這些守則是怎麼來的？

本書的這些程式設計守則，其實是來自一堆令人抓狂的經驗。

我曾在微軟待了大約十年左右，在微軟負責管理程式設計團隊，然後在 1997 年和朋友一起創立了 Sucker Punch 這家遊戲公司。這兩家公司都算是相當成功——因為大體上來說，這兩家公司都能招募到相當優秀的人才，也能培養出一流的程式設計團隊。對於 Sucker Punch 公司來說，這一點直接導致我們公司 25 年以來，獲得了相當成功的遊戲開發成果。我們開發了三款《Sly Cooper》系列遊戲，讓各年齡層的孩子都能體驗到浣熊大盜 Sly Cooper 與他的小伙伴們驚險刺激的生活。我們也開發了五款《inFamous》系列遊戲，賦予遊戲玩家各種超能力，讓玩家可以自由選擇如何運用超能力來行善或作惡。後來我們還推出了《對馬戰鬼》（*Ghost of Tsushima*）這個代表作，玩家在遊戲中扮演一個孤獨的武士，獨自一人反抗 1274 年入侵日本的大軍 [1]。

微軟與 Sucker Punch 公司的招聘策略，大多是僱用年輕、聰明的程式設計者，然後再把他們訓練成專業的開發者。不可否認，這種做法相當成功，但同時也給我們帶來某些很特殊的挫折感。

我發現自己總是反覆遇到一些同樣的問題。我們幫團隊找來的新程式設計者，通常都是剛從大學畢業的新人。我在審視他們的程式碼時，經常發現他們會引進一些新功能，來解決某個非常簡單的問題——我發現他們都想要解決更大的問題，至於原本想解決的那個非常簡單、非常具體的問題，往往只不過是他們的解法其中一個小小的案例而已。

哎呀呀！其實很多時候我們並不需要去解決什麼大問題；就算想解決，也不是現在呀！那種能夠解決大問題的做法，對於我們所遇到的小問題來說，往往都是比較平庸的解法——不但用起來比較複雜，理解起來也比較複雜，而

1 眼尖的人或許已經注意到，我遺漏了 Sucker Punch 公司的第一款遊戲《Rocket: Robot on Wheels》。這其實是因為，你們大概很少有人玩過這個遊戲吧；如果你是少數玩過此遊戲的人，請受我一拜。

且還有可能把更多的 bug 隱藏起來。可是，在程式碼審查階段[2]（此時我們並不需要去解決大問題，只要去解決自己所能理解的問題就行了）貿然去嘗試解決大問題，實在是很沒效率的做法。即使如此，大家還是死性不改，怎麼講都講不聽。

因為我實在太沮喪，所以我的態度開始強硬了起來。「好吧，」我說：「新守則如下。除非你能針對同樣的問題舉出三個例子，否則絕不能採用通用化的解法。」

讓我感到很驚訝也很欣慰的是，這竟然是個很有效的做法！把「一般通用的理念」轉化成「具有實際標準的守則」，真的是成功傳達某種理念的有效做法。當然，我們大多數的新程式設計者，都曾犯過「太早去考慮通用性」的錯誤，而這個守則確實可以幫助他們，避免一再犯下這樣的錯誤。而且，這還可以幫助他們認識到，什麼時候才應該「考慮採用通用化的做法」。能舉出的例子還不到三個？那就別去考慮什麼通用性了吧。超過三個以上呢？這樣就可以開始考慮有沒有通用化的機會了。

這條守則之所以奏效，是因為它很好記，而且守則可適用的情況也很容易分辨。如果程式設計者碰到某個明確的問題，就想開始考慮某種通用的做法，這時候他可以先退一步，先算一下這問題能找出幾個具體的例子，然後他往往就能做出更好的判斷，決定要不要採取通用的做法。這樣一來，他們往往就能寫出更好的程式碼。

隨著時間的推移，我們發現 Sucker Punch 公司這一整套理念，可以提煉成幾句很好記的小短句——準確來說，應該叫做格言才對。格言這東西很早之前就有了——格言總是可以透過一些簡短、精闢的陳述，擷取出一些很基本的真理。我敢打賭，你隨隨便便就能想出幾句來。就算我給你一些限制，只能說**鳥類相關**的格言，你還是可以隨便想出至少兩句以上！我就先來說幾句好了：

- 小雞孵出來之前，先別急著算有幾隻[3]。
- 一鳥在手，勝過二鳥在林。

---

2 Sucker Punch 公司裡每個專案所提交的程式碼，全都必須通過程式碼審查。詳見守則 6。

3 其原始形式取自 Thomas Howell 的《New Sonnets and Pretty Pamphlets》（新十四行詩和精美小冊，1570）：「不要去數你還沒孵出來的小雞……」如果你想知道一句好格言能持續存在多久，這就是個活生生的例子。

- 早起的鳥兒有蟲吃。
- 別把雞蛋放在同一個籃子裡。

像這樣的格言之所以廣為流傳，主要是因為它確實很管用。如果用現代的說法，這就像是「病毒式傳播」——幾千年來，格言一直用它所蘊含的智慧來「感染」每個人[4]。Sucker Punch 公司之所以選擇這樣的方式，把程式設計理念傳染給團隊的新成員，這也沒什麼好奇怪的。

所以，我們就這樣一點一滴從一條一條的守則，演化出一整套守則——也就是本書所要談的「程式設計守則」。這些守則代表了 Sucker Punch 公司工程文化的許多重要面向：我們認為它造就了我們的成功，而且也是我們團隊裡每個新成員必須好好學習，才能變得更有效率的一些想法。即使像我這樣的程式設計老手，偶爾也需要自我提醒一下！

接下來每一章都會說明一條守則，並提供了大量的範例，來說明其背後的想法。每讀完一章，你應該就會更清楚瞭解，該守則所鼓勵的程式設計實務做法，以及可適用的情境。

這些守則用書本的形式來呈現，能不能同樣具有傳染性呢？且讓我們拭目以待吧。

4 「**格言**」（*aphorism*）這個詞是 Hippocrates 在西元前 400 年左右創造出來的。好吧，嚴格來說，他所創造的其實是 Ἀφορισμός。這是他一本關於醫學診斷和治療守則的書名，其中有一些守則在經過幾千年之後，直到現在還是很有道理，例如第 6 節的第 13 句格言——「打噴嚏可以治癒打嗝」。這實在說得太對了。

越簡單越好、但也不能太過於簡單

程式設計真的很困難。

我猜你早就知道了。你會把這本《程式設計守則》拿起來讀，就說明你可能：

- 具有程式設計的能力（至少算是略懂吧）
- 心情還蠻沮喪，因為程式設計真的沒有想像中簡單

程式設計之所以困難，有很多原因，不過我們也有很多不同的策略，可以讓它變容易一點。本書會精心挑選出一些「把事情搞砸」的常見做法，然後再審視一些能避開這些錯誤的守則。這全都是我自己多年來犯下許多錯誤，以及應對他人錯誤的一些經驗累積。

這些守則整體上來說都具有特定的模式，大多數守則背後都有個共同的主題。當年愛因斯坦為了說明理論物理學家的目標，曾經說過一句話，這句話正好可用來作為最好的總結：「越簡單越好、但也不能太過於簡單[1]。」愛因斯坦的意思是，最好的物理理論，就是有能力完整描述所有可觀察到的現象，其中最簡單的那個理論。

如果把這樣的想法套在程式設計領域，那就是——任何問題的最佳解法，就是能滿足問題所有的要求，其中最簡單的那個做法。最好的程式碼，就是最簡單的那段程式碼。

---

1 幾乎可以肯定的是，愛因斯坦本人並沒有確切說過那樣的話——應該是後人把他說過的話稍作改編，才變成這樣的一句格言。在正式的書面記錄裡最接近的一句話是：「幾乎難以否認的是，所有理論最高的目標，就是讓一些無法再進一步簡化的基本元素，盡可能越少越簡單越好，同時在面對經驗上任何單一事物時，也不必為了找出適當的解釋而做出妥協。」所以囉，雖然在概念上幾乎是一樣的東西，但原本的說法確實沒那麼討喜啦。此外，愛因斯坦原始的說法，要拿來作為一條守則，感覺也過於冗長了。

想像一下，假設你正在寫程式，想要計算出任何整數在化為二進位之後，其中有幾個位元的值為 1。想要計算出結果，有很多種不同的做法。你或許可以運用一些位元操作技巧[2]，把每一個非零位元逐一歸零，再計算出有幾個位元被歸零：

```
int countSetBits(int value)
{
    int count = 0;

    while (value)
    {
        ++count;
        value = value & (value - 1);
    }

    return count;
}
```

你也可以選擇不採用迴圈的做法，直接利用位元平移和遮罩的操作，以平行的方式計算出非零位元的數量：

```
int countSetBits(int value)
{
    value = ((value & 0xaaaaaaaa) >> 1) + (value & 0x55555555);
    value = ((value & 0xcccccccc) >> 2) + (value & 0x33333333);
    value = ((value & 0xf0f0f0f0) >> 4) + (value & 0x0f0f0f0f);
    value = ((value & 0xff00ff00) >> 8) + (value & 0x00ff00ff);
    value = ((value & 0xffff0000) >> 16) + (value & 0x0000ffff);

    return value;
}
```

或者你也可以採用下面這種最簡單明瞭的寫法：

```
int countSetBits(int value)
{
    int count = 0;

    for (int bit = 0; bit < 32; ++bit)
    {
        if (value & (1 << bit))
            ++count;
    }
```

2 在這裡要先向所有非 C++ 程式設計者道個歉，因為接下來的三個範例都會用到一些位元相關的操作。我向各位保證，本書其餘部分就不太會用到位元相關操作了。

```
        return count;
    }
```

前兩種解法都很巧妙……但我這樣說並不是稱讚的意思[3]。如果只是很快看一眼，你應該也搞不太清楚前兩種解法實際的工作原理吧——這兩種解法的程式碼，其實都隱藏了「等一下……這是什麼意思呀？」這樣的東西。只要稍微想一下，你或許就能搞懂怎麼回事，並且看出其中所採用的技巧其實還蠻有趣的。不過，你終究還是要多花點力氣，才能搞清楚怎麼回事。

況且在這個例子裡，你其實已經先佔有先天上的優勢了！我一開始就已經先告訴你這個函式的作用，然後才給你看程式碼，而且函式的名稱也可以明顯看出它的功用。如果你「**並不知道**」這些程式碼是用來計算位元值為 1 的數量，那你肯定要再多花點功夫，才能搞懂前兩段程式碼的功用。

至於最後一種解法，情況就不同了。它很明顯就是在計算位元值為 1 的數量。它所採用的做法再簡單也不過了，可是又不會太過於簡單，因此，它確實是比前兩種更好的解法[4]。

簡不簡單怎麼衡量呢？

有很多方法，可以讓程式碼變得更簡單。

你或許認為，可以根據團隊中其他人想要理解程式碼的難易程度，來衡量程式碼的簡單程度。如果隨機挑選的同事可以毫不費力讀懂你的程式碼，那麼你的程式碼就可以說是相當簡單。

或者你可能認為，可以用程式碼建立過程的難易程度，來衡量程式碼的簡單程度——你不但可以把輸入程式碼的時間考慮進來，還可以把程式碼功能完整而且沒有錯誤所需要花費的時間列入考量[5]。複雜的程式碼總需要很長的時間才能完全寫正確；簡單的程式碼則比較容易跨越那條名為「正確」的終點線。

---

3 在另一個平行宇宙裡，這叫做「聰明並不是一種美德」。

4 現代處理器有一條專用的指令，可用來計算出任何值其中非零位元的數量——例如在 x86 處理器中就是 `popcnt`，它只需要一個執行週期就能計算出結果。你也可以用 SIMD 指令來做這類的計算，速度甚至比 `popcnt` 還要快。但所有這些做法都很難理解，實際上可支援哪些指令，則取決於你所採用的是哪種處理器。除非有非常非常好的理由，我才會考慮去使用比較複雜的東西，否則我寧願使用最簡單的 `countSetBits`。

5 當然囉，這裡所說的「沒有錯誤」，是指在實驗的錯誤範圍內沒有錯誤。實際上，永遠都還是會有一些你尚未發現的 bug。

當然囉，這兩種衡量方式有很多重疊之處。很容易就能寫出來的程式碼，讀起來往往也比較容易懂。你當然也可以採用其他一些有效的衡量方式，來衡量程式碼的複雜程度：

寫了多少程式碼

比較簡單程式碼，往往都比較簡短一些，不過也有可能是因為把很多複雜性，全都塞進一行程式碼中了。

引入了多少構想

比較簡單的程式碼，總是比較傾向於採用團隊裡每個人都知道的概念作為基礎；它不會輕易引入一些需要重新思考問題的新方法或新術語。

解釋起來需要花多少時間

比較簡單的程式碼，往往比較容易解釋——在程式碼審查過程中，這一點就很明顯，因為這樣審查者才能迅速理解程式碼的意圖。比較複雜的程式碼，往往就需要許多額外的解釋與說明。

如果程式碼用某一種衡量方式來看很簡單，用另一種衡量方式來看應該也會很簡單才對。你只需要選出一些可以讓你最清楚聚焦的衡量方式就行了——我建議可以先考慮採用「比較容易建立」、「比較容易理解」的衡量方式。如果你能寫出很容易看懂的程式碼，而且很快就能派上用場，那你所建立的應該就是簡單的程式碼了。

……但也不能太過於簡單

程式碼簡單一點確實很不錯，但它終究還是要能把問題解決才行。

想像一下，假設你每跨出一步，就會踏上一階、兩階或三階的階梯，然後你想計算出有多少種方法，可以爬完某階梯數量的梯子。如果梯子有兩階，就有兩種攀爬的方式——其中一種會踏上第一階，另一種則不會踏上第一階。類似的情況，如果要爬三階的梯子，就會有四種方式——只踏第一階、只踏第二階、第一、二階都踏，或是第一、二階都不踏，直接踏上第三階。四階的梯子就有七種方式，五階有十三種方式，其他則可依此類推。

你可以寫個簡單的程式碼，來進行遞迴運算：

```
int countStepPatterns(int stepCount)
{
    if (stepCount < 0)
        return 0;

    if (stepCount == 0)
        return 1;

    return countStepPatterns(stepCount - 3) +
           countStepPatterns(stepCount - 2) +
           countStepPatterns(stepCount - 1);
}
```

這段程式碼基本的想法是，如果要爬上梯子的最高一階，前一步肯定是在前三階的其中一階。只要把爬到前三階的方法數量全部加起來，就可以計算出爬到最高階的方法有多少數量了。有了這樣的理解之後，只要再把所謂的「基礎狀況」（base cases）搞清楚就行了。前面那段程式碼裡的基礎狀況，可以接受負值的步數，這樣一來就可以讓遞迴運算變得更簡單了。

遺憾的是，這個解法有點限制。沒錯，它確實是有效的做法，至少在 `stepCount` 值比較小的情況下沒什麼問題，不過 `countStepPatterns(20)` 計算完成的時間，大約會是 `countStepPatterns(19)` 的兩倍。電腦的計算速度確實很快，但像這種指數型成長的情況，很快就會讓電腦的速度不夠用了。在我個人的測試中，這段程式碼裡的 `stepCount` 只要來到二十幾左右，就會開始變得非常緩慢。

如果你想計算出爬梯子的方法數量，這段程式碼就太過於簡單了。最核心的問題在於，計算過程中 `countStepPatterns` 會被一次又一次重新計算，這也就直接導致執行時間呈現出指數型成長的結果。解決此問題其中的一個標準做法，就是用記憶體把計算過的值記起來，隨後又用到時再直接拿來用，做法如下：

```
int countStepPatterns(unordered_map<int, int> * memo, int rungCount)
{
    if (rungCount < 0)
        return 0;

    if (rungCount == 0)
        return 1;

    auto iter = memo->find(rungCount);
```

```
    if (iter != memo->end())
        return iter->second;

    int stepPatternCount = countStepPatterns(memo, rungCount - 3) +
                           countStepPatterns(memo, rungCount - 2) +
                           countStepPatterns(memo, rungCount - 1);

    memo->insert({ rungCount, stepPatternCount });
    return stepPatternCount;
}

int countStepPatterns(int rungCount)
{
    unordered_map<int, int> memo;
    return countStepPatterns(&memo, rungCount);
}
```

像這樣運用了記憶體的做法之後，每個值只要被計算過一次，就會被保存到一個雜湊對應表（hash map）中。後續在調用到相同的計算值時，就可以直接到雜湊對應表中找出計算值；由於這大概只需要固定長的時間，所以執行時間呈現指數型成長的情況就消失了。這個運用記憶體的做法，程式碼會稍微複雜一些，但這樣就不會遭遇到執行效能上的瓶頸了。

或許你也可以採用動態程式設計（dynamic programming）的做法，在概念複雜性方面做出一點犧牲，就可以讓程式碼本身變得更簡單一點：

```
int countStepPatterns(int rungCount)
{
    vector<int> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }

    return stepPatternCounts.back();
}
```

這種做法執行起來也夠快，而且比起運用記憶體的遞迴版本來說，又更簡單了。

有時候簡化問題比簡化解法更重要

原本最前面那個遞迴版本的 countStepPatterns，只要遇到階梯數量比較大的情況就會出問題。那段最簡單的程式碼，在階梯數量比較小的情況下非常有效，但是階梯數量變大之後，執行時間就會遇上指數型成長的瓶頸。隨後的版本雖然稍微複雜了一些，但以此作為代價，就可以避免指數型成長的問題……不過，這樣的做法很快就遇到了另一個不同的問題。

如果執行前面那段程式碼來計算 countStepPatterns(36)，可以得出正確的答案 2,082,876,103。但如果執行 countStepPatterns(37) 則會得到 –463,960,867 的結果。這顯然是不正確的！

這其實是因為我所使用的 C++ 版本，會把整數存為帶有正負號的 32 位元值，在計算 countStepPatterns(37) 時發生了位元溢出（overflow）的問題。爬上 37 階的方法有 3,831,006,429 種，這個數字實在太大了，根本無法放入一個帶有正負號的 32 位元整數之中。

所以，這段程式碼或許還是太過於簡單了。我們希望 countStepPatterns 這個函式可適用於任何數量的階梯，這好像也是合理的期待，對吧？ C++ 在面對超大的整數時，並沒有標準的解法，不過還是有（許多）開源的函式庫，可以實作出任意精度的各類整數。或者你也可以靠自己寫個幾十行程式碼，就能實作出來了：

```
struct Ordinal
{
public:

    Ordinal() :
        m_words()
        { ; }
    Ordinal(unsigned int value) :
        m_words({ value })
        { ; }

    typedef unsigned int Word;

    Ordinal operator + (const Ordinal & value) const
    {
        int wordCount = max(m_words.size(), value.m_words.size());

        Ordinal result;
        long long carry = 0;
```

```
        for (int wordIndex = 0; wordIndex < wordCount; ++wordIndex)
        {
            long long sum = carry +
                            getWord(wordIndex) +
                            value.getWord(wordIndex);

            result.m_words.push_back(Word(sum));
            carry = sum >> 32;
        }

        if (carry > 0)
            result.m_words.push_back(Word(carry));

        return result;
    }

protected:

    Word getWord(int wordIndex) const
    {
        return (wordIndex < m_words.size()) ? m_words[wordIndex] : 0;
    }

    vector<Word> m_words;
};
```

接著我們只要用 `Ordinal` 替換掉之前範例程式碼裡的 `int`，就能針對更多的階梯數量計算出精確的答案：

```
Ordinal countStepPatterns(int rungCount)
{
    vector<Ordinal> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }

    return stepPatternCounts.back();
}
```

所以……問題解決了嗎？引入了 Ordinal 之後，就能針對更多的階梯數量，計算出準確的答案。當然，增加幾十行程式碼來實作出 Ordinal 並不是什麼特別棒的做法，尤其是考慮到實際上原本的 countStepPatterns 函式只有 14 行而已，難道這就是正確解決問題一定要付出的代價嗎？

這倒也不見得。如果問題本身並沒有簡單的解法，在接受比較複雜的解法之前，你還是可以先仔細審視一下所要解決的問題。你想要解決的問題，實際上真的是一定要解決的問題嗎？有沒有可能是你對問題做了沒必要的假設，所以你的解法也跟著複雜化了？

在這裡的例子中，如果你實際上要計算的對象是現實世界裡真實的階梯，或許你就可以假設階梯的數量有個最大值。如果階梯數量的最大值是 15，那麼本節裡的每個解法全都完全夠用了，就算是最簡單樸實的第一個遞迴解法，也不會有什麼問題。只要在程式碼裡添加一個 assert，提醒一下這個函式有個預設的限制，這樣也就行了：

```
int countStepPatterns(int rungCount)
{
    // 注意！ (chris) 如果階梯數量超過 36 以上，
    // 計算出來的結果就會超出 int 的限制 ...

    assert(rungCount <= 36);

    vector<int> stepPatternCounts = { 0, 0, 1 };

    for (int rungIndex = 0; rungIndex < rungCount; ++rungIndex)
    {
        stepPatternCounts.push_back(
            stepPatternCounts[rungIndex + 0] +
            stepPatternCounts[rungIndex + 1] +
            stepPatternCounts[rungIndex + 2]);
    }

    return stepPatternCounts.back();
}
```

如果真的需要支援很大的階梯數量——比如處理風力發電機的檢查專用梯——只要能算出大概的估計值，這樣是否就足夠了呢？如果真是如此的話，只要改用浮點數去替換掉原本的整數型別，這也是非常容易做到的解法。這個解法實在太簡單了，我甚至連程式碼都懶得秀出來了。

你想想看，實際上不管是什麼東西，到最後終究會遇到東西滿出來放不下的溢出（overflow）問題。為了解決這種極端的臨界狀況，結果總是會讓我們引入太過於複雜的解法。請不要為了解決問題中最嚴格的情況，而掉入這樣的陷阱之中。你應該針對實際需要解決的部分，提供一個簡單的解法，而不要為了更廣泛解決所有的情況，反而提出更複雜的解法；務實而簡單的解法肯定是比較好的。如果無法簡化你的解法，或許你可以先嘗試簡化你要解決的問題。

簡單的演算法

有時候，如果選擇了不恰當的演算法，就會增加程式碼的複雜性。任何特定的問題，都有很多種解法，而其中確實有些解法會比其他解法更複雜。比較簡單的演算法，往往可以讓我們寫出比較簡單的程式碼。問題是，簡單的演算法並不總是那麼顯而易見！

比方說，你正在寫一個程式，想針對一副紙牌進行洗牌的動作。其中一種很明顯的做法，就是模擬你小時候可能學過的洗牌方式——先把一副牌分成兩堆，然後讓兩邊交錯混合，使得兩邊的牌都有大致相等的機會重組成新的順序。只要重複這個動作，直到整副牌的順序被徹底打亂為止[6]。

相應的程式碼或許就像下面這樣：

```
vector<Card> shuffleOnce(const vector<Card> & cards)
{
    vector<Card> shuffledCards;

    int splitIndex = cards.size() / 2;
    int leftIndex = 0;
    int rightIndex = splitIndex;

    while (true)
    {
        if (leftIndex >= splitIndex)
        {
            for (; rightIndex < cards.size(); ++rightIndex)
                shuffledCards.push_back(cards[rightIndex]);
```

6 在洗了七次牌之後，紙牌的順序就會變得非常隨機了。如果只洗四、五次牌，整副牌根本還達不到隨機的混亂程度。沒錯，我總是堅持多洗幾次牌才肯發牌，連我的家人都有點受不了：「Chris，我們是來打牌的，不是來看你洗牌的。」這時候我只能說，一知半解的知識，真的是很危險的東西呀。

```
            break;
        }
        else if (rightIndex >= cards.size())
        {
            for (; leftIndex < splitIndex; ++leftIndex)
                shuffledCards.push_back(cards[leftIndex]);

            break;
        }
        else if (rand() & 1)
        {
            shuffledCards.push_back(cards[rightIndex]);
            ++rightIndex;
        }
        else
        {
            shuffledCards.push_back(cards[leftIndex]);
            ++leftIndex;
        }
    }

    return shuffledCards;
}

vector<Card> shuffle(const vector<Card> & cards)
{
    vector<Card> shuffledCards = cards;

    for (int i = 0; i < 7; ++i)
    {
        shuffledCards = shuffleOnce(shuffledCards);
    }

    return shuffledCards;
}
```

這種模擬人工洗牌的演算法確實是有效的，而我在這裡所寫出來的程式碼，就是此演算法一個相當簡單的實作結果。你在程式碼裡必須花一點力氣來確保所有索引值全都正確無誤，不過這並不算是太麻煩的事。

然而，洗牌這件事其實還有更簡單的演算法。舉例來說，你也可以用一次處理一張的方式來進行洗牌。在每次的迭代過程中，你都可以直接取一張新牌，然後再與整副牌裡隨機取的某張牌進行交換。實際上，你可以直接就地（in place）執行此操作：

```
vector<Card> shuffle(const vector<Card> & cards)
{
    vector<Card> shuffledCards = cards;

    for (int cardIndex = shuffledCards.size(); --cardIndex >= 0; )
    {
        int swapIndex = rand() % (cardIndex + 1);
        swap(shuffledCards[swapIndex], shuffledCards[cardIndex]);
    }

    return shuffledCards;
}
```

如果用前面介紹過的方式來衡量程式碼的簡單程度，後面這個版本顯然更勝一籌。寫出這段程式碼的時間更短了[7]。程式碼也更容易看得懂。程式碼的行數比較少。解釋起來也更容易。它顯然更簡單、更好——這一切並不是因為程式碼本身，而是因為我們選擇了更好、更簡單的演算法。

別破壞故事的流暢性

簡單的程式碼總是比較容易閱讀——尤其是那種最簡單的程式碼，根本就可以直接從上面一路往下讀，就像在讀一本書一樣。不過，程式畢竟不是書。如果程式的整個流程本身就沒有那麼簡單，寫出來的程式碼自然也會比較難以理解。如果程式碼本身令人費解，而你又經常要跟著執行的流程，從某個地方跳到另一個地方去，這樣閱讀起來就會困難許多。

有時候程式碼之所以讓人覺得費解，或許是因為程式設計師太過於想要在某處把每個想法表達清楚，最後反而把大家搞昏頭了。我們就用之前洗牌的那段程式碼為例好了。處理左右兩堆牌的程式碼，看起來非常相似。我們可以把其中「移動一張牌」和「移動一堆牌」的邏輯拆分出來，變成兩個單獨的函式，然後再讓 `shuffleOnce` 分別去調用：

```
void copyCard(
    vector<Card> * destinationCards,
    const vector<Card> & sourceCards,
    int * sourceIndex)
{
    destinationCards->push_back(sourceCards[*sourceIndex]);
```

7 根據我實驗性的一番測量來看，實際上所花的時間有可能因人而異。在寫洗牌的程式碼時，由於我對索引和條件的寫法不太熟悉，嘗試了好幾次才把程式碼寫好。至於隨機選牌的洗牌程式碼，則是第一次嘗試就成功了。

```
    ++(*sourceIndex);
}

void copyCards(
    vector<Card> * destinationCards,
    const vector<Card> & sourceCards,
    int * sourceIndex,
    int endIndex)
{
    while (*sourceIndex < endIndex)
    {
        copyCard(destinationCards, sourceCards, sourceIndex);
    }
}

vector<Card> shuffleOnce(const vector<Card> & cards)
{
    vector<Card> shuffledCards;

    int splitIndex = cards.size() / 2;
    int leftIndex = 0;
    int rightIndex = splitIndex;

    while (true)
    {
        if (leftIndex >= splitIndex)
        {
            copyCards(&shuffledCards, cards, &rightIndex, cards.size());
            break;
        }
        else if (rightIndex >= cards.size())
        {
            copyCards(&shuffledCards, cards, &leftIndex, splitIndex);
            break;
        }
        else if (rand() & 1)
        {
            copyCard(&shuffledCards, cards, &rightIndex);
        }
        else
        {
            copyCard(&shuffledCards, cards, &leftIndex);
        }
    }

    return shuffledCards;
}
```

shuffleOnce 之前的版本比較容易從上面一路往下讀；但這個版本就比較沒辦法了。這樣一來程式碼就會變得比較難閱讀。在閱讀 shuffleOnce 的程式碼過程中，你必須隨著執行流程進入到 copyCard 或 copyCards 的函式裡。然後你必須跟隨這些函式的邏輯，想辦法搞清楚它們的作用，接著再回到原本的函式，然後還要把你從 shuffleOnce 送進去的參數，與你剛才好不容易才搞懂的 copyCard 或 copyCards 對應起來。這比起直接讀原始版本的 shuffleOnce 困難多了。

因此，雖然你為了不想做重複工作而寫了那兩個函式，但它花了你更多的時間來寫程式[8]，結果竟然還讓你的程式碼變得更難閱讀。而且，你寫的程式碼也變多了！你本來只是為了刪掉重複的程式碼，結果卻讓程式碼變得更複雜，而不是變得更簡單。

很顯然的，減少重複程式碼這件事，並沒有那麼單純！很重要的是，我們應該要知道，刪除重複程式碼其實是要付出一些代價的——如果只是少量的程式碼或單純的想法，最好還是保留原本重複的版本就行了。這樣一來，程式碼會比較好寫，而且也會比較容易閱讀。

所有守則都要遵守的守則

本書其餘的許多守則，其實都要回頭呼應這個「簡單」的主題，也就是要設法讓程式碼越簡單越好，但也不能太過於簡單。

從本質上來說，程式設計本身就是一場與複雜性之間的奮戰。添加新功能通常也就表示程式碼會變得更複雜——程式碼越來越複雜，就會變得越來越難處理，進展也會變得越來越緩慢。最後你就會來到一個極限，任何往前推進的嘗試（例如修正一個 bug 或添加一個功能）都會導致一堆問題，而所造成的問題就與所要解決的問題一樣多。至此想要進一步獲得進展，實際上就變成不可能的事了。

「複雜性」這東西到最後一定會扼殺掉你的專案。

8 同樣的，這也是我實驗而來的看法。實際上，我在使用指針和引用參照時搞了好一陣子，試了好幾遍才讓程式碼順利完成編譯。

這也就表示，所謂有效的程式設計，其實就是想辦法讓那些無可避免的情況越晚發生越好。在添加功能、修正 bug 時，一定要盡可能少添加一些複雜性。你可以想辦法找出能夠消除掉複雜性的機會，或是建構出一些東西，讓新的功能不會增加太多的系統整體複雜性。團隊的合作方式，也應該盡可能加以簡化。

如果你在這些方面做得非常勤懇而努力，你就有機會把那些無可避免的情況往後無限期拖延，讓它越晚發生越好。我在 25 年前寫下了 Sucker Punch 公司的第一行程式碼，此後整個程式碼庫經歷了持續的發展。至少到現在還看不到盡頭——我們現在的程式碼，確實比 25 年前複雜許多，但我們一直把複雜性控制得很好，因此我們還是可以持續獲得很有效率的進展。

既然我們有辦法管理好複雜性，你當然也可以。請保持敏銳，別忘了複雜性就是你最終極的敵人，你一定可以做得很好的。

Bug 是會傳染的

程式設計有個真理，那就是越早發現 bug，修起來越容易。這個說法通常是正確的……不過我認為更正確的說法是，你的 bug 發現得越晚，修起來就越痛苦。

程式碼裡一旦有 bug，大家無意中就會寫出與這個 bug 有關聯的程式碼。有時候，跟 bug 脫不了關係又很不穩定的這些程式碼，與 bug 本身離得並不遠，程式碼和 bug 全都位於同一個系統之中。有時則離得比較遠——也許程式碼位於你的下游，每當它調用你這個有 bug 的系統時，你的 bug 所給出的錯誤結果，反而會成為所有下游程式碼必須依賴的結果；程式碼也有可能在你的上游——而上游的程式碼之所以還能正常運作，很可能是因為你的 bug 會讓你只用特定的方式去調用上游的程式碼。

這其實是很自然的事——也是不太可能完全避免的事。通常我們會注意到的都是出問題的部分，而不是做對的部分。只要一出現問題，我們就會進行調查以找出原因。但只要沒出問題，我們就不會去做任何調查。如果你的程式碼可以正常運作，或是看起來至少還可以正常運作，你很自然就會認為它是以你所想的方式在運作，但實際上它運作的方式，很可能與你想像的不同。由於你並沒有去做調查，所以你永遠不會發現，究竟是什麼樣錯綜複雜的環境，導致你的程式碼正好可以如你所願正常運作。

這樣的問題不但會出現在你自己所寫的程式碼中，如果有別人寫了一些程式碼來調用你的程式碼，同樣也會有一樣的問題。如果你在整個團隊的程式碼庫裡提交了某個 bug，整個程式碼庫就會以緩慢而無可避免的方式，逐漸積累出大量受你 bug 所影響的其他程式碼。而當你修正掉這個明顯的 bug 之後，你整個專案的其他部分反而會突然很神祕地無法正常運作，這時你才會察覺到這些檯面下糾纏不清的關係。

你越早找出 bug，這些糾纏關係就越難有機會發芽、茁壯。因為這樣一來需要清理的糾纏關係就會比較少一點——像這樣的清理工作，通常都是修正 bug 時最花時間的部分。處理 bug 所影響到的部分，往往比修復 bug 本身更花時間，這其實是很常見的情況。

把 bug 視為一種具有傳染性的東西，是很有用的一種概念。系統裡的每個 bug 都會製造出新的 bug，因為有一些新的程式碼可能需要依賴這個 bug 才能正常運作。如果想要阻止這樣的傳染效應，最好的方式就是在 bug 產生邪惡影響並逐漸傳播出去之前，儘早把 bug 處理掉。

不要指望你的使用者

好，所以我們要儘早檢測出問題。但是，我們該怎麼做呢？

第一個你不能指望的，就是你的使用者。無論是那些會去調用你程式碼的團隊伙伴，還是那些會去使用你程式中各種功能的使用者，總之不管是哪一種使用者，都不會是你很好的第一道防線。當然囉，有時他們會向你回報一些問題，但其實他們更多時候會在心裡假設，他們所看到的行為就是你要提供給他們的行為。這其實就是一切糾纏的來源——這些問題往往沒有人會注意到；當然囉，有些問題還是會被注意到，只是後來都被當成設計的一部分了。

關於這點，你也可以嘗試做出一點改善。你可以嘗試寫出一些更符合使用者取向的文件。你可以把你的整個團隊拉進會議室裡，好好解釋一下新的系統或功能。你可以維護一個內部用的最新 wiki 百科網頁，其中包含所有關於如何套用各種東西的詳細資訊，或者把技術相關說明放到你的某個網站上。所有這些做法都是值得的——雖然可能要花不少錢，而且效果各有不同，不過這些做法都還蠻有用的——只不過，這樣並不能真正解決問題。從根本上來說，你的使用者並不像你，能夠如此清楚理解你的意圖，因此無論你怎麼做，他們就是會把 bug 當成是功能的一部分。

還有另一個更好的做法，就是採用某種持續進行的自動化測試做法。大多數程式設計者都同意，自動化測試是一件好事。至少，程式設計者應該都認同，自動化測試對於「其他」程式設計者來說是一件好事，只不過他們自己並不一定有意願去做這件事。

拔草囉

我女兒還小的時候，我們家裡有一台任天堂的 GameCube。事實證明，有個以製作遊戲為生的父親，其中一個副作用就是，你家裡會有市面上的每一款電視遊樂器。我的孩子們一直到很久之後才意識到，並不是每個人家裡都是這樣的。

我們最喜歡的遊戲是《動物森友會》；在這個遊戲裡，我們三人共享一個充滿擬人化動物的小村莊。你可以在村子裡做各式各樣的事情——挖掘化石、設計新衣服、裝飾你的房子、蒐集貝殼、釣魚、與鎮上的動物交朋友，或是單純放鬆一下，聽聽 KK Slider 演奏他的吉他。

在《動物森友會》這個遊戲裡，你要做的其中一件事就是拔草。每天晚上，無論你那一天有沒有玩過遊戲，你的村子裡一定會長出一些雜草。拔掉這些雜草還蠻容易的——只要跑到雜草旁，按下按鈕，然後「咻！」一聲，雜草就沒了。不過，這件事你必須一直去做。不管你有沒有拔，雜草都會一直不斷長出來。甚至在你沒玩遊戲的日子裡，雜草還是會一直長出來！如果你不去拔掉那些雜草，雜草就會接管你的小村莊。

經過 20 年之後，這個遊戲的團隊依然持續在製作《動物森友會》這個遊戲。數以千萬計的人都玩過這個遊戲的某一代，每個人都有相同的經歷——你只不過離開了幾週，當你再次回到之前辛辛苦苦打掃乾淨的小村莊，就會發現它已經又雜草叢生了。雖然經過了二十年，我還是能感受到那種不暢快的感覺。

你的專案其實就像是那個村莊。你一定要拔掉那些雜草——也就是在你的程式碼庫裡不斷出現的那些小瑕疵。無論你是否正在做專案的工作，無論你有沒有在拔雜草，每天都還是會有更多的雜草一直長出來。如果你不拔掉那些雜草，它就會把這個專案堵得讓人難過得要命。

所以，在這個比喻裡的雜草，究竟是指什麼東西呢？它指的就是那些很容易修正、但也很容易被忽略的小問題。想一想《動物森友會》裡的雜草——要拔掉它，只不過就是按下按鈕這麼簡單而已。拔掉雜草也不會有什麼副作用。它並不會在其他任何地方引起什麼問題。所有的改變，就只是少了一些雜草而已。

下面就是一段有雜草的程式碼：

```
// @brief 移除掉向量裡重複的整數
//
// @param values 想要進行壓縮的整數向量

template <class T>
void compressVector(
    vector<T> & values,
    bool (* is_equal)(const T &, const T &))
{
    if (values.size() == 0)
        return;

    int iDst = 1;

    for (int iSrc = 1, c = values.size(); iSrc < c; ++iSrc) {
        // 檢查看看有沒有重復值
        if (!is_equal(values[iDst - 1], values[iSrc]))
        {
            values[iDst++] = values[iSrc];
        }
    }

    values.resize(iDst);
}
```

這段程式碼裡的註解說明，有幾個明顯的問題。首先，註解說明的內容與函式本身的功能並不相符。這感覺上好像是一開始的時候，它本來只是個用來針對整數向量裡的重複值進行壓縮的函式，可是後來有人把它改成樣板的寫法，卻忘記更新註解說明了。最上面的註解說明，實在太不精確了——我們並不是刪除掉所有的重複值，只是刪除掉**相鄰的**重複項目而已。除非事先對陣列進行過排序，否則這絕對不是同一回事。另外，第二個註解說明裡也有個錯別字。這些問題全都解決掉之後：

```
// @brief 針對向量裡一連串相同的值進行壓縮
//
// 如果向量裡出現一連串相同的值，就只保留第一個值，
// 其他重複的值全都移除掉。
```

```
//
// @param values 想要進行壓縮的向量
// @param is_equal 所要使用的比較函式

template <class T>
void compressVector(
    vector<T> & values,
    bool (* is_equal)(const T &, const T &))
{
    if (values.size() == 0)
        return;

    int iDst = 1;

    for (int iSrc = 1, c = values.size(); iSrc < c; ++iSrc) {
        // 檢查看看有沒有重複值
        if (!is_equal(values[iDst - 1], values[iSrc]))
        {
            values[iDst++] = values[iSrc];
        }
    }

    values.resize(iDst);
}
```

修正這些問題，其實就是在拔草。做起來很容易。修正註解說明並不會在其他地方引起任何問題。而且我確實讓程式碼變得更好了——把註解說明裡模棱兩可的說明修正掉，確實有可能會在某個時候，讓某個人避開犯錯的機會。

不過，這裡還有更多的事可以做。其實這裡還有一些命名和格式上的問題。`i` 和 `c` 這兩個變數並沒有遵循標準的約定慣例——按照這個專案的約定慣例，應該要使用 `index` 和 `count`，而不是單獨的一個字母。`is_equal` 這個參數也應該改成 `isEqual`，才能符合這個專案的函式命名風格。大括號的寫法也不一致，而且這個專案的約定慣例並不贊成把多個參數全都包在 `for` 語句之中。約定慣例也要求註解說明的後面應該要有空行，但第二個註解說明並沒有這麼做。

這些全都很容易進行修正：

```
// @brief 針對向量裡一連串相同的值進行壓縮
//
// 如果向量裡出現一連串相同的值，只保留第一個值即可，
// 其他重複的值全都移除掉。
```

```
//
// @param values 想要進行壓縮的向量
// @param isEqual 所要使用的比較函式

template <class T>
void compressVector(
    vector<T> & values,
    bool (* isEqual)(const T &, const T &))
{
    int count = values.size();
    if (count == 0)
        return;

    // 只要不同於前一個值，就把它複製起來

    int destIndex = 1;
    for (int sourceIndex = 1; sourceIndex < count; ++sourceIndex)
    {
        if (!isEqual(values[destIndex - 1], values[sourceIndex]))
        {
            values[destIndex++] = values[sourceIndex];
        }
    }

    values.resize(destIndex);
}
```

雖然不像第一輪只改動註解說明那麼安全，但這一輪改動還是蠻安全的。這些改動確實有可能引入某個 bug——比如說，你可能會把 `destIndex` 不小心改成了 `sourceIndex`。雖然實際上不太可能，但還是有機會發生這種事。

雜草的鑑定

定義你所發現的問題是否為雜草，主要的考慮就是安不安全。如果你能很安全地修正這個問題，那它應該就是要被拔掉的雜草。修正註解說明裡的錯別字，絕對是很安全的。至於註解說明裡更實質性的錯誤（例如我們在第一輪改動裡清除掉的那些模棱兩可的說明），修正這類問題也是很安全的……只要你可以確定自己對函式的認知與理解是正確的，那就沒問題了！

你也可以很安全地解決命名的問題。只要針對原始程式碼其中的一部分，進行「搜索 + 替換」就可以了，而且編譯器或許也會幫你抓出一些錯誤，至少對於像 C++ 這類的編譯語言來說，編譯器確實可以幫上一點忙。

在第二輪的改動中，我在幫某些變數重新命名時，還順便稍微移動了位置。這樣做應該還算安全，不過並不像其他改動那麼安全。這或許依然可以算雜草，不過也可以說比較沒那麼像雜草了。

這裡顯然還是有某種程度上的區別！到目前為止我所做過的改動，全都跟程式碼的功能沒什麼關係——編譯器大概都會生成相同的程式碼。我們基本上就是在不影響功能的情況下，提高了程式碼的可讀性和一致性。

你可以想像，一定還有更多不會影響程式碼功能的實質性改動，例如改動物件類別成員的名稱，這有可能會改動到許多原始檔案，每一處改動都必須保持住一致性；另外，你也有可能針對某個物件類別，預先寫出一些新的用法說明。只要沒影響到程式碼的功能，都可以當作雜草拔掉。也許你只是想把細節做對，只要沒改變到功能，例如只是移動變數或幫變數重新命名，這些或許都可以算是雜草，只不過一定要多加小心就是了。

如果你主動去修改功能，那就不能再算是拔雜草而已了：你在處理的就是一個 bug，所以一定要遵守不同的規則。你可以用自動的方式把雜草拔掉；但你絕對不能用自動的方式去修正某個 bug，因為這樣經常會引入新的問題。根據定義，拔草並不會引入新的問題。

拔草是很容易的，沒有雜草的程式碼庫，一定可以讓你工作起來更加愉快……問題是，為什麼大多數的專案都有這麼多的雜草呢？

程式碼裡的雜草是怎麼長出來的？

好吧，要拔掉雜草很容易，不過，忽略掉它也很容易。我們所要做的工作，永遠都比時間多。拔掉雜草的成本雖然很低，但你當下就是要花時間去做；不過，它的好處會慢慢擴散，通常都要過一陣子你才會有感覺。你的眼睛天生就很容易沒注意到它的存在。

此外，對某個程式設計者來說很像雜草的東西，在另一個程式設計者眼中卻有可能是一朵花[1]。或許你對某些註解說明感到很疑惑，心裡很懷疑它也許有點問題，但你對程式碼的理解又沒有足夠的信心，不知道該不該對它進行

1 園丁有句格言是這麼說的，「雜草只不過是種錯地方的植物。」這讓我想起有一次，我想給老婆一個驚喜——我決定偷偷幫我老婆的菜園除草。結果那次驚喜不足、驚嚇有餘，因為我把她剛種的蘆筍全都拔光了。如果我真正的目標，是永遠不再被要求去除草……嗯，那我的確達成目標了。

改動。這時候你可以更徹底查看程式碼，也可以請比較熟悉程式碼的人，幫你再次檢查你所懷疑的東西，但是（請參閱前一段）你還有一大堆工作等著你去完成，修正註解說明這件事，根本就排不進你的工作計畫中。

你也有可能會在團隊新成員所寫出的程式碼裡，看到某些格式上的問題。雖然你可以靠自己把那些問題修正過來，但你很容易就會告誡自己別去做修正，因為這樣一來新的團隊成員就永遠沒機會學會正確的格式了。最好還是在下次進行程式碼審查時，再去把那些錯誤揪出來吧。

雖然要拔掉雜草很容易，但是讓雜草留在原處，幾乎也是同樣容易的事。那些讓你不願意拔掉雜草的因素——你有更重要的問題需要去關注啦、不確定那些雜草是否真的是雜草啦——這些理由全都是很真實的。

但是，雜草只會滋生出更多的雜草。你可能有一套明確定義的命名和格式約定慣例，但如果你的專案到處都是這種雜草叢生的不合格程式碼，最後就沒有人知道該相信什麼了。大家究竟應該相信約定慣例，還是相信程式碼呢？我很清楚這種情況下會發生什麼事：大家只會聳聳肩，然後去採用他們自己覺得比較舒服的那個做法。

註解說明讓你感到困惑嗎？那它一定也會讓下一個看到它的人感到困惑。你往往會很驚訝地發現，在修正註解說明的過程中——檢查函式有沒有按照你的想法在運作、檢查它周圍的程式碼有沒有做出正確的假設、在進行程式碼審查時討論要不要加入一些新的註解說明——情況到最後經常都會演變成，真的解決掉程式碼裡某個「真正」的 bug。

拔草這件事做起來就是快。這種事根本不需要特別安排時間，想到就做就對了。如果需要相當長的時間來做，那就不能算是雜草了。

我們 Sucker Punch 公司很看重拔草這件事，而它之所以有效果，主要是因為大家對於雜草是什麼都很清楚。根據守則 12，我們確實擁有很強大而嚴格的團隊約定慣例。許多拔草的工作，確實可以修正一些不符合約定慣例的東西。這同時也會強化約定慣例本身的地位——尤其是改動都需要進行審查，而在審查過程中，你們兩個人都會看到不合規與合規的版本，也會看到除草前與除草後的差別，然後同意那的確是要拔掉的雜草。如果身為一個程式碼審查者，你認為接受審查者其實是在不重要的問題上浪費時間，那麼真正的問題其實是，你們對於什麼是重要的事，各有不同的看法。那才是你們真正應該要去解決的問題。