軟件設計模式·復用·變化

2022-07-31 21:11:51

最近讀了《設計模式: 可復用面向對象軟件的基礎》（*Design Patterns: Elements of Reusable Object-Oriented Software*）一書，由埃里克·伽瑪（Erich Gamma）等著。此書英文版於 1995 年始發行。中文版我看的是機械工業出版社的版本(該出版社翻譯的書籍向來是詰屈聱牙的，這次也不例外)，該出版社於 2000 發行第一版，2019 年又發行了典藏版。本人借閱的正是這典藏版。

此書討論的主題是如何構建可復用的面向對象軟件，並引出 23 種設計模式。

本人閱讀此書後，結合自己以往設計的經驗作是文，發表自己的一些體悟。

## 什麼是設計模式

追本溯源，設計模式的概念是源自建築學的，特別是模式語言之父克里斯托弗·亞歷山大（Christopher Alexander）思想。他在1977年出版的 *A Pattern Language* 這樣說道：**每一個模式描述了一個在我們身邊不斷發生的重複的問題以及該問題的解決方案的核心。**

爲什麼我們要從建築學借來設計模式的概念呢？

首先，我們從軟件架構開始就是從建築架構中學來的，再借借設計模式的概念也不稀奇。更為重要的是，軟件設計過程中同樣會出現重複的問題，如果我們能從實踐中總結出解決該問題的核心，那麼以後我們的工作就只要識別出問題並復用對應的設計模式解決就好了，大大降低設計的難度。

## 面向對象軟件中的設計模式

面向對象設計就是將整個軟件系統拆分成一個對象集合。每個對象具備一定的職責，並與其他對象進行協作，共同完成軟件的功能。

在進一步討論面向對象軟件中的設計模式之前，有必要先釐清幾個面向對象軟件中的概念。

* **對象**。對象就是數據以及對數據的操作的結合體。數據就是對象的屬性，在代碼中稱之爲成員變量；對數據的操作就是對象的行爲，在代碼中稱之爲成員方法。
* **對象的接口**。接口是對象的接口，是對象行為的抽象，是對象成員方法的簽名(或稱型構)的集合。
* **對象的類型**。類型是對象所實現接口的標識。一個對象可實現多個接口，因而可表現爲多類型。
* **接口的繼承**。一個接口可以包含另一個接口，實現接口方法的擴展。另一個接口類型稱爲這個接口類型的超類型(supertype)，而這個接口類型是另一個接口類型的子類型(subtype)。
* **多態**。運行時可替換具有相同接口的對象，此種特性稱之爲多態。接口是對象間交互的協議，多態者就是允許針對接口協議有不同的實現。
* **類**。類是對象的描述，描述對象的數據及操作。類通過實例化成爲對象。
* **類的繼承**。一個類可以包含另一個類，包括成員變量和方法。被包含者稱之爲父類，反之稱之爲子類。子類以白盒方式知曉父類的可見的變量和方法，並可以重寫之、擴展之。

面向對象軟件中復用代碼的機制有兩種： 1. **類繼承**。類繼承是一種白盒復用，子類可以知曉父類的內部細節。 2. **對象組合**。對象組合是一種黑盒復用，一個對象組合了另一個對象，但並不知曉該對象內部的實現細節。

值得注意的是，復用代碼的機制很多，我們這裡討論的是面向對象軟件中的復用代碼機制。比如，我定義一個公共函數，每次需要我就調用這個函數，這同樣實現了代碼復用，只不過不是面向對象中的復用。

優良的設計總是「**高內聚、低耦合**」的。所謂「內聚」是指對象職責的內聚，便於復用，而「耦合」是指一個對象對另一個對象細節的鈍感程度，越遲鈍耦合越低，就越能應對未來的變化。

我們常說「**組合優於繼承**」，就是出於這樣的考慮。組合對被組合對象內部細節無感，因而是低耦合的；而繼承反之，是高耦合的。我們還說「**面向接口編程，而不是面向實現編程**」，也是這個道理。接口是對象方法的抽象，是對象間交互的協議，因爲多態的緣故，可以方便地使用新的實現動態綁定，因而是低耦合的；反之，面向具體的實現則是高耦合的。

設計模式如何幫助我們在進行面向對象設計時實現「高內聚、低耦合」的優良設計呢？

我的理解是分離變和不變的部分，讓未來可能變化的部分能夠變化。用四個字說就是「**封裝變化**」。

變化總是在進行的，無視變化將造成軟件的大災難，即**重新設計**。重新設計好比建築過程中的拆掉重建，是一種很大的浪費和損耗。設計模式要求在復用代碼的同時考慮可變的部分，對可變的部分進行封裝（或者說抽象），爲未來的變化留下可能。

最常見的變化有： \* 依賴的具體實現類； \* 依賴的軟硬件平台； \* 依賴的第三方代碼庫； \* 依賴的具體算法等等。

當依賴具體時，要知道具體未來可預知的變化是否在可接受範圍內，否則當具體變化時，很可能需要重新設計。當依賴抽象時，具體如何變化都不影響調用。

## 是否有面向過程軟件中的設計模式

伽馬書中沒有討論面向過程軟件中的設計模式。

在這裡我想擴展下，按照設計模式的概念，應該是有的。設計模式是實踐中總結出來的應對重複問題的解決方案的核心，不管是用面向對象設計還是面向過程設計，都是存在設計模式的。

同樣，問題的核心還是在於識別和分離變化。區別在於兩者使用的工具不同：面向過程使用函數封裝變化，面向對象使用接口或抽象類封裝變化。

舉個例子，我們知道貓和人類跑的方式是不一樣的，貓是四條腿跑的，而人是兩條腿跑的。這是一個變化項，需要封裝。

在面向對象設計中，我們通常會封裝出一個 Runner 接口，並讓貓和人類實現該接口，達到封裝變化的目的。下面以 Go 語言爲例，編碼如下：

// 首先有一個 Runner接口  
type Runner interface {  
 Run()  
}  
  
// Cat 實現了 Runner接口  
type Cat struct{}  
func (c Cat) Run() {  
 println("Cat Run")  
}  
  
// Human 實現了 Runner接口  
type Human struct{}  
func (h Human) Run() {  
 println("Human Run")  
}  
  
func main() {  
 var runner Runner  
 runner = Cat{}  
 runner.Run()  
 // 現在可以方便地更換 Runner實現  
 runner = Human{}  
 runner.Run()  
}

在面向過程設計中，我們可以使用函數封裝。面向對象設計中的接口是對象方法簽名的集合，同樣我們可以使用函數簽名抽離出跑步行爲，使用 Go 語言編碼如下：

func main() {  
 // Cat 的 Run行爲  
 runner := func() {  
 println("Cat Run")  
 }  
 Run(runner)  
  
 // Human 的 Run行爲  
 runner = func() {  
 println("Human Run")  
 }  
 Run(runner)  
}  
  
// Run 方法跟具體的 Runner無關  
func Run(runner func()) {  
 runner()  
}

其實，面向對象設計中的接口最小的粒度（除了空接口）即單個方法簽名，跟面向過程中的單個函數簽名是一致的。函數即面向對象設計中的方法。

面向對象設計中接口的概念可以很好地控制抽象的粒度（接口中包含的方法的多寡），這是面向過程設計所不具備的。

## 再談「復用」

讀完本文，你是否對伽馬書中所說的「復用」有更多的理解？

在我看來，書中的「復用」是有多重含義的：

一是指**模式的復用**。不斷用之前總結的設計模式來解決不斷出現的重複的問題； 二是指**對象的復用**。使用繼承、組合等方式實現對象的復用，不斷重用不變的代碼。