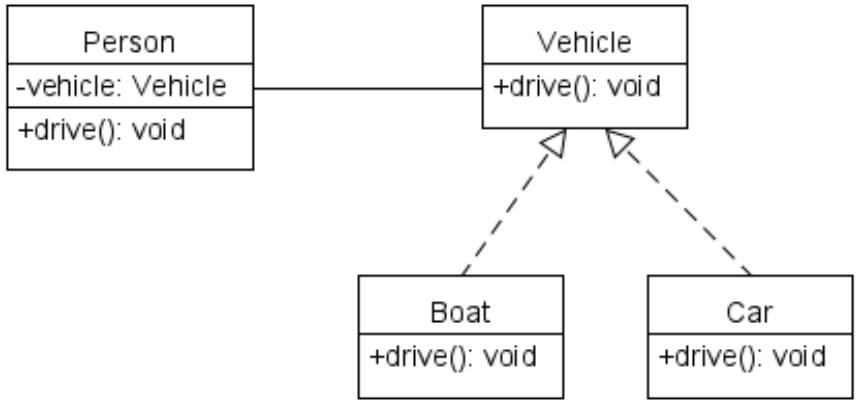
開閉原則

- 思考：這樣的設計有什麼問題？
  - 彈性？
  - 擴充性
    - 顯然，這樣的擴充性不好
    - 增加交通公具的種類，都會需要修改 Person 類



開閉原則

- 思考：有什麼方法可以改善？
  - 使用介面將關係抽象化
    - 透過實作 Vehicle 介面，就可以在不修改其他程式的情況下，進行功能擴充



LISKOV 替換原則

LISKOV 替換原則

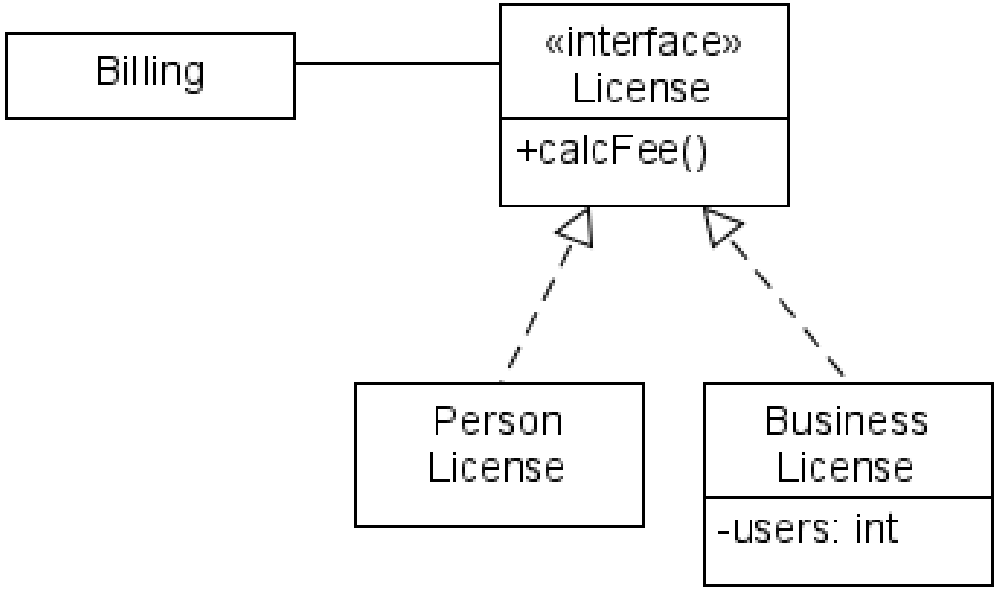
- 物件導向設計中，與繼承相關的一個重要的替代原則
- 假設 S 是類別 T 的子類別
  - 那麼在系統中，屬於類別 T 的物件，將可以被類別 S 所產生的物件替換 / 取代

LISKOV 替換原則

- **範例：軟體的授權費用計算**
  - 購買軟體，個人版、教育版、企業版的授權…
    - 費用不同
    - 限制不同
    - 計價方式也不同

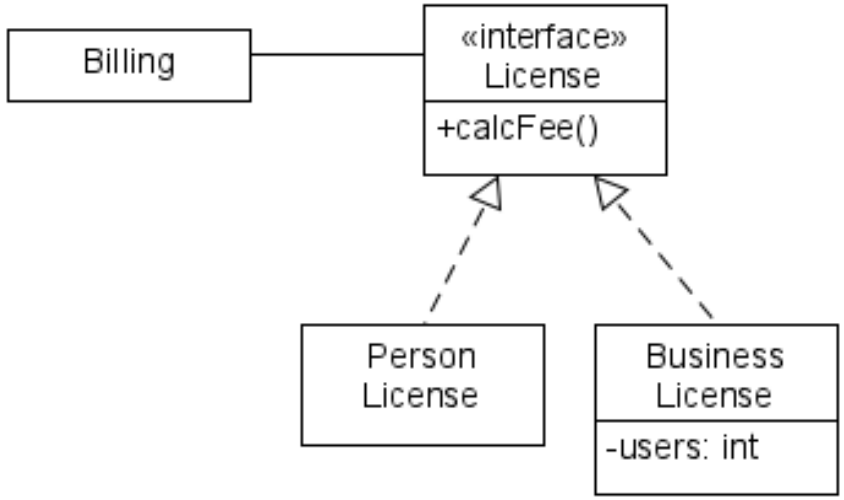
LISKOV 替換原則

- 思考：如何讓計價程式，可以在不同的條件下正常運作？



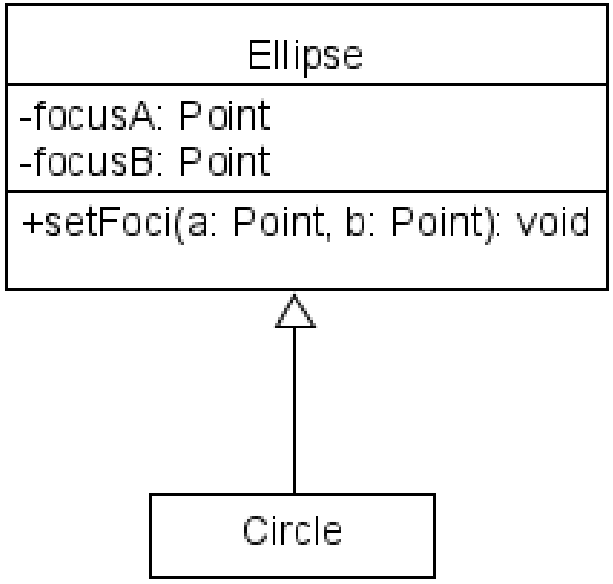
LISKOV 替換原則

- **思考:**
  - 符合 LSP 的要求
  - 讓 Billing 類與實際的計算類別脫勾
    - Person 與 Business License 都實作了 License
    - 因此它們都能直接替換 License 而不會發生問題



LISKOV 替換原則

- **思考：課本上的例子是反例 (p.16-6)**
  - 違反 LSP
    - 因為 Circle 雖然是 Ellipse 的子類別，但卻不能替換 Ellipse
  - 若在 client 的 f() 裡，以 if...else... 來改善，又違反了 OCP
    - client 的程式，在面對不同的 Ellipse 及它的子類，需要特別做判斷



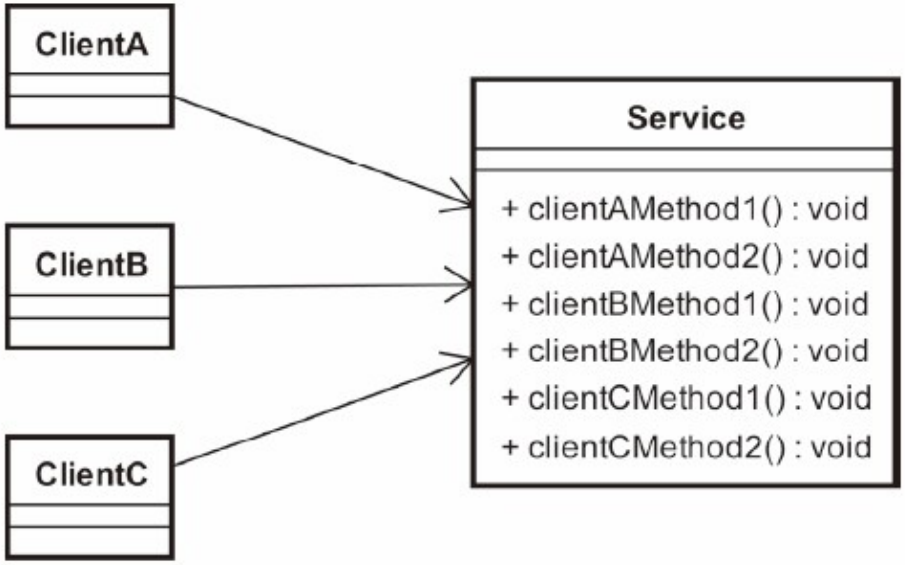
介面分離原則

介面分離原則

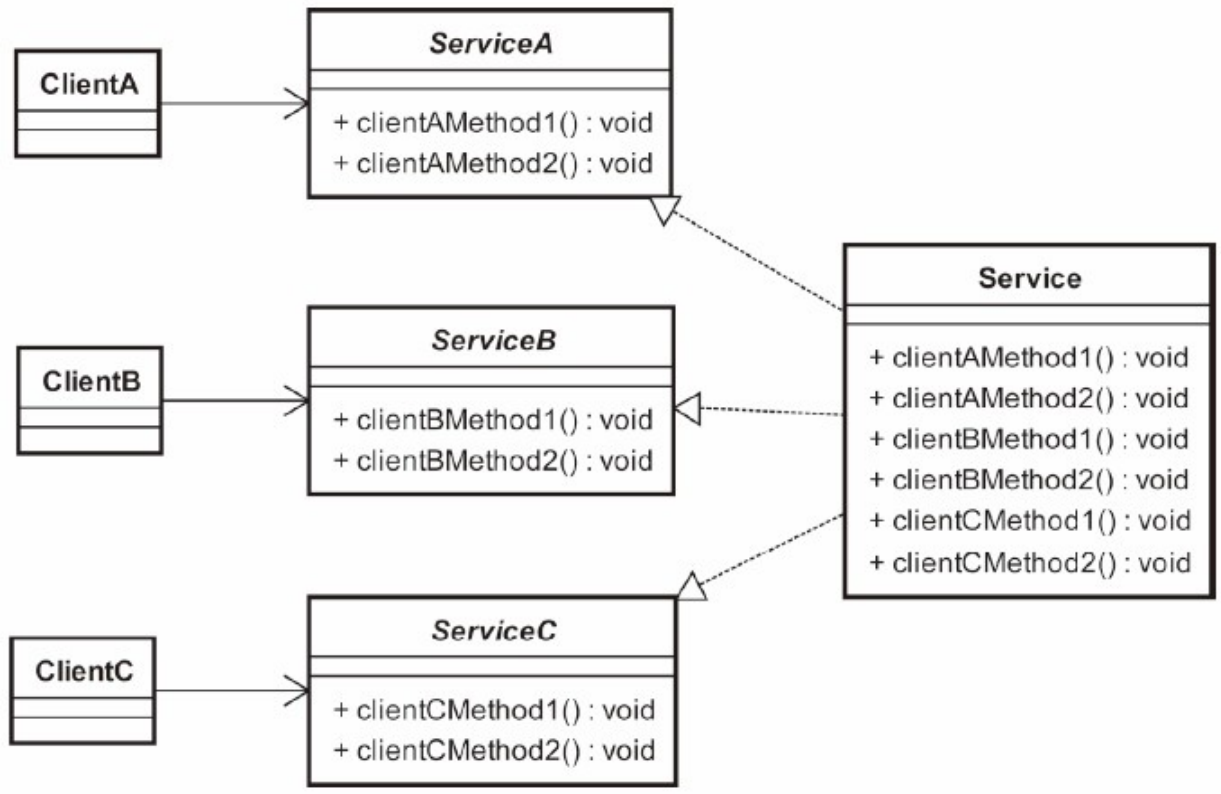
- **使用多個專門的介面比使用單一的總介面要好**
  - 客戶端（ client ）不應被迫使用對其而言無用的方法或功能
  - 拆分非常龐大臃腫的介面成為更小的和更具體的介面
  - 目的是系統解開耦合
    - 容易重構，更改和重新部署

介面分離原則

- **思考：大雜燴的問題**
  - 如果 ClientA 所使用到的介面需要更改
    - ClientB 及 ClientC 都可能被影響到
  - 遵循 ISP，透過介面來進行拆分



介面分離原則


ClientA
ServiceA
+ clientAMethod1() : void
+ clientAMethod2() : void
Service
+ clientAMethod1() : void
+ clientAMethod2() : void
+ clientBMethod1() : void
+ clientBMethod2() : void
+ clientCMethod1() : void
+ clientCMethod2() : void
ClientB
ServiceB
+ clientBMethod1() : void
+ clientBMethod2() : void
ClientC
ServiceC
+ clientCMethod1() : void
+ clientCMethod2() : void

介面分離原則

- **它所解決的問題…**
  - 在 C++ 或 Java 等靜態型別語言，若要使用前面的 Service，必須要宣告 include 或是 use 之類的關鍵字來進行引用
  - 假設前面的 Service 被修改了，那麼引用它的程式碼需要重新編譯部署
  - 在動態型別語言，例如 Python，對於 Service 的修改，則會在執行的過程中才去引用它
  - 因此 ISP 解決了靜態語言程式碼中的相依關係

相依反轉原則

相依反轉原則

- **依賴抽象，不要依賴具體**
  - 把 OCP 看成是描述達成物件導向設計的目標
    - 那 DIP 就是描述達成這個目標的主要機制
  - 依賴介面或抽象方法及類別的設計策略
    - 而不是依賴具體的類別或是具體的方法
  - 依賴注入 (Dependency Injection)

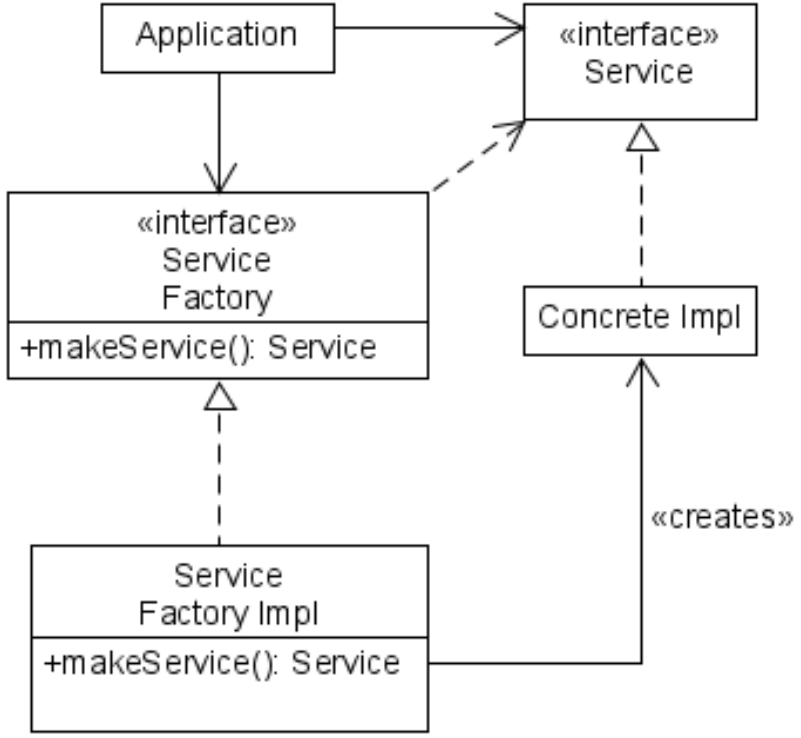
相依反轉原則

- **然而，過度的強調這原則是不切實際的…**
  - 例如， Java 中的 String 是 Concrete Class
    - 它相對很穩定，從 1.0 以來，很少改變
    - 試圖強制將與它之間的關係進行抽象化是不現實的
    - 不能，也不應該避免依賴於具體 java.lang.String
- **因此，只針對目前正在開發，且預期會頻繁更動的部份，套用 DIP**

相依反轉原則

- **套用 DIP 的幾個實踐**
  - 不要參照頻繁變動的具體類別
  - 不要試著從頻繁變動的具體類別上，衍生出其他類別
  - 不要試著複寫具體函數
    - 這樣子的函數中，可能會參考到其他的程式碼
    - 很難斷開其中的相依關係
  - 絕不在程式碼中，提到任何具體和經常異動類別的名字

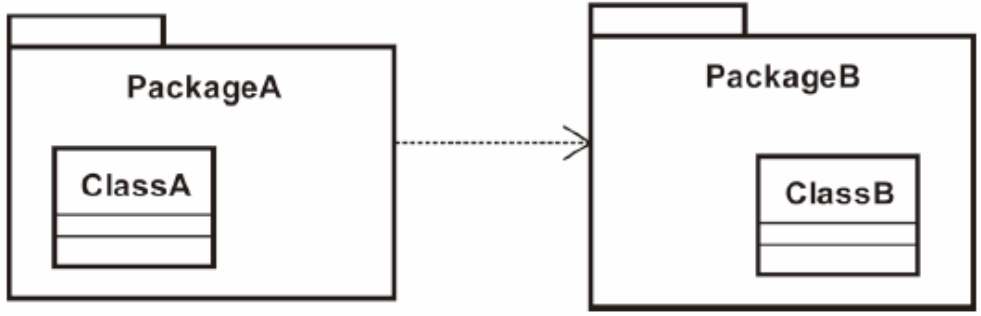
相依反轉原則


Application
«interface»
Service
«interface»
Service
Factory
+makeService(): Service
Concrete Impl
«creates»
Service
Factory Impl
+makeService(): Service

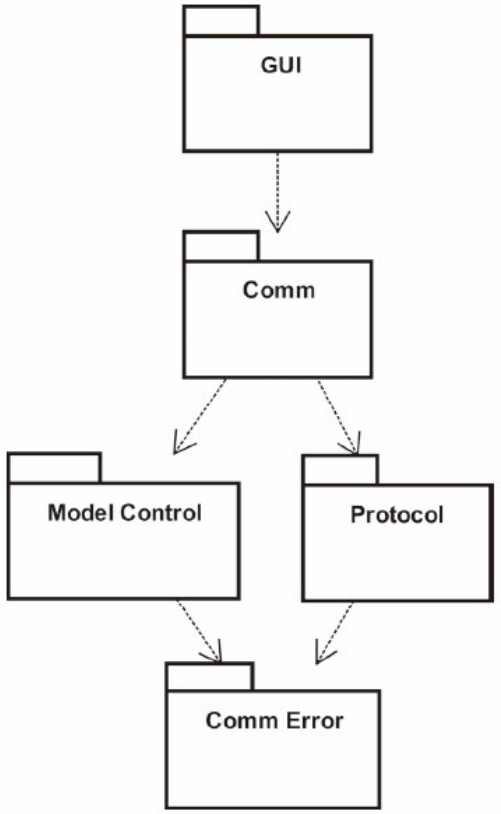
類別庫設計原則

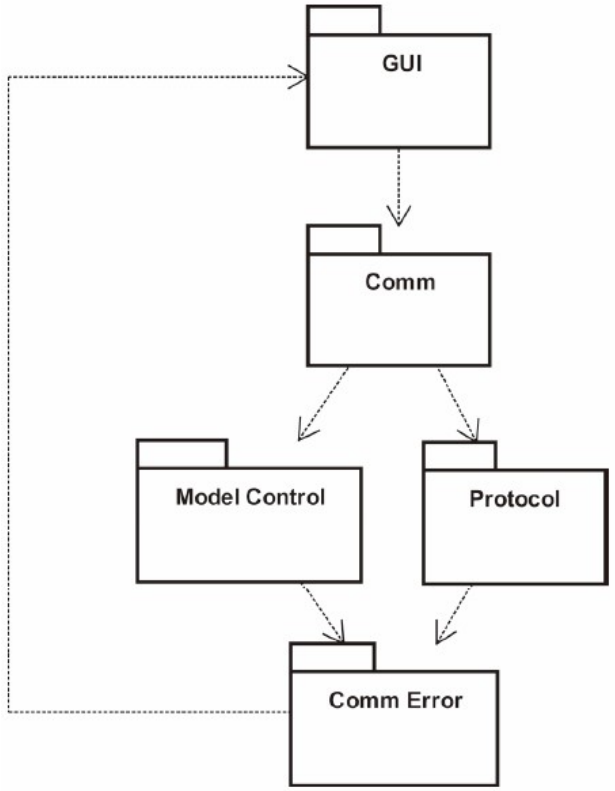
類別庫相依性

- 類別庫是一群類別的容器
  - 概念很類似作業系統中的檔案目錄
- 類別庫相依性，指的是類別庫中所含類別間的相依



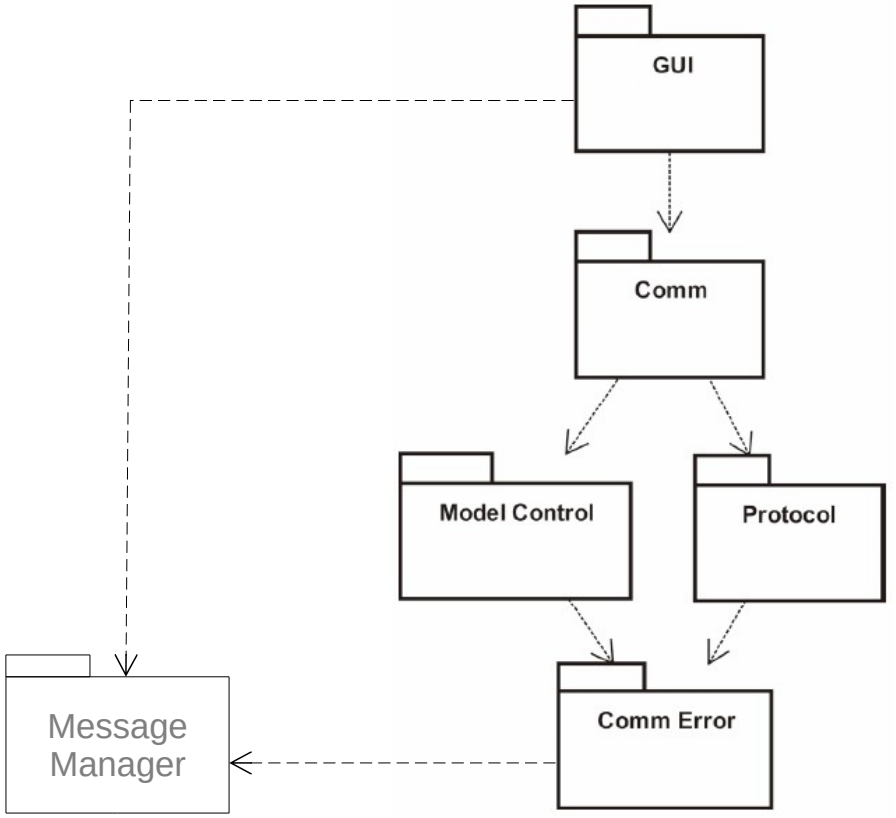
非循環相依原則


GUI
Comm
Model Control
Protocol
Comm Error


GUI
Comm
Model Control
Protocol
Comm Error

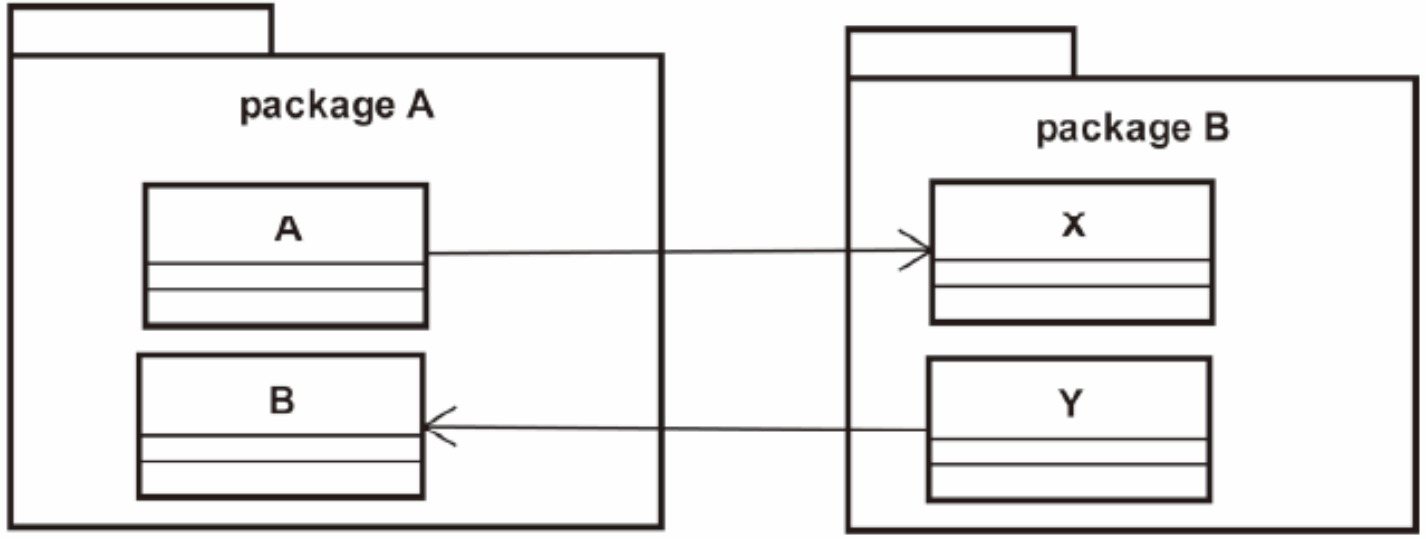
非循環相依原則

- 要避免類別庫間的相依關係造成循環
- 解決方法
  - 建立一個新的類別庫
  - 讓相關的循環消失



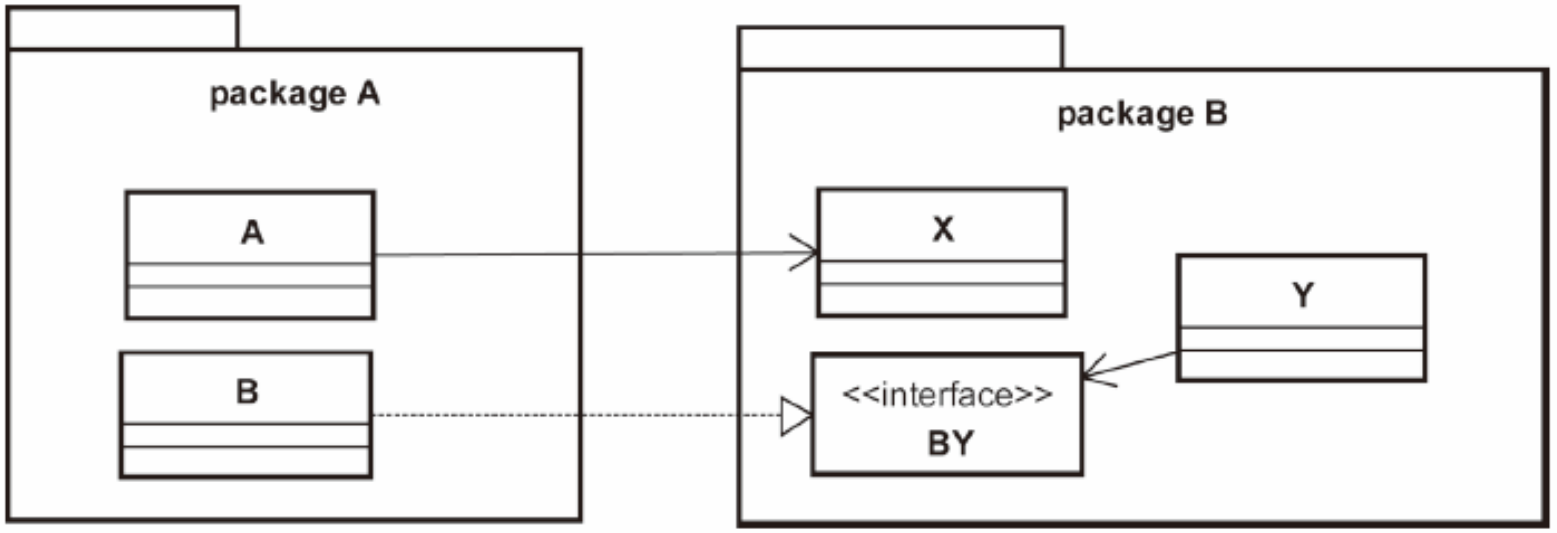
非循環相依原則

- 另一種類型的循環相依



非循環相依原則

- 透過 ISP 的概念，移除具體的相依



穩定相依原則

- 對於一個改變需要完成多少的工作？
  - 所謂穩定的軟體，即指它很難被改變
- 有很多的因素造成類別庫很難被改變
  - 其中之一就是相依性
  - 被許多其他的類別庫相依的類別庫是「穩定的」

穩定相依原則

- **更動頻率較頻繁的類別庫**
  - 應該有較少的進入相依
  - 較多的向外相依
- **更動頻率較不頻繁的類別庫**
  - 應該有較少的向外相依
  - 較多進入的相依

穩定的抽象化原則

- 除了穩定相依之外，更進一步…
  - 穩定的類別庫應該是抽象類別庫
  - 讓大部份的相依，都透過抽象類別庫來完成
    - 能夠有更好的彈性及維護性