前言

很多年前，我在 Google 工作時，同事們都非常重視程式品質，對程式品質的追求甚至到了「吹毛求疵」的程度，程式註解中一個小小的標點符號錯誤都會被指出並要求改正。而正是得益於對程式品質的嚴格把關，專案的維護成本變得非常低。

離開 Google 之後，我任職過多家公司。很多國內的企業，包括很多頂尖的互聯網公司，都不是很重視程式品質。因為需求多、時間少，所以專案負責人往往只關心團隊開發了多少功能，並不關心程式寫得是好還是壞。在開發中，很少有人寫單元測試程式，也沒有 CodeReview 環節，程式能用即可。在這種「快、糙、猛」的研發氛圍以及「爛」程式的「薰陶」下，很多工程師都沒有時間和心思，更沒有能力去寫高品質的程式。

在清楚地認識到國內開發現狀之後，我就有了寫本書的打算，希望將我多年積累的開發經驗彙集成一本書，幫助那些對程式品質有追求的程式設計師。

本書的書名為《設計模式之美》，不過書名有點「以偏概全」，因為本書不僅僅講解設計模式，而是以寫高品質程式為主旨，全面講解了與此有關的 5 個方面：物件導向程式設計範例、設計原則、程式規範、重構技巧和設計模式。

儘管市面上有很多講解如何寫高品質程式的圖書，但大部分圖書為了在簡短的篇幅內將重點講清楚，大多選擇比較簡單的程式範例，這就導致很多讀者在讀完這些書之後，感覺理論知識都懂了，但仍然不知道如何將理論知識應用到真實的專案開發中。因此，在本書寫作的過程中，我竭盡全力讓本書的講解更加貼近實戰。

在權衡篇幅和學習效果的情況下，對於每個重點，我都結合真實的專案程式來做講述，並且，側重講解本質的或貼近應用的知識，例如，為什麼會有這種設計模式？它用來解決什麼樣的程式設計問題？應用時有何利弊需要權衡？等等。讓讀者知其然，知其所以然，並學會應用。

實際上，我還出版過一本資料結構和演算法相關的圖書——《資料結構與演算法之美》。在寫那本書時，我希望做到「一本在手，演算法全有」。參考那本書的寫作風格，對於本書，我希望做到「一本在案，程式不爛」，透過閱讀本書，讀者能夠全面、系統地掌握寫高品質程式所需的所有技能！

本書內容

本書分為 8 章，每章包含的主要內容如下：

第 1 章 為概述，簡單介紹了本書涉及的各個模組，以及各個模組之間的聯繫。本章作為全書的開篇，可以幫助讀者建構系統的知識體系。

第 2 章 介紹物件導向程式設計範例。物件導向程式設計範例是目前流行的一種程式設計範例，是設計原則、設計模式寫程式實作的基礎。

第 3 章 介紹設計原則，包括 SOLID 原則、KISS 原則、YAGNI 原則、DRY 原則和 LoD 原則。

第 4 章 介紹程式規範，主要包括命名與註解、程式風格，以及程式設計技巧。

第 5 章 介紹重構技巧，包括重構四要素、程式的可測試性、單元測試和解耦等。

第 6 章 介紹建立型設計模式，包括單例模式、工廠模式、生成器模式和原型模式。

第 7 章 介紹結構型設計模式，包括代理模式、修飾模式、配接器模式、橋接模式、外觀模式、組合模式和享元模式。

第 8 章 介紹行為型設計模式，包括觀察者模式、模板方法模式、策略模式、責任鏈模式、狀態模式、迭代器模式、訪問者模式、備忘錄模式、命令模式、直譯器模式和中介模式。

注意，儘管書中大部分程式以 Java 寫，但本書講解的重點與具體的程式設計語言無關。

本書內容適合熟悉任何程式設計語言的讀者。

1 概述

撰寫本書的主要目的是幫助讀者寫高品質的程式。在正式學習程式設計的方法論之前，我們有必要先弄清楚一些與程式品質有關的問題，如什麼是高品質的程式。

本章可以作為本書的大綱或學習框架，幫助讀者有系統地瞭解本書涉及的重點。

1.1 為什麼學習程式設計

雖然本書的書名是《設計模式之美》，但本書並不僅講解設計模式，還包括一系列與程式設計相關的知識，如物件導向程式設計範例、設計原則、程式規範和重構技巧等。如果說資料結構和演算法可以幫助讀者寫出高效能程式，那麼程式設計相關的知識可以幫助讀者寫出可擴展、可讀和可維護的高品質程式。上述重點可以直接應用到平時的開發中，對它們的掌握程度直接影響程式設計師的開發能力。不過，有些讀者認為，這些程式設計相關的知識像「屠龍刀」，看起來很厲害，但平時的開發根本用不上。基於這種觀點，我就具體談一下為什麼要學習程式設計。

1.1.1 寫高品質的程式

我相信，軟體工程師都很重視程式品質，畢竟誰也不想寫出被人詬病的「爛」程式。但是，就我的瞭解來看，毫不誇張地講，很多軟體工程師，甚至一些知名互聯網公司的員工，寫的程式都不盡如人意。一方面，在目前很多盲目追求速度的開發環境下，很多軟體工程師並沒有太多時間去思考如何寫高品質的程式；另一方面，在「爛」程式的影響和沒有人指導的情況下，很多軟體工程師不太清楚高品質程式到底應該是什麼樣子。

這導致很多軟體工程師雖然寫了多年程式，但功力沒有太大長進，對於寫的程式，只追求「能用即可，能執行就好」。很多軟體工程師一直在重複勞動，工作多年，但能力只停留在初級工程師的水準。

儘管我已經工作近十年，但一直在程式語言的第一線工作，現在每天都在堅持寫程式、審查同事寫的程式、重構遺留系統中的「爛」程式。在這些年的工作中，我見過太多的「爛」程式，如不規範命名方式、類別設計不合理、分層不明確、沒有模組化概念、程式結構混亂和高度耦合等。維護這樣的程式非常費力，因為增加或修改一個功能，經常會牽一髮而動全身，維護者無從下手，恨不得將全部程式刪除並重寫！

如何提高寫高品質程式的能力呢？首先要對程式設計方面的理論知識有一定的了解。理論知識既是前人智慧的結晶，又是解決問題的工具。沒有理論知識，相當於遊戲時沒有厲害的「武器裝備」，一定會影響到自身水準的發揮。

1.1.2 應對複雜程式的開發

軟體發展的難度通常會展現在兩個方面：一方面是技術難，程式量不一定大，但要解決的問題比較難，如自動駕駛、影像辨識和高性能訊息佇列等，需要用到比較高深的技術或演算法，不是依靠「人海戰術」就能完成的；另一方面是複雜，技術不高深，但專案龐大、商業複雜、程式量大和參與開發的人多，如物流系統、財務系統和大型 ERP 系統等。「技術難」方面涉及細分專業領域的知識，與本書介紹的程式設計主題無關，因此，我們圍繞「複雜」方面來展開，即如何應對軟體發展的複雜性問題。

大多數的軟體工程師均具備可以輕鬆寫出簡單的「Hello World」程式的能力，幾千行的程式基本上維護不成問題。隨著程式從幾萬行、十幾萬，達到幾十萬行，甚至上百萬行，這時候軟體的複雜度呈指數級提升。在這種情況下，我們不僅要求程式可以執行、正確執行，還要求寫的程式易懂和可維護。此時，程式設計相關的知識就有了用武之地，真正成為軟體工程師手中的「屠龍刀」。

大部分軟體工程師熟悉程式設計語言、開發工具和開發框架，他們的日常工作就是在使用框架，根據商業需求填充程式。我剛出社會的時候，也是做這類事情。其實，這樣的工作並不需要我們具備很強的程式設計能力，只要理解商業，並將商業翻譯成程式就可以了。但是，當我的上司突然安排了一個與商業無關的通用功能模組的開發任務時，面對這樣一個稍微複雜的程式的設計和開發任務，我發現就有點力不從心，不知從何下手。單純實作功能並做到程式可用可能並不複雜，但要寫出可用又好用的程式，其實並不容易。

如何分層和分模組？如何劃分類別？每個類別有哪些屬性和方法？怎麼設計類別之間的互動？應該使用繼承還是組合？應該使用介面還是抽象類別？怎樣做到解耦，以及高內聚、低耦合？應該使用單例模式還是靜態方法？應該使用工廠模式建立物件還是直接用 new 建立？如何在引入設計模式提高擴展性的同時避免帶來可讀性降低問題？這一系列問題是我之前都沒有思考過的。

而當時的我對程式設計並沒有太多的知識儲備和經驗積累，有些手足無措。正因如此，我意識到了程式設計方面的重要性，在之後的很多年，一直刻意鍛鍊自己的程式設計能力。面對複雜程式的設計和開發，我也越來越得心應手。

1.1.3 程式設計師的基本功

對於程式設計師，技術的積累既要有廣度，又要有深度。其實，很多人早早意識到了這一點，在學習框架、中介軟體時，會抽空研究相關原理，並閱讀原始碼，希望能在深度上有所認識，而不只是略知皮毛，會用而已。

從我的經驗和同事的回饋來看，在看原始碼的時候，有些人經常看不懂，或者無法堅持看下去。讀者是否遇過這種情況？實際上，這個問題的原因很簡單，那就是基本功還不夠，自身能力還不足以完全看懂這些程式。

優秀開源專案、框架和中介軟體的程式量與類別的數量都比較大，類別結構、類別之間的關係都極其複雜，呼叫關係也是錯綜複雜。因此，為了確保程式的擴展性、靈活性和可維護性等，程式中會使用較多的設計模式和設計原則。如果讀者不理解這些設計模式和設計原則，那麼，在閱讀程式時，就可能不能完全理解我的設計思維。對於一些意圖明顯的設計思維，這些讀者可能需要花費很長時間才能參悟。如果讀者對設計模式和設計原則非常瞭解，一眼就能看出程式如此設計的原因，那麼閱讀程式就會變得輕鬆。

實際上，除看不懂、無法堅持看下去的問題以外，還有一個隱藏問題：讀者認為自己看懂了，實際上，並沒有理解程式的精髓。優秀的開源專案、框架和中介軟體就像一個集各種頂尖技術於一身的「戰鬥機」。如果想剖析它的原理、學習它的技術，在沒有深厚的基本功的情況下，就算把這台「戰鬥機」擺在我們面前，我們也不能完全理解它的精髓，只是瞭解了「皮毛」而已。

因此，程式設計相關的知識是程式開發的基本功，不僅能讓我們輕鬆地讀懂開源專案，還能幫助我們瞭解程式中的技術精髓。

1.1.4 職場發展的必備技能

初級開發工程師只需要學會熟練操作框架、開發工具和程式設計語言，再做幾個專案練習，基本上就能應付平時的開發工作。但是，如果讀者不想一輩子只當個初級工程師，想成長為高級工程師，希望在職場中獲得更高的成就和更好的發展，就要重視基本功的訓練和基礎知識的積累。

我們發現，一些優秀軟體工程師寫的程式相當「優雅」。如果我們只是將框架用得很好，聊起架構時頭頭是道，但程式寫得很「爛」，那麼我們永遠都不會成為優秀的軟體工程師。

在技術這條職場道路上，當我們成長到一定階段之後，勢必要承擔一些培養和指導技術新人與初級工程師，以及 Code Review 的工作。如果我們自己都對什麼是高品質的程式，以及不瞭解如何寫出高品質的程式，那麼又該如何指導別人？如何讓他人信服？

還有，當我們成長為技術上司之後，需要負責專案的整體開發工作，為開發進度、開發效率和專案品質負責。我們不希望團隊堆砌「垃圾」程式，讓整個專案變得無法維護，新增、修改一個功能都很困難，最終拉低了整個團隊的開發效率。

除此之外，程式品質低還會導致線上 bug 頻繁發生，檢查困難，整個團隊陷在不斷修改無意義的低級 bug、修補爛程式之類的事情中。而一個設計良好、易維護的系統，可以讓我們有時間去做更加有意義的事情。

1.1.5 思考題

請讀者談一下對學習程式設計相關知識的重要性的看法。

1.2 如何評價程式品質

在我的工作經歷中，每當同事評論專案程式品質的時候，我聽到最多的評論是「程式寫得很爛」或「程式寫得很好」。我認為，用「好」、「爛」這樣的字眼來描述程式品質是非常籠統的。當我詢問程式到底「爛」在何處或「好」在哪裡時，儘管大部分同事都能簡單地舉幾個「爛」或好的例子，但他們的回答往往都不夠全面，重點零碎，也無法切中要害。

當然，也有一些軟體工程師對如何評價程式品質有所認識，如認為好程式是易擴展、易讀、簡單、易維護的，等等，但他們對於這些評價的理解往往只停留在表面上，對於諸多更加深入的問題，如「怎麼才算可讀性好？什麼樣的程式才算易擴展、易維護？可讀、可擴展與可維護之間有什麼關係？可維護中的「維護」兩字該如何理解？」，等等，他們並沒有太清晰的認識。

實際上，對於程式品質的描述，除了簡單籠統「好」或「爛」之外，還有很多語義豐富、專業和細化的描述方式，包括：

靈活性（flexibility）、可擴展性（extensibility）、可維護性（maintainability）、可讀性（readability）、可理解性（understandability）、易修改性（changeability）、可複用性（reusability）、可測試性（testability）、模組化（modularity）、高內聚低耦合（high cohesion loose coupling）、高效率（high effciency）、高性能（high performance）、安全性（security）、相容性（compatibility）、易用性（usability）、簡潔（clean）、清晰（clarity）、簡單（simple）、直接（straightforward）、少即是多（less code is more）、文件詳盡（well-documented）、分層清晰（well-layered）、正確性（correctness、bug free）、強健性（robustness）、可靠性（reliability）、可伸縮性（scalability）、穩定性（stability）和優雅（elegant）等。

面對如此多的名詞，我們到底應該使用哪些名詞來描述一段程式的品質呢？

實際上，我們很難透過其中的某個或某幾個名詞來全面地評價程式品質，因為這些名詞是從不同角度描述程式品質的。例如，在評價一個人的時候，我們往往透過多個方面進行綜合評價，如性格、能力等，否則，對一個人的評價可能是片面的。同樣，對於程式品質，我們也需要綜合多種因素進行評價，不應該從單一的角度去評價。例如，一段程式的可擴展性很好，但可讀性很差，那麼，我們不能片面地認為這段程式的品質高。

注意，不同的評價角度並不是完全獨立的，有些之間存在包含關係、重疊關係等，或者可以互相影響。例如，程式的可讀性和可擴展性好，可能意味著程式的可維護性好。而且，各種評價角度不是「非黑即白」。例如，我們不能簡單地將程式評價為可讀或不可讀。如果用數字來量化程式的可讀性，那麼應該是一個連續的區間值，而非 0、1 這樣的離散值。

不過，我們真的可以客觀地量化一段程式的品質嗎？答案是否定的。對一段程式品質的評價，常常帶有很強的主觀性。例如，對於什麼樣的程式才算是可讀性好，每個人的評判標準都不一樣。

正是因為程式品質評價的主觀性，使得這種主觀評價的準確度與軟體工程師自身的經驗有極大的關係。軟體工程師的經驗越豐富，提供的評價往往越準確。形成差異的是，資歷較淺的軟體工程師常常覺得沒有一個可量化的評價標準作為參考，很難準確判斷一段程式的品質。如果無法辨別程式寫得好或壞，那麼，即使寫再多的程式，寫程式能力也可能沒有太大提高。

在仔細閱讀前面舉例的程式品質評價標準之後，讀者會發現，有些名詞過於籠統、抽象，而且偏向於對整體的描述，如優雅、好、壞、整潔和清晰等；有些過於注重細節、偏重方法論，如模組化、高內聚低耦合、檔案詳盡和分層清晰等；有些可能並不僅僅局限於寫程式，與架構設計等也有關係，如可伸縮性、可複用性和穩定性等。

為了讀者有重點地進行學習，我挑選了 7 個常用且重要的評價標準來詳細講解，包括可維護性、可讀性、可擴展性、靈活性、簡潔性、可複用性和可測試性。

1.2.1 可維護性（maintainability）

對於程式開發，「維護」無外乎修改 bug、修改舊的程式和增加新的程式等。「程式易維護」是指，在不破壞原有程式設計、不引入新的 bug 的情況下，能夠快速修改或增加程式。「程式不易維護」是指，修改或增加程式需要冒極大引入新 bug 的風險，並且需要很長的時間才能完成。

對於一個專案，維護程式的時間可能遠遠大於寫程式的時間。軟體工程師可能將大部分時間花在修復 bug、修改舊的功能邏輯，和增加新的功能邏輯之類的工作上。因此，程式的可維護性就顯得格外重要。

對於維護、易維護和不易維護這 3 個概念，我們不難理解。不過，對於實際的軟體發展，更重要的是需要清楚如何判斷程式可維護性的高低。

事實上，可維護性是一個難以量化、偏向對程式整體進行評價的標準，它類似之前提到的「好」、「壞」、「優雅」之類的籠統評價。程式的可維護性高低是由很多因素共同作用的結果。程式簡潔、可讀性好、可擴展性好，往往就會使得程式易維護。更深入地說，如果程式分層清晰、模組化程度高、高內聚低耦合、遵守基於介面而非實作程式設計的設計原則等，就可能意味著程式易維護。除此之外，程式的易維護性還與專案的程式量、商業的複雜程度、技術的複雜程度、檔案的全面性和團隊成員的開發水準等諸多因素有關。

1.2.2 可讀性（readability）

軟體設計專家 Martin Fowler 曾經說過：「Any fool can write code that a computer can understand. Good programmers write code that humans can understand. 」（任何人都可以寫電腦能理解的程式，而好的程式設計師能夠寫人能理解的程式。）在 Google 內部，有一個稱為「Readability」的認證。只有拿到這個認證的軟體工程師，才有資格在 Code Review 的時候批准別人提交的程式。可見，程式的可讀性有多麼重要，畢竟，程式被閱讀的次數有時候遠遠超過被寫和執行的次數。

程式的可讀性如此重要，在寫程式的時候，我們要隨時考慮程式是否易讀、易理解。程式的可讀性很大程度上會影響程式的可維護性，因為無論是修復 bug 還是增加 / 修改功能程式，我們首先要讀懂程式。如果我們對程式一知半解，就有可能因為考慮不周而引入新 bug。

既然程式的可讀性如此重要，那麼我們如何評判一段程式的可讀性呢？

我們需要查看程式是否符合程式規範，如命名是否達意、註解是否詳盡、函式長度是否合適、模組劃分是否清晰，以及程式是否「高內聚、低耦合」等。除此之外，Code Review 也是一個很好的測試程式可讀性的手段。如果我們的同事可以輕鬆地讀懂我們寫的程式，往往能夠說明我們的程式的可讀性不差；如果同事在讀我們寫的程式時有很多疑問，那麼可能在提示我們，程式的可讀性存在問題，需要重點關注。

1.2.3 可擴展性（extensibility）

程式的可擴展性是指在不修改或少量修改原有程式的情況下，能夠透過擴展方式增加新功能程式。換句話說，程式的可擴展性是指在寫程式時預留了一些功能擴展點，我們可以把新功能程式直接插入擴展點，而不會因為增加新的功能程式而改動大量的原始程式。可擴展性也是評價程式品質的重要標準。程式的可擴展性表示程式應對未來需求變化的能力。與程式的可讀性一樣，程式是否易擴展也在很大程度上決定了程式是否易維護。

1.2.4 靈活性（flexibility）

靈活性也可以用來描述程式品質。例如，我們經常會聽到這樣的描述：「程式寫得很靈活」。那麼，我們如何理解這裡提到的「靈活」呢？

儘管很多人用「靈活」描述程式品質，但實際上，「靈活」是一個抽象的評價標準，給「靈活」下定義是很難的。不過，我們可以想一下，我們在什麼情況下才會說程式寫得很靈活呢？

我舉了 3 種情境，幫助讀者理解什麼是程式的靈活性。

1）當我們增加新功能程式時，由於原有程式中已經預留了擴展點，因此，我們不需要修改原有程式，只需要在擴展點上增加新程式。這個時候，我們除了可以說程式易擴展之外，還可以說程式寫得很靈活。

2）當我們要實作一個功能時，如果原有程式中已經提取出了很多位於底層且可複用的模組、類別等，那麼我們可以直接使用。這種情況下，我們不僅可以說程式易複用，還可以說程式寫得很靈活。

3）當我們使用某個類別時，如果這個類別可以應對多種使用情境，滿足多種不同需求，那麼，我們除可以說這個類別易用以外，還可以說這個類別設計得很靈活或程式寫得很靈活。

從上述情境來看，如果一段程式易擴展、易複用，或者易用，我們一般可以認為這段程式寫得很靈活。因此，「靈活」的含義廣泛，很多情境都可以使用。

1.2.5 簡潔性（simplicity）

有一條非常著名的設計原則，大部分讀者應該都聽過，那就是 KISS 原則：「Keep It Simple，Stupid」。該原則的意思是「儘量保持程式簡單」。程式簡單、邏輯清晰往往意味著程式易讀、易維護。在寫程式的時候，我們通常會把「簡單、清晰」原則放到首位。

不過，很多程式設計經驗不足的程式設計師會覺得，簡單的程式沒有技術含量，喜歡在專案中引入一些複雜的設計模式，覺得這樣才能體現自己的技術水準。實際上，思從深而行從簡，真正的程式設計高手往往能用簡單的方法解決複雜的問題。

除此之外，雖然我們都能認識到，程式要儘量寫得簡潔，要符合 KISS 原則，但怎樣的程式才算足夠簡潔？怎樣的程式才算符合 KISS 原則呢？實際上，不是每個人都能準確地做出判斷，因此，在第 3 章介紹 KISS 原則的時候，我們會透過具體的程式範例詳細說明。

1.2.6 可複用性（reusability）

我們可以將程式的可複用性簡單地理解為「儘量減少重複的程式，複用已有程式」。在後續章節中，我們會經常提到「可複用性」這一程式評價標準。例如，當介紹物件導向特性的時候，我們會提到繼承、多型存在的目的之一，就是提高程式的可複用性；當介紹設計原則的時候，我們會提到單一職責原則與程式的可複用性相關；當介紹重構技巧的時候，我們會提到解耦、高內聚和模組化等能夠提高程式的可複用性。可見，可複用性是一個重要的程式評價標準，也是很多設計原則、設計思維和設計模式等所要實作的最終效果。

實際上，程式的可複用性與 DRY（Don't Repeat Yourself）原則的關係緊密，因此，在第 3 章介紹 DRY 原則的時候，我們還會介紹程式複用相關的更多知識，如提高程式的可複用性的程式設計方法等。

1.2.7 可測試性（testability）

相較於上述 6 個程式品質評價標準，程式的可測試性較少被提及，但它同樣重要。程式可測試性的高低可以從側面準確地反映程式品質的高低。程式的可測試性低，難以寫單元測試，那麼，基本能夠說明程式的設計有問題。關於程式的可測試性，我們將在重構部分（見 5.3 節）詳細講解。

1.2.8 思考題

除本節提到的程式品質評價標準，還有哪些程式品質評價標準？讀者心目中的高品質程式是什麼樣子的呢？

1.3 如何寫出高品質程式

每位軟體工程師都想寫出高品質程式，那麼，如何才能寫出高品質程式呢？在 1.2 節中，我們提到了 7 個常用且重要的程式品質評價標準。高品質的程式也就等同於易維護、易讀、易擴展、靈活、簡潔、可複用、可測試的程式。

想要寫出滿足上述程式品質評價標準的高品質程式，我們需要掌握一些細化、可落地的程式設計方法論，包括物件導向設計範例、設計原則、程式規範、重構技巧和

設計模式等。而掌握這些程式設計方法論的最終目的是寫出高品質的程式。這些程式設計方法論是後續章節講解的重點內容，我們先熟悉一下它們。

本節相當於本書的一個學習框架，先簡單介紹後續章節涉及的重點，使讀者對全書有整體性瞭解，幫助讀者將後面零散的知識重點在大腦中有系統地組織起來。

1.3.1 物件導向

目前，程式設計範例或程式設計風格主要有 3 種：程序導向、物件導向和函式語言程式設計。物件導向程式設計風格是其中的主流。現在流行的程式設計語言大部分屬於物件導向程式設計語言。另外，大部分專案也都是基於物件導向程式設計風格開發的。物件導向程式設計因其具有豐富的特性（封裝、抽象、繼承和多型），可以實作很多複雜的設計思維，所以，它是很多設計原則、設計模式寫程式實作的基礎。

對於物件導向，讀者需要掌握下面 7 個重點（詳見第 2 章）。

1）物件導向的四大特性：封裝、抽象、繼承和多型。

2）物件導向程式設計與程序導向程式設計的區別和聯繫。

3）物件導向分析、物件導向設計和物件導向程式設計。

4）介面和抽象類別的區別，以及各自的應用情境。

5）基於介面而非實作程式設計的設計思維。

6）「多用組合，少用繼承」設計思維。

7）程序導向的「貧血」模型和物件導向的「充血」模型。

1.3.2 設計原則

設計原則是程式設計時的一些經驗總結。設計原則有一個特點：這些設計原則看起來比較抽象，定義描述比較模糊，不同的人對同一個設計原則會有不同的解讀。因此，如果我們單純地記憶它們的定義，那麼對程式設計、設計能力的提高並沒有太大幫助。對於每一種設計原則，我們需要掌握它能解決什麼問題和應用情境。只有掌握這些內容，我們才能在專案中靈活、恰當地應用這些設計原則。實際上，設計原則是心法，設計模式是招式。因此，設計原則比設計模式普適、重要。只有掌握了設計原則，我們才能清楚地瞭解為什麼使用某種設計模式，並且恰到好處地應用設計模式，甚至還可以創造新的設計模式。

對於設計原則，讀者需要理解並掌握下列 9 種原則（詳見第 3 章）。

1）單一職責原則（SRP）。

2）開閉原則（OCP）。

3）里氏替換原則（LSP）。

4）介面隔離原則（ISP）。

5）依賴反轉原則（DIP）。

6）KISS 原則、YAGNI 原則、DRY 原則和 LoD 法則。

1.3.3 設計模式

設計模式是針對軟體發展中經常遇到的一些設計問題而總結的一套解決方案或設計思維。大部分設計模式解決的是程式的解耦、可擴展性問題。相對於設計原則，設計模式沒有那麼抽象，而且大部分不難理解，程式實作也並不複雜。對於設計模式的學習，我們需要重點掌握它們能夠解決哪些問題和典型的應用情境，並且不過度使用。

隨著程式設計語言的演進，一些設計模式（如單例模式）逐漸過時，甚至成為反模式，一些設計模式（如迭代器模式）則被內建在程式設計語言中，還有一些新設計模式出現，如單態模式。

在本書中，我們會重點講解 22 種經典設計模式，它們分為三大類：建立型、結構型和行為型。在這 22 種設計模式中，有些設計模式常用，有些設計模式很少被用到。對於常用的設計模式，我們要花費多一些時間理解和掌握。對於不常用的設計模式，我們瞭解即可。

按照類型，我們對本書中提到的設計模式進行了簡單的分類。

1）建立型設計模式：單例模式、工廠模式（包括簡單工廠模式、工廠方法模式、抽象工廠模式）、生成器模式和原型模式。

2）結構型設計模式：代理模式、修飾模式、配接器模式、橋接模式、外觀模式、組合模式和享元模式。

3）行為型設計模式：觀察者模式、模板方法模式、策略模式、責任鏈模式、狀態模式、迭代器模式、訪問者模式、備忘錄模式、命令模式、直譯器模式和中介模式。

1.3.4 程式規範

程式規範主要解決的是程式的可讀性問題。相對於設計原則、設計模式，程式規範更加具體且偏重程式細節。如果軟體工程師開發的專案並不複雜，那麼可以不必瞭解設計原則和掌握設計模式，但起碼需要熟練掌握程式規範，如變數、類別和函式的命名規範，程式註解的規範等。因此，相較於設計原則、設計模式，程式規範基礎且重要。

不過，相對於設計原則、設計模式，程式規範更容易理解和掌握。學習設計原則和設計模式需要融入很多個人的理解和思考，但學習程式規範並不需要。每條程式規範都非常簡單且明確，讀者只要照著做即可，所以，本書並沒有花費太大篇幅講解所有的程式規範，而是總結了我認為能夠有效改善程式品質的 17 條規範。

除程式規範以外，我還會介紹一些程式的「壞味道」，幫助讀者瞭解什麼樣的程式是不符合規範的，以及應該如何最佳化。參照程式規範，讀者可以寫出可讀性高的程式；在瞭解了程式的「壞味道」後，讀者可以找出程式存在的可讀性問題。

1.3.5 重構技巧

在軟體發展中，只要軟體不停迭代，就沒有一勞永逸的設計。隨著需求的變化，程式的不停堆砌，原有的設計必定存在問題。針對這些問題，我們需要對程式進行重構。重構是軟體發展中的重要環節。持續重構是保持程式品質不下降的有效手段，能夠有效避免程式「腐化」到「無可救藥」的地步。

重構的工具有物件導向程式設計範例、設計原則、設計模式和程式規範。實際上，設計原則和設計模式的重要應用情境就是重構。我們知道，雖然設計模式可以提高程式的可擴展性，但過度或不恰當地使用它，會增加程式的複雜度，影響程式的可讀性。在開發初期，除非必要，我們一定不要過度設計，應用複雜的設計模式，而是當程式出現問題的時候，我們再針對問題，應用設計原則和設計模式進行重構，這樣就能有效避免前期的過度設計問題。

關於重構，本書重點講解以下 3 方面的內容。透過對這些內容的講解，希望讀者不但可以掌握一些重構技巧，更重要的是建立持續重構意識，把重構當作開發的一部分，融入日常的開發中。

1）重構的目的（why）、物件（what）、時機（when）和方法（how）。

2）保證重構不出錯的技術手段：單元測試，以及程式的可測試性。

3）兩種不同規模的重構：大重構（大規模，高層次）和小重構（小規模，低層次）。

下面總結一下物件導向程式設計、設計原則、設計模式、程式規範和重構技巧的關係。

1）物件導向程式設計範例因其豐富的特性（封裝、抽象、繼承和多型），可以實作很多複雜的設計思維，所以，它是很多設計原則、設計模式寫程式實作的基礎。

2）設計原則是指導程式設計的一些經驗總結，是程式設計的心法，指明了程式設計的大方向。相較於設計模式，它更加無處不在。

3）設計模式是針對軟體發展中經常遇到的一些設計問題而總結的一套解決方案或設計思維。應用設計模式的主要目的是解耦，提高程式的可擴展性。從抽象程度上來講，設計原則比設計模式更抽象。設計模式更加具體，更加容易落地執行。

4）程式規範主要解決程式可讀性問題。相較於設計原則、設計模式，程式規範更加具體、更加偏重程式細節和更加可落地執行。持續的小重構主要依賴的理論就是程式規範。

5）重構作為保持程式品質不下降的有效手段，依靠的就是物件導向程式設計範例、設計原則、設計模式和程式規範這些理論知識。

實際上，物件導向程式設計範例、設計原則、設計模式、程式規範和重構技巧都是保持或提高程式品質的方法論，本質上都是服務於寫高品質的程式這一件事。當我們看清這個本質之後，很多選擇如何做就清楚了。例如，在某個情境下，是否使用某個設計模式，判斷的標準就是能否能夠提高程式品質。

想要寫高品質的程式，除了累積上述理論知識以外，我們還需要進行一定強度的刻意訓練。很多程式設計師提到過，雖然學習了相關的理論知識，但是容易忘記，而且在遇到問題時想不到對應的重點。實際上，這就是缺乏理論結合實踐的刻意訓練。例如，在上學的時候，老師在講解完某個重點之後，往往配合講解幾題範例，然後讓我們透過課後習題來強化這個重點。這樣，當我們再次遇到類似問題時，就能夠立即想到相應的重點。

除掌握理論知識、刻意訓練之外，具備程式品質意識也非常重要。在寫程式之前，我們要多思考未來有哪些擴展需求，哪部分程式是會變的，哪部分程式是不變的，這樣寫程式會不會導致以後增加新功能時比較困難、程式的可讀性不高等問題。具備了這樣的程式品質意識，也就離寫出高品質的程式不遠了。

1.3.6 思考題

結合自己的工作，讀者認為本節介紹的哪一部分內容能夠有效提高程式品質？讀者還知道哪些提高程式品質的方法？

1.4 如何避免過度設計

我們常說，一定要重視程式品質，寫程式之前，不要忽略程式設計環節。實際上，不做程式設計不好，過度設計也不好。在我過往的工作經歷中，遇到過很多同事，特別是開發經驗比較少的同事，喜歡對程式進行過度設計，濫用設計模式。在開始寫程式之前，他們會花很長時間進行程式設計。對於簡單的需求或簡單的程式，他們經常會在開發過程中應用各種設計模式，希望程式更加靈活，為未來的擴展打好基礎，實則過度設計，因為未來的需求並不一定會實作，這樣做徒增程式的複雜度。因此，我們有必要講一下如何避免過度設計，特別是如何避免濫用設計模式（物件導向程式設計範例、設計原則、程式規範和重構技巧等不容易被過度使用）。

1.4.1 程式設計的初衷是提高程式品質

談到創業，我們經常聽到一個詞：初心。「初心」的意思是我們到底為什麼做這件事。無論產品經過多少次迭代、轉變多少次方向，「初心」一般不會改變。當我們在為產品該不該轉型、該不該實作某個功能猶豫不決時，想想我們創業時的初心，自然就有答案了。

實際上，應用設計模式時也是如此。設計模式只是方法，應用它的最終目的（也就是初心）是提高程式的品質，也就是提高程式的可讀性、可擴展性和可維護性等。所有的程式設計都是圍繞這個初心來進行的。

因此，在進行程式設計時，我們一定要先思考一下為什麼要這樣設計，為什麼要應用這種設計模式，以及這樣做是否能夠真正提高程式品質，能夠提高程式哪些方面的品質。如果自己很難想清楚這些問題，或者提供的理由比較牽強，那麼基本上可以斷定這是一種過度設計，是「為了設計而設計」。

1.4.2 程式設計的原則是「先有問題，後有方案」

如果我們把程式看作產品，那麼，在做產品時，我們就要先思考產品的「痛點」在哪裡，用戶的真正需求是什麼，然後開發滿足需求的功能，而不是先實作一個「花俏」的功能，再東拼西湊出一個需求。

程式設計與此類似。我們先分析程式存在的「痛點」，如可讀性不高、可擴展性不高等，再有針對性地利用設計模式、設計原則對程式進行改善，而不是見到某個情境之後，就盲目地認為與之前看到的某個設計模式、設計原則的應用情境相似，隨意套用，不考慮是否合適。如果有人問起，就找幾個假需求隨便應付，如提高了程式的擴展性、滿足開閉原則等，這樣是不可取的。

實際上，很多沒有太多開發經驗的新手，往往在學完設計模式之後會非常「學生做派」，不懂得具體問題具體分析，手裡拿著錘子，看哪個都是釘子，不分青紅皂白，套用各種設計模式。寫完之後，看著自己寫的很複雜的程式，還沾沾自喜，這樣的做法很不可取。希望本節內容能夠給讀者帶來一些啟發。

1.4.3 程式設計的應用情境是複雜程式

一些設計模式書會給一些簡單的例子，但這些例子僅僅是為了能在有限的篇幅內向讀者講清楚設計模式的原理和實作，並沒有實戰意義。而有些讀者會誤以為這些簡單的例子就是這些設計模式的典型應用情境，常常依樣畫葫蘆，盲目地應用到自己的專案中，用複雜的設計模式去解決簡單的問題。在我看來，這是很多初學者在學完設計模式之後，在專案中進行過度設計的首要原因。

應用設計模式的目的是解耦，也就是利用更好的程式結構，將一大段程式拆分成職責單一的「小」類別，讓程式滿足「高內聚，低耦合」等特性。建立型設計模式是將建立程式和使用程式解耦，結構型設計模式是將不同的功能程式解耦，行為型設計模式是將不同的行為程式解耦。而解耦的主要目的是應對程式的複雜性問題。也就是說，設計模式是為了解決複雜程式問題而產生的。如果我們開發的程式不複雜，那麼就沒有必要引入複雜的設計模式。這與資料結構和演算法應對的是大規模資料的問題類似。如果資料規模很小，那麼再高效能的資料結構和演算法也發揮不了太大作用。例如，對幾十個字元長度的字串進行比對，使用簡單的樸素字串比對演算法即可，沒有必要使用具備更高性能的 KMP 演算法，因為 KMP 演算法儘管在性能上比樸素字串比對演算法高一個量級，但演算法本身的複雜度也高很多。

對於複雜程式，如專案的程式量大、開發週期長、參與開發的人員多，我們在前期要多花點時間在設計上。程式越複雜，我們花在設計上的時間就越多。不僅如此，對於每次提交的程式，我們都要確保品質，並經過足夠的思考和精心設計，這樣才能避免出現「爛程式效應」（每次提交的程式的品質都不高，累積起來，整個專案最終的程式品質就會很差）。如果我們參與的只是一個簡單的專案，程式量不大，開發人員也不多，那麼，簡單的問題用簡單的解決方案處理即可，不必引入複雜的設計模式，不要將簡單問題複雜化。

1.4.4 持續重構可有效避免過度設計

我們知道，應用設計模式可以提高程式的可擴展性，但同時會降低程式的可讀性。一旦我們引入某個複雜的設計，之後即便在很長一段時間都沒有擴展的需求，也不可能將這個複雜的設計刪除，整個團隊要一直背負著這個複雜的設計前行。

為了避免錯誤的需求預判導致的過度設計，我推薦持續重構的開發方法。持續重構不僅是保證程式品質的重要手段，也是避免過度設計的有效方法。在真正有痛點時，我們再考慮用設計模式來解決，而不是一開始就為不一定實作的未來需求而應用設計模式。

當對是否應用某種設計模式模稜兩可時，我們可以思考一下，如果暫時不用這種設計模式，隨著程式的演進，當某一天不得不去使用它時，需要改動的程式是否很多。如果不是，那麼能不用就不用，遵守 KISS 原則。對於 10 萬行以內的程式，如果團隊成員穩定，對程式涉及的商業熟悉，那麼，即便將所有的程式重寫，也不會花費太多時間，因此，不必為程式的擴展性過度擔憂。

1.4.5 不要脫離具體的情境談程式設計

程式設計是一件主觀的事。毫不誇張地講，程式設計可以稱為一種「藝術」。因此，程式設計的好壞很難評判。如果真的要進行評判，那麼儘量將其放到具體的情境中。我認為，脫離具體的情境探討程式設計是否合理是空談。這就像我們經常說的，脫離商業談架構是不切實際的。

例如，一個手機遊戲專案是否能被市場接受，往往非常不確定。很多手機遊戲開發出來之後，市場回饋很差，馬上就被放棄了。另外，儘快推出並佔領市場是手機遊戲致勝的關鍵。所以，對於一些手機遊戲專案的開發，前期往往不會在程式設計、程式品質上花費太多時間。但是，如果我們開發的是 MMORPG（大型多人線上角色

扮演遊戲）類的大型使用者端遊戲，那麼資金和人力投資相當大，專案推倒重來的成本很大。這個時候，程式的品質就很重要了。因此，在專案前期，我們就要多花點時間在程式設計上，否則，程式品質太差，bug 太多，後期將無法維護，也會導致很多用戶放棄而選擇同類型的其他遊戲。

又如，如果我們開發的是偏底層的、框架類的、通用的程式，程式品質就比較重要，因為一旦出現問題或程式需要改動，影響面會比較大。如果我們開發的是商業系統或不需要長期維護的專案，那麼放低對程式品質的要求是可以接受的，因為自己開發的專案的程式與其他專案沒有太多耦合，即便出現問題，影響也不大。

在學習程式設計時，我們要重視分析問題能力和解決問題能力的鍛鍊。在看到某段程式時，我們要能夠分析程式的優秀之處和不足之處並說明原因，還需要知道如何改善程式。反之，如果我們只是掌握了理論知識，即便把 22 種設計模式的原理和程式實作背得滾瓜爛熟，若不具備具體問題具體分析的能力，那麼，在面對多種多樣的真實專案的程式時，也很容易濫用設計模式而過度設計。

1.4.6 思考題

如何避免過度設計？關於這個話題，讀者有哪些心得體會和經驗教訓？