The Single Responsibility Principle

Mikesc Huang

The Single Responsibility Principle (SRP)

The Single Responsibility Principle

- 首先要澄清一下，這裡的 SRP，不是指 refatoring 中，每個 function 只做一件事的那個 SRP

The SIngle Responsibility Principle

- 根據很威的康威定律(Conway's law[註1]), 軟體系統架構會受到公司組織或團隊間的溝通方式所影響

註1: Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.
- Melvin Conway

The SIngle Responsibility Principle

- 舉例來說
  - 只有一個團隊開發，團隊的溝通很容易，code 想改就改，整個系統就成為一個 module
  - 如果好幾個團隊一起開發，不同團隊之間的溝通沒那麼容易，code 不能說改就改，為了整合與擴充性，整個系統會包含數個 modules

The SIngle Responsibility Principle

- 基本上一個系統不會只有一個 module，不然這個系統大概只能稱為玩具
- 既然一個系統會包含數個 modules，那每個 module 只負責好自己的事，大家都會相安無事
- SRP 所提到的A class should have only one reason to change，就是說要把 module 的職責分清楚

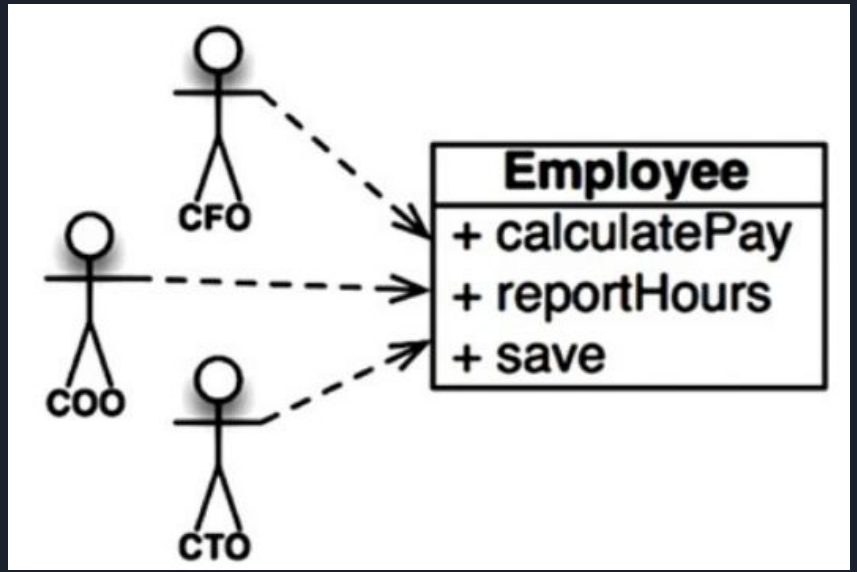
The Single Responsibility Principle

- 作者再進階用一句話說明 SRP
  - A module should be responsible to one, and only one, actor.
- 直接舉例說明

Symptom 1: Accidental Duplication

Symptom 1: Accidental Duplication

- 薪資系統中有個 Employee class



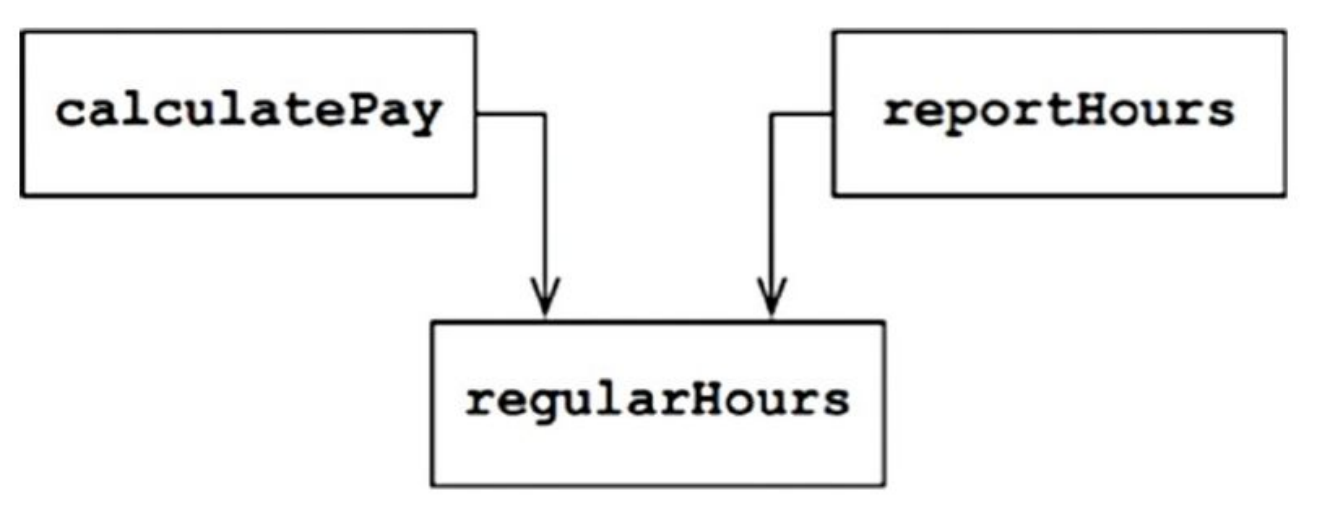
Symptom 1: Accidental Duplication

- 會計部門隸屬於 CFO，用到 calculatePay() 計算支出
- 人資部門隸屬於 COO，用到 reportHours() 計算工時
- 資料管理部門隸屬於 CTO，用到 save() 儲存員工資料

Symptom 1: Accidental Duplication

- 悲劇是這樣發生的
  - calculatePay() 與 reportHours() 是用同一個演算法來計算一般工時
  - 所以 RD 就建了一個 regularHours()
  - 為了避免重複的程式碼，所以兩個 method 都直接使用 regularHours()

Symptom 1: Accidental Duplication



Symptom 1: Accidental Duplication

- 有一天，CFO’s team 覺得要稍微修改一下工時計算方式，但 COO’s team 還是維持原來的工時計算方式
- RD trace code 發現 calculatePay() 有用到 regularHours()，那不就改這邊就好
- 於是 RD 就依照 CFO’s spec，修改 regularHours() 的計算方式

Symptom 1: Accidental Duplication

- 但 RD 沒注意到 reportHours() 也有用到 regularHours()
- 所以 COO’s team 得到的結果就錯了，這就是我們常遇到的，改 code 改出 side effect

Symptom 1: Accidental Duplication

- Employee class 違反了 SRP，因為三個 method 負責不同的 actor
- 解決方法就是把 code 拆開，拆成數個 class，每個 class 只負責對應單一 actor

Symptom 2: Merges

Symptom 2: Merges

- 再來一個悲劇
  - CTO’s team 要修改 Employee table
  - COO’s team 要修改工時報表的格式
  - 可能是不同 team 會修改同一個 Employee class, 所以彼此並不知道有誰同時在修改

Symptom 2: Merges

- 這樣的結果會造成一個 merge
- 雖然現在有強大的工具可以避免 merge，但畢竟不是萬能的，還是有風險

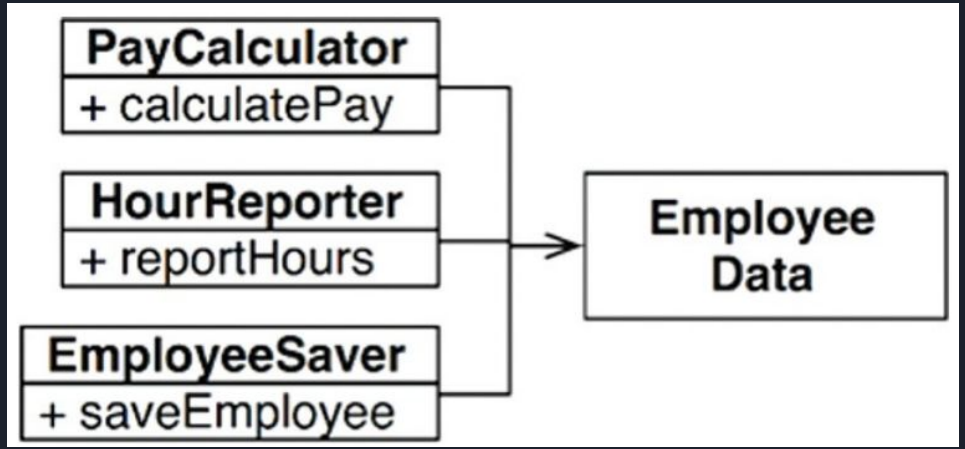
Symptom 2: Merges

- 這樣的悲劇例子有很多，但大部分都是因為改到一樣的檔案造成的
- 還是老話一句，把負責不同 actor 的 code 拆成不同 class 去負責處理

Solutions

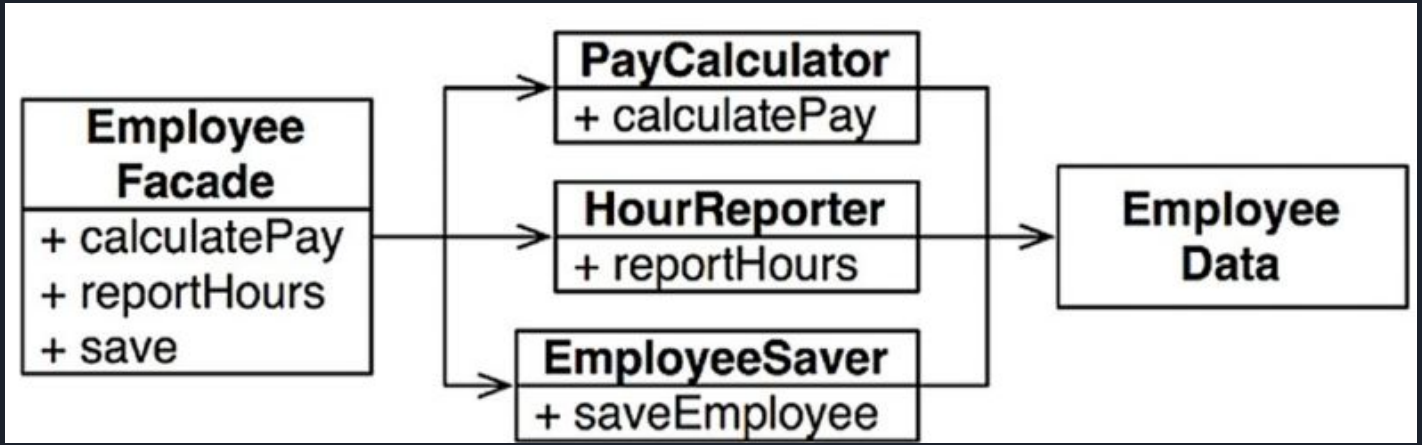
Solutions

- 建立三個 class，可以解決 accidental duplication
- EmployeeData 只放 data，沒有 method



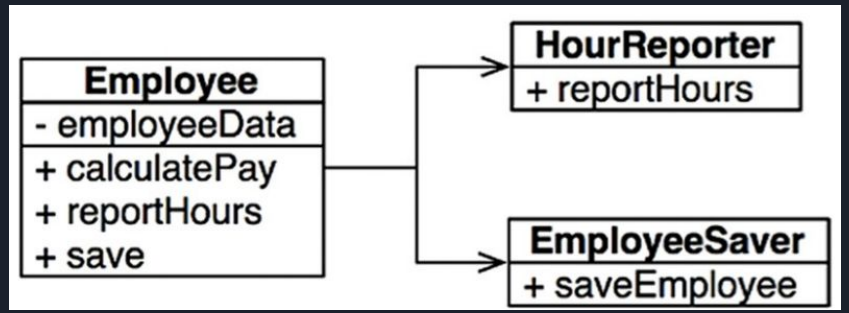
Solutions

- 對於 client 要同時處理三個 class 太複雜的話，常用的方法是 Facade Pattern



Solutions

- 有些人習慣把比較重要的 business rule 跟相關的 data 放在一起
- 把 EmployeeData 宣告在 Employee class 裡面，calculatePay() 也直接實作在 Employee class，其它不重要的還是透過 Facade Pattern



Conclusion

Conclusion

- 每個 module 只負責好自己的事, 大家都會相安無事
- 一個 module 負責太多事遲早會出事