http://teddy-chen-tw.blogspot.tw/2012/01/4.html

搞笑談軟工

敏捷開發，設計模式，精實開發，Scrum，軟體設計，軟體架構

2012 年 1 月 2 日 星期一

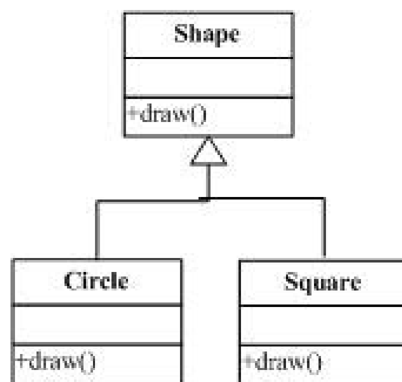
亂談軟體設計（4）：Liskov Substitution Principle

這一集輪到了 The Liskov Substitution Principle，內容請參考 Agile Software Development 這本第 111-125 頁：

> LSP: The Liskov Substitution Principle
> *Subtypes must be substitutable for their base types.*

首先說明一下，Liskov 指的是 Barbara Liskov 女士，這個原則是她在 1988 年所提出來的，在 Design by Contract 中（請參考 Object-Oriented Software Construction, 2nd 這本書），有一個規則叫做 Subcontracting rule 所要表達的意義與 LSP 是一樣的。「*Subtypes must be substitutable for their base types*」翻成白話文的意思就是說：「如果你的程式有採用繼承，或是定義了類似 Java 的 interface 然後提供若干個該 interface 的實作（implementation），則在你的系統中，只要是 base types（父類別或是 interfaces）出現的地方，都可以用 subtypes （子類別）或是該 interface 的實做來取代，而不會破壞程式原有的行為」

這樣解釋可能還是不太容易理解，看一張圖先：



這是一個很常見的繪圖系統設計，假設鄉民們要設計一個繪圖系統，可以畫圓（Circle）與正方形等形狀，於是鄉民們設計了一個父類別叫做 Shape（形狀），然後讓 Circle 與 Square 都繼承自 Shape。為了把這些圖形畫在螢幕上，鄉民們的程式中會有長成類似下圖的程式碼：

```
Shape myShape;
...
myShape = new Cirlce();
myShape.draw();
...
myShape = new Square();
myShape.draw();
```

程式中宣告 Shape 型態的 myShape 物件（instance），然後在 runtime（程式實行的時候）用這個 myShape 物件指到不同的子類別，然後再呼叫 draw method 就可以把各種不同的圖形畫在螢幕上。

上述作法其實是很基本的物件導向程式設計技術（polymorphism and dynamic binding）。這和 LSP 有什麼關係？因為上面這個設計符合 LSP，所以才能夠讓 Shape 物件（父類別）所出現的地方使用子類別（Circle 與 Square）取代。鄉民們看了 Robert C. Martin 對於 LSP 另一種解釋就比較容易理解：

LSP: The Liskov Substitution Principle

> Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

不知道鄉民們有沒有想過這樣一個問題：「為什麼寫程式的時候，可以宣告父類別的物件，然後在 runtime 的時候讓這個父類別的物件指到子類別的實例（instance）然後程式還是可以正確執行？」鄉民們想一下上面那段程式碼中， draw 這個 method 在父類別（Shape）中的「行為（behavior）」被定義為把圖形物件畫在畫面上。之所以可以做到「Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.」是因為子類別（Circle 與 Square）的 draw method 在實做的時候也遵循了原本父類別（Shape）對於 draw 的行為定義（把自己畫在畫面上）。如果子類別（Circle 與 Square）沒有遵循了原本父類別（Shape）對於 draw 的行為定義，那麼就不無法達到 LSP 的要求。

上面這段話需要思考一下，請鄉民們多看幾次。

那麼沒有達到 LSP 會怎樣？如果程式沒有達到 LSP 則程式的行為將變得「不可預測」，換句話說可能產生不可預知且不容易察覺的 bugs。 舉個例子，如果 Circle 物件的 draw method 沒以把自己畫在畫面上，而是把自己存到檔案中，或是輸出到印表機中，或是輸出到畫面的時候使用的座標和 父類別（Shape）所定義的座標系統不同。這樣的話，當鄉民們從 source code 看起來程式都是正常的，但是在真正執行的時候程式的行為卻不是自己所預期的（一位 Circle 沒有被輸出到畫面上，或是有輸出到畫面上但是位置卻不正確）。

看到這邊假設鄉民們知道 LSP 的意義以及違反 LSP 的後果，最後還剩下一個重要的問題：

如何定義程式的行為？

要達到 LSP 就表示子類別或是實做的行為要和父類別或是介面所定義的行為一致，問題是，平常宣告父類別或是介面的時候，只宣告 method 的 signature 而已啊，一般的程式語言並沒有提供什麼方法讓開發人員可以宣告「行為」啊？

沒錯，這就是為什麼 LSP 容易被忽略的地方。一般的物件導向程式語言（甚至是 C 語言）讓開發人員可以輕易做到「polymorphism and dynamic binding」但是卻沒有提供定義程式「行為」的機制。所以，Bertrand Meyer 才會提倡所謂的「Design by Contract（參考 Object-Oriented Software Construction, 2nd）」，建議開發人員幫每一個 method 定義 preconditions, postconditions 以及定義 class invariants 來規範程式的行為。有興趣深入了解的鄉民可以參考一下 Design by Contract 或是 Agile Software Development 這本書的 117-125。

還記得 GoF 在 design patterns 書中開宗明義所說的：

program to an interface, not an implementation

上面這個原則要能夠成立，interface 的實做就必須要遵守定義 interface 的人所期待的行為（因為 client 端只會透過 interface 看這個事件，並無法得知 runtime 時這個 interface 會被指派到那一個實做）。否則，就算是你的設計達到「program to an interface, not an implementation」也會因為沒有遵守 LSP 而破功。

友藏內心獨白：怎麼有種愈來越不好寫的預感。

張貼者： Teddy Chen 於 上午 11:31