# THE MYTHICAL MAN-MONTH

# 人月神話

FREDERICK P. BROOKS, JR.

翻譯：Adams Wang

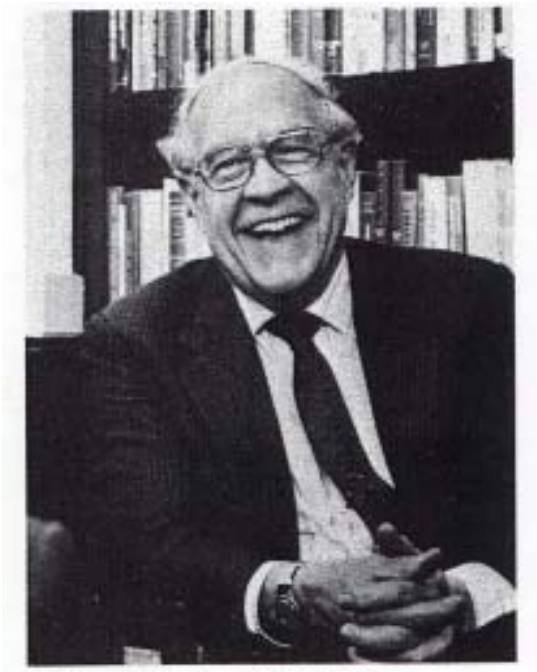


Photo credit: © Jerry Markatos

## 關於作者

Frederick P. Brooks，Jr.是北卡羅來納大學Kenan-Flagler商學院的電腦科學教授，北卡來羅來納大學位於美國北卡來羅來納州的查布林希爾。Brooks被認爲是“IBM 360系統之父”，他擔任了360系統的專案經理，以及360作業系統專案設計階段的經理。憑藉在上述項目的傑出貢獻，他、Bob Evans和Erich Bloch在1985年榮獲了美國國家技術獎（National Medal of Techology）。早期，Brooks曾擔任IBM Stretch和Harvest電腦的體系結構師。

在查布林希爾，Brooks博士創立了電腦科學系，並在1964至1984年期間擔任主席。他曾任職於美國國家科技局和國防科學技術委員會。Brooks目前的教學和研究方向是電腦體系結構、分子模型繪圖和虛擬環境。

## 1975年版獻辭

致兩位特別豐富了我IBM歲月的人：
Thomas J. Watson, Jr.，
他對人們的關懷在他的公司依然無所不在
和
Bob O. Evans，

他大膽的領導使工作成爲了探險。

## 1995年版獻辭

致Nancy，
上帝賜給我的禮物。

# 二十周年紀念版序言（*Preface to the 20th Anniversary Edition*）

令我驚奇和高興的是，《人月神話》在20年後仍然繼續流行，印數超過了250,000。人們經常問，我在1975年提出的觀點和建議，哪些是我仍然堅持的，哪些是已經改變觀點的，是怎樣改變的？儘管我在一些講座上也分析過這個問題，我還是一直想把它寫成文章。

Peter Gordon現在是Addison-Wesley的出版夥伴，他從1980年開始和我共事。他非常耐心，對我幫助很大。他建議我們準備一個紀念版本。我們決定不對原版本做任何修訂，只是原封不動地重印（除了一些細小的修正），並用更新的思想來擴充它。

第16章重印了一篇在1986年IFIPS會議上的論文《沒有銀彈：軟體工程的根本和次要問題》。這篇文章來自我在國防科學委員會主持軍用軟體方面研究時的經驗。我當時的研究合作者，也是我的執行秘書，Robert L. Patrick幫助我回想和感受那些做過的軟體大專案。1987年，IEEE的《電腦》雜誌重印了這篇論文，使它傳播得更廣了。

《沒有銀彈》被證明是富有煽動性的，它預言十年內沒有任何編程技巧能夠給軟體的生產率帶來數量級上的提高。十年只剩下一年了，我的預言看來安全了。《沒有銀彈》激起了越來越多文字上的劇烈爭論，比《人月神話》還要多。因此在第17章，我對一些公開的批評作了說明，並更新了在1986年提出的觀點。

在準備《人月神話》的回顧和更新時，一直進行的軟體工程研究和經驗已經批評、證實和否定了少數書中斷言的觀點，也影響了我。剝去輔助的爭論和資料後，把那些觀點粗略地分類，對我來說很有幫助。我在第18章列出這些觀點的概要，希望這些單調的陳述能夠引來爭論和證據，然後得到證實、否定、更新或精煉。

第19章是一篇更新的短文。讀者應該注意的是，新觀點並不象原來的書一樣來自我的親身經歷。我在大學裏工作，不是在工業界，做的是小規模的項目，不是大項目。自1986年以來，

我就只是教授軟體工程，不再做這方面的研究。我現在的研究領域是虛擬環境及其應用。

在這次回顧的準備過程中，我找了一些正工作在軟體工程領域的朋友，徵求他們的當前觀點。他們很樂意和我共用他們的想法，並仔細地對草稿提出了意見，這些都使我重新受到啓發。感謝Barry Boehm、Ken Brooks、Dick Case、James Coggins、Tom Demarco、Jim McCarthy、David Parnas、Earl Wheeler和Edward Yourdon。感謝Fay Eard出色地對新的章節進行了技術加工。

感謝我在國防科學委員會軍事軟體工作組的同事Gordon Bell、Bruce Buchanan、Rick Hayes-Roth，特別是David Parnas，感謝他們的洞察力和生動的想法。感謝Rebekah Bierly對16章的論文進行了技術加工。我把軟體問題分成"根本的"和"次要的"，這是受Nancy Greenwood Brooks的啓發，她在一篇Suzuki小提琴教育的論文中應用了這樣的分析方法。

在1975年版本的序言中，Addison-Wesleys出版社按規定不允許我向它的一些扮演了關鍵角色的員工致謝。有兩個人的貢獻必須被特別提到：執行編輯Norman Stanton和美術指導Herbert Boes。Boes設計了優雅的風格，他在評注時特別提到："頁邊的空白要寬，字體和版面要有想像力"。更重要的是，他提出了至關重要的建議：爲每一章的開頭配一幅圖片（當時我只有"焦油坑"和"蘭斯大教堂"的圖片）。尋找這些圖片使我多花了一年的時間，但我永遠感激這個忠告。

Soli Deo gloria—願神獨得榮耀。

查珀爾希爾，北卡羅來納 F.P.B., Jr.

1995年3月

# 第一版序言（*Preface to the First Edition*）

在很多方面，管理一個大型的電腦編程專案和其他行業的大型工程很相似——比大多數程式師所認爲的還要相似；在很多另外的方面，它又有差別——比大多數職業經理所認爲的差別還要大。

這個領域的知識在累積。現在AFIPS（美國資訊處理學會聯合會）已經有了一些討論和會議，也出版了一些書籍和論文，但是還沒有成型的方法來系統地進行闡述。提供這樣一本主要反映個人觀點的小書看來是合適的。

雖然我原來從事電腦科學的編程方面的工作，但是在1956-1963年間自動控制程式和高階

語言編譯器開發出來的時候，我主要參加的是硬體構架方面的工作。在1964年，我成爲作業系統OS/360的經理，發現前些年的進展使編程世界改變了很多。

管理OS/360的開發是很有幫助的經歷，雖然是失敗的。那個團隊，包括我的繼任經理F. M. Trapnell，有很多值得自豪的東西。那個系統包括了很多優秀的設計和實施，成功地應用在很多領域，特別是設備無關的輸入輸出和外部庫管理，被很多技術革新廣泛複製。它現在是十分可靠的，相當有效，和非常通用的。

但是，並不是所有的努力都是成功的。所有OS/360的用戶很快就能發現它應該做得更好。設計和實現上的缺陷在控制程式中特別普遍，相比之下，語言編譯器就好得多。大多數這些缺陷發生在1964-1965年的設計階段，所以這肯定是我的責任。此外，這個產品發佈推遲了，需要的記憶體比計畫中的要多，成本也是估計的好幾倍，而且第一次發佈時並不能很好地運行，直到發佈了幾次以後。

就象當初接受OS/360的任務時協商好的，在1965年離開IBM後，我來到查珀爾希爾。我開始分析OS/360的經驗，看能不能從中學到什麼管理和技術上的教訓。特別地，我要說明System/360硬體開發和OS/360軟體發展中的管理經驗是非常不同的。對Tom Watson關於爲什麼編程難以管理的探索性問題，這本書是一份遲來的答案。

在這次探索中，我和1964-65年的經理助理R.P.Case，還有1965-68年的經理F.M.Trapnell，進行了長談，從中受益良多。我對比了其他大型編程專案的經理的結論，包括M.I.T.的F.J.Corbato，Bell電話實驗室的V.Vyssotsky，International Computers Limited的Charles Portman，蘇聯科學院西伯利亞分部計算實驗室的A.P.Ershov，和IBM的A.M.Pietrasanta。

我自己的結論體現在下面的文字中，送給職業程式師、職業經理、特別是程式師的職業經理。

雖然寫出來的是分離的章節，還是有一個中心的論點，特別包含在第2-7章。簡言之，我相信由於人員的分工，大型編程專案碰到的管理問題和小專案區別很大；我相信關鍵需要是維持產品自身的概念完整性。這些章節探討了其中的困難和解決的方法。後續的章節探討軟體工程管理的其他方面。

這個領域的文獻並不多，但散佈很廣。因此我嘗試給出參考資料，說明某個特定知識點和指引感興趣的讀者去看其他有用的工作。很多朋友讀過了本書的手稿，其中一些朋友給出了很有幫助的意見。這些意見很有價值，但爲了不打亂文字的通順，我把它們作爲注解包含在書中。

因爲這本書是隨筆不是課本，所有的參考文獻和注解都被放到書的末尾，建議讀者在讀第一遍時略去不看。

深切感謝Sara Elizabeth Moore小姐，David Wagner先生，和Rebecca Burris夫人，他們幫助我準備了手稿。感謝Joseph C.Sloane教授在圖解方面的建議。

查珀爾希爾，北卡羅來納 F.P.B., Jr

1974年10月

# 目錄（*Contents*）


二十周年紀念版序言（***PREFACE TO THE 20TH ANNIVERSARY EDITION***）.....**I**

第一版序言（***PREFACE TO THE FIRST EDITION***）.....**III**

目錄（***CONTENTS***）.....**V**

焦油坑（***THE TAR PIT***）.....**1**

- 編程系統產品.....1
- 職業的樂趣.....3
- 職業的苦惱.....4

人月神話（***THE MYTHICAL MAN-MONTH***）.....**6**

- 樂觀主義.....7
- 人月.....8
- 系統測試.....10
- 空泛的估算.....11
- 重複產生的進度災難.....12

外科手術隊伍（***THE SURGICAL TEAM***）.....**16**

- 問題.....16
- MILLS的建議.....17
- 如何運作.....20
- 團隊的擴建.....21

貴族專制、民主政治和系統設計（***ARISTOCRACY, DEMOCRACY, AND SYSTEM DESIGN***）.....**22**

- 概念一致性.....22
- 獲得概念的完整性.....23
- 貴族專制統治和民主政治.....24
- 在等待時，實現人員應該做什麼？.....26

畫蛇添足（***THE SECOND-SYSTEM EFFECT***）.....**29**

結構師的交互準則和機制.....29
自律——開發第二個系統所帶來的後果.....30

貫徹執行（***PASSING THE WORD***）.....**33**

文檔化的規格說明——手冊.....33
形式化定義.....34
直接整合.....36
會議和大會.....36
多重實現.....38
電話日誌.....38
產品測試.....38

爲什麼巴比倫塔會失敗？（***WHY DID THE TOWER OF BABEL FAIL?***）.....**40**

巴比倫塔的管理教訓.....41
大型編程專案中的交流.....41
專案工作手冊.....42
大型編程專案的組織架構.....44

胸有成竹（***CALLING THE SHOT***）.....**49**

PORTMAN的數據.....50
ARON的數據.....51
HARR的數據.....51
OS/360的數據.....53
CORBATO的數據.....53

削足適履（***TEN POUNDS IN A FIVE-POUND SACK***）.....**55**

作爲成本的程式空間.....55
規模控制.....56
空間技能.....57
資料的表現形式是編程的根本.....58

提綱挈領（***THE DOCUMENTARY HYPOTHESIS***）.....**60**

電腦產品的文檔.....60
大學科系的文檔.....62
軟體專案的文檔.....62
爲什麼要有正式的文檔？.....63

未雨綢繆（***PLAN TO THROW ONE AWAY***）.....**64**

試驗性工廠和增大規模.....64
唯一不變的就是變化本身.....65
爲變更計畫系統.....66
爲變更計畫組織架構.....66
前進兩步，後退一步.....68
前進一步，後退一步.....69

幹將莫邪（***SHARP TOOLS***）.....**71**

目的機器.....72
輔助機器和資料服務.....73
高階語言和互動式編程.....76

整體部分（***THE WHOLE AND THE PARTS***）.....**78**

剔除BUG的設計.....78
構件單元調試.....80
系統集成調試.....82

禍起蕭牆（***HATCHING A CATASTROPHE***）.....**85**

里程碑還是沉重的負擔？.....85
"其他的部分反正會落後".....86
地毯的下麵.....87

另外一面（***THE OTHER FACE***）.....**92**

需要什麼樣的文檔.....93
流程圖.....95
自文檔化（SELF-DOCUMENTING）的程式.....96

沒有銀彈－軟體工程中的根本和次要問題(***NO SILVER BULLET – ESSENCE AND ACCIDENT IN SOFTWARE ENGINEERING***）.....**102**

摘要1.....102
介紹.....103
是否一定那麼困難呢？——根本困難.....103
以往解決次要困難的一些突破.....106
銀彈的希望.....108
針對概念上根本問題的頗具前途的方法.....113
*NO*.....*118*

再論《沒有銀彈》（***"NO SILVER BULLET"REFIRED***）.....**120**

人狼和其他恐怖傳說.....120
存在著銀彈－就在這裏！.....121
含糊的表達將會導致誤解.....121
HAREL的分析.....124
JONE的觀點——品質帶來生產率.....127
那麼，生產率的情形如何？.....128
面向物件編程——這顆銅質子彈可以嗎？.....129
重用的情況怎樣？.....130
學習大量的辭彙——對軟體重用的一個可預見，但還沒有被預言的問題.....132
子彈的本質——形勢沒有發生改變.....133

《人月神話》的觀點：是或非？(***PROPOSITIONS OF* THE MYTHICAL MAN-MONTH: *TRUE OR FALSE***？）.....**134**

第1章 焦油坑.....134
第2章 人月神話.....135

第3章 外科手術隊伍.....136
第4章 貴族專制、民主政治和系統設計.....137
第5章 畫蛇添足.....137
第6章 貫徹執行.....138
第7章 爲什麼巴比倫塔會失敗？.....139
第8章 胸有成竹.....141
第9章 削足適履.....141
第10章 提綱挈領.....143
第11章 未雨綢繆.....143
第12章 幹將莫邪.....146
第13章 整體部分.....148
第14章 禍起蕭牆.....149
第15章 另外一面.....150
原著結束語.....152

**20年後的人月神話（THE MYTHICAL MAN-MONTH *AFTER* 20 YEARS）.....153**

爲什麼會出現二十周年紀念版本？.....153
核心觀點：概念完整性和結構師.....154
開發第二個系統所引起的後果：盲目的功能和頻率猜測.....156
圖形（WIMP）介面的成功.....157
沒有構建捨棄原型——瀑布模型是錯誤的！.....160
增量開發模型更佳——漸進地精化.....162
關於資訊隱藏，PARNAS是正確的，我是錯誤的.....165
人月到底有多少神話色彩？BOEHM的模型和資料.....167
人就是一切（或者說，幾乎是一切）.....168
放棄權力的力量.....169
最令人驚訝的新事物是什麼？數百萬的電腦.....171
全新的軟體產業——塑膠薄膜包裝的成品軟體.....173
買來開發——使用塑膠包裝的成品套裝軟體作爲構件.....174
軟體工程的狀態和未來.....176

結束語：令人嚮往、激動人心和充滿樂趣的五十年（***EPILOGUE FIFTY YEARS OF WONDER, EXCITEMENT, AND JOY***）**.....178**

注解和參考文獻（***NOTES AND REFERENCES***）**.....180**

第1章.....180
第2章.....180
第3章.....180
第4章.....181
第5章.....181
第6章.....182
第7章.....182
第8章.....182
第9章.....183
第10章.....183
第11章.....184
第12章.....184

第13章.....185
第14章.....186
第15章.....187
第16章.....187
第17章.....188
第18章.....190
第19章.....190

索引（***INDEX***）.....**193**

# 焦油坑（*The Tar Pit*）

岸上的船兒，如同海上的燈塔，無法移動。

- 荷蘭諺語

*Een schip op het strand is een baken in zee.*

*[A ship on the beach is a lighthouse to the sea.]*

*- DUTCH PROVERB*

史前史中，沒有別的場景比巨獸在焦油坑中垂死掙扎的場面更令人震撼。上帝見證著恐龍、猛獁象、劍齒虎在焦油中掙扎。它們掙扎得越是猛烈，焦油糾纏得越緊，沒有任何猛獸足夠強壯或具有足夠的技巧，能夠掙脫束縛，它們最後都沉到了坑底。

過去幾十年的大型系統開發就猶如這樣一個焦油坑，很多大型和強壯的動物在其中劇烈地掙扎。他們中大多數開發出了可運行的系統——不過，其中只有非常少數的專案滿足了目標、時間進度和預算的要求。各種團隊，大型的和小型的，龐雜的和精幹的，一個接一個淹沒在了焦油坑中。表面上看起來好像沒有任何一個單獨的問題會導致困難，每個都能被解決，但是當它們相互糾纏和累積在一起的時候，團隊的行動就會變得越來越慢。對問題的麻煩程度，每個人似乎都會感到驚訝，並且很難看清問題的本質。不過，如果我們想解決問題，就必須試圖先去理解它。

因此，首先讓我們來認識一下軟體發展這個職業，以及充滿在這個職業中的樂趣和苦惱吧。

## 編程系統產品

報紙上經常會出現這樣的新聞，講述兩個程式師如何在經改造的簡陋車庫中，編出了超過大型團隊工作量的重要程式。接著，每個編程人員準備相信這樣的神話，因爲他知道自己能以超過產業化團隊的1000代碼行/年的生產率來開發任何程式。

爲什麼不是所有的產業化隊伍都會被這種專注的二人組合所替代？我們必須看一下產出的是什麼。

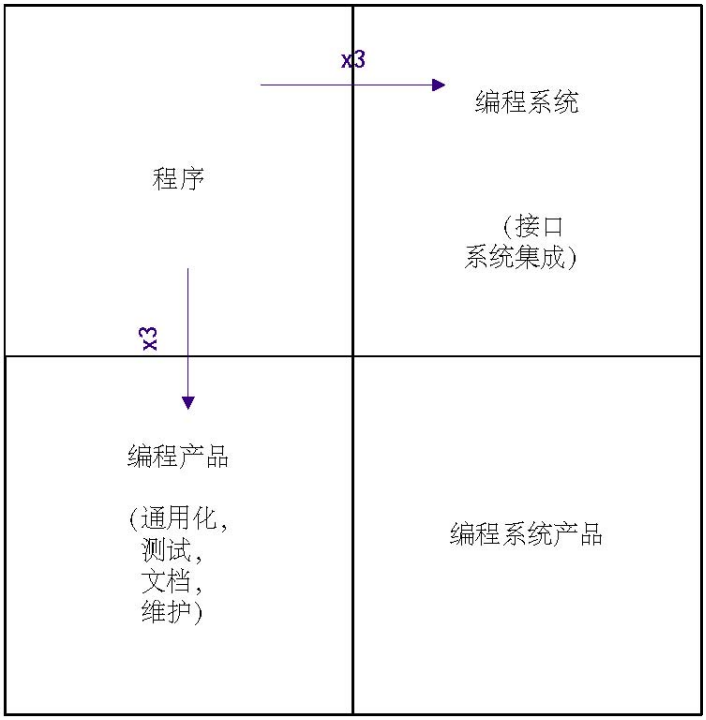



圖1.1：編程系統產品的演進

在圖1.1的左上部分是程式（Program）。它本身是完整的，可以由作者在所開發的系統平臺上運行。它通常是車庫中產出的產品，以及作爲單個程式師生產率的評估標準。

有兩種途徑可以使程式轉變成更有用的，但是成本更高的東西，它們表現爲圖中的邊界。

水準邊界以下，程式變成編程產品（Programming Product）。這是可以被任何人運行、測試、修復和擴展的程式。它可以運行在多種作業系統平臺上，供多套數據使用。要成爲通用的編程產品，程式必須按照普遍認可的風格來編寫，特別是輸入的範圍和形式必須擴展，以適用於所有可以合理使用的基本演算法。接著，對程式進行徹底測試，確保它的穩定性和可靠性，使其值得信賴。這就意味著必須準備、運行和記錄詳盡的測試用例庫，用來檢查輸入的邊界和

範圍。此外，要將程式提升爲程式產品，還需要有完備的文檔，每個人都可以加以使用、修復和擴展。經驗資料表明，相同功能的編程產品的成本，至少是已經過測試的程式的三倍。

回到圖中，垂直邊界的右邊，程式變成編程系統（Programming System）中的一個構件單元。它是在功能上能相互協作的程式集合，具有規範的格式，可以進行交互，並可以用來組裝和搭建整個系統。要成爲系統構件，程式必須按照一定的要求編制，使輸入和輸出在語法和語義上與精確定義的介面一致。同時程式還要符合預先定義的資源限制——記憶體空間、輸入輸出設備、計算機時間。最後，程式必須同其他系統構件單元一道，以任何能想像到的組合進行測試。由於測試用例會隨著組合不斷增加，所以測試的範圍非常廣。因爲一些意想不到的交互會產生許多不易察覺的bug，測試工作將會非常耗時，因此相同功能的編程系統構件的成本至少是獨立程式的三倍。如果系統有大量的組成單元，成本還會更高。

圖1.1的右下部分代表編程系統產品（Programming Systems Product）。和以上的所有的情況都不同的是，它的成本高達九倍。然而，只有它才是真正有用的產品，是大多數系統開發的目標。

## 職業的樂趣

編程爲什麼有趣？作爲回報，它的從業者期望得到什麼樣的快樂？

首先是一種創建事物的純粹快樂。如同小孩在玩泥巴時感到愉快一樣，成年人喜歡創建事物，特別是自己進行設計。我想這種快樂是上帝創造世界的折射，一種呈現在每片獨特、嶄新的樹葉和雪花上的喜悅[1]。

其次，快樂來自于開發對其他人有用的東西。內心深處，我們期望其他人使用我們的勞動成果，並能對他們有所幫助。從這個方面，這同小孩用粘土爲“爸爸辦公室”捏制鉛筆盒沒有本質的區別。

第三是整個過程體現出魔術般的力量——將相互嚙合的零部件組裝在一起，看到它們精妙地運行，得到預先所希望的結果。比起彈珠遊戲或點唱機所具有的迷人魅力，程式化的電腦毫不遜色。

第四是學習的樂趣，來自于這項工作的非重複特性。人們所面臨的問題，在某個或其他方面總有些不同。因而解決問題的人可以從中學習新的事物：有時是實踐上的，有時是理論上的，或者兼而有之。

最後，樂趣還來自於工作在如此易於駕馭的介質上。程式師，就像詩人一樣，幾乎僅僅工作在單純的思考中。程式師憑空地運用自己的想像，來建造自己的“城堡”。很少有這樣的介質——創造的方式如此得靈活，如此得易於精煉和重建，如此得容易實現概念上的設想。（不過我們將會看到，容易駕馭的特性也有它自己的問題）

然而程式畢竟同詩歌不同，它是實實在在的東西；可以移動和運行，能獨立產生可見的輸出；能列印結果，繪製圖形，發出聲音，移動支架。神話和傳說中的魔術在我們的時代已變成了現實。在鍵盤上鍵入正確的咒語，螢幕會活動、變幻，顯示出前所未有的或是已經存在的事

物。

編程非常有趣，在於它不僅滿足了我們內心深處進行創造的渴望，而且還愉悅了每個人內在的情感。

## 職業的苦惱

然而這個過程並不全都是喜悅。我們只有事先瞭解一些編程固有的煩惱，這樣，當它們真的出現時，才能更加坦然地面對。

首先，必須追求完美。因爲電腦也是以這樣的方式來變戲法：如果咒語中的一個字元、一個停頓，沒有與正確的形式一致，魔術就不會出現。（現實中，很少的人類活動要求完美，所以人類對它本來就不習慣。）實際上，我認爲學習編程的最困難部分，是將做事的方式往追求完美的方向調整。

其次，是由他人來設定目標，供給資源，提供資訊。編程人員很少能控制工作環境和工作目標。用管理的術語來說，個人的權威和他所承擔的責任是不相配的。不過，似乎在所有的領域中，對要完成的工作，很少能提供與責任相一致的正式權威。而現實情況中，實際（相對於正式）的權威來自于每次任務的完成。

對於系統編程人員而言，對其他人的依賴是一件非常痛苦的事情。他依靠其他人的程式，而往往這些程式設計得並不合理，實現拙劣，發佈不完整（沒有源代碼或測試用例），或者文檔記錄得很糟。所以，系統編程人員不得不花費時間去研究和修改，而它們在理想情況下本應該是可靠完整的。

下一個煩惱——概念性設計是有趣的，但尋找瑣碎的bug卻只是一項重複性的活動。伴隨著創造性活動的，往往是枯燥沉悶的時間和艱苦的勞動。程式編制工作也不例外。

另外，人們發現調試和查錯往往是線性收斂的，或者更糟糕的是，具有二次方的複雜度。結果，測試一拖再拖，尋找最後一個錯誤比第一個錯誤將花費更多的時間。

最後一個苦惱，有時也是一種無奈——當投入了大量辛苦的勞動，產品在即將完成或者終於完成的時候，卻已顯得陳舊過時。可能是同事和競爭對手已在追逐新的、更好的構思；也許替代方案不僅僅是在構思，而且已經在安排了。

現實情況比上面所說的通常要好一些。當產品開發完成時，更優秀的新產品通常還不能投入使用，而僅僅是爲大家談論而已。另外，它同樣需要數月的開發時間。事實上，只有實際需要時，才會用到最新的設想，因爲所實現的系統已經能滿足要求，體現了回報。

誠然，產品開發所基於的技術在不斷地進步。一旦設計被凍結，在概念上就已經開始陳舊了。不過，實際產品需要一步一步按階段實現。實現落後與否的判斷應根據其他已有的系統，而不是未實現的概念。因此，我們所面臨的挑戰和任務是在現有的時間和有效的資源範圍內，尋找解決實際問題的切實可行方案。

這，就是編程。一個許多人痛苦掙扎的焦油坑以及一種樂趣和苦惱共存的創造性活動。對於許多人而言，其中的樂趣遠大於苦惱。而本書的剩餘部分將試圖搭建一些橋樑，爲通過這樣的焦油坑提供一些指導。

# 人月神話（*The Mythical Man-Month*）

美酒的釀造需要年頭，美食的烹調需要時間；片刻等待，更多美味，更多享受。

- 新奧爾良*Antoine*餐廳的菜單

*Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.*

*- MENU OF RESTAURANT ANTOINE, NEW ORLEANS*

在眾多軟體專案中，缺乏合理的時間進度是造成專案滯後的最主要原因，它比其他所有因素加起來的影響還大。導致這種普遍性災難的原因是什麼呢？

首先，我們對估算技術缺乏有效的研究，更加嚴肅地說，它反映了一種悄無聲息，但並不真實的假設——一切都將運作良好。

第二，我們採用的估算技術隱含地假設人和月可以互換，錯誤地將進度與工作量相互混淆。

第三，由於對自己的估算缺乏信心，軟體經理通常不會有耐心持續地進行估算這項工作。

第四，對進度缺少跟蹤和監督。其他工程領域中，經過驗證的跟蹤技術和常規監督程式，在軟體工程中常常被認爲是無謂的舉動。

第五，當意識到進度的偏移時，下意識（以及傳統）的反應是增加人力。這就像使用汽油滅火一樣，只會使事情更糟。越來越大的火勢需要更多的汽油，從而進入了一場註定會導致災難的迴圈。

進度監督是另一篇論文的主題，而本文中我們將對問題的其他方面進行更詳細的討論。

## 樂觀主義

所有的編程人員都是樂觀主義者。可能是這種現代魔術特別吸引那些相信美滿結局的人；也可能是成百上千瑣碎的挫折趕走了大多數人，只剩下了那些習慣上只關注結果的人；還可能僅僅因爲電腦還很年輕，程式師更加年輕，而年輕人總是些樂觀主義者——無論是什麼樣的程式，結果是勿庸置疑的："這次它肯定會運行。"或者"我剛剛找出了最後一個錯誤。"

所以系統編程的進度安排背後的第一個假設是：一切都將運作良好，每一項任務僅花費它所"應該"花費的時間。

對這種彌漫在編程人員中的樂觀主義，理應受到慎重的分析。Dorothy Sayers在她的"The Mind of the Maker"一書中，將創造性活動分爲三個階段：構思、實現和交流。書籍、電腦、或者程式的出現，首先是作爲一個構思或模型出現在作者的腦海中，它與時間和空間無關。接著，借助鋼筆、墨水和紙，或者電線、矽片和鐵氧體，在現實的時間和空間中實現它們。然後，當某人閱讀書本、使用電腦和運行程式的時候，他與作者的思想相互溝通，從而創作過程得以結束。

以上Sayers的闡述不僅僅可以描繪人類的創造性活動，而且類似於"基督的教義"，能指導我們的日常工作。對於創造者，只有在實現的過程中，才能發現我們構思的不完整性和不一致性。因此，對於理論家而言，書寫、試驗以及"工作實現"是非常基本和必要的。

在許多創造性活動中，往往很難掌握活動實施的介質，例如木頭切割、油漆、電器安裝等。這些介質的物理約束限制了思路的表達，它們同樣對實現造成了許多預料之外的困難。

由於物理介質和思路中隱含的不完善性，實際實現起來需要花費大量的時間和汗水。對遇到的大部分實現上的困難，我們總是傾向於去責怪那些物理介質，因爲它們不順應"我們"設定的思路。其實，這只不過是我們的驕傲使判斷帶上了主觀主義色彩。

然而，電腦編程基於十分容易掌握的介質，編程人員通過非常純粹的思維活動——概念以及靈活的表現形式來開發程式。正由於介質的易於駕馭，我們期待在實現過程中不會碰到困難，因此造成了樂觀主義的彌漫。而我們的構思是有缺陷的，因此總會有bug。也就是說，我們的樂觀主義並不應該是理所應當的。

在單個的任務中，"一切都將運轉正常"的假設在時間進度上具有可實現性。因爲所遇的延遲是一個概率分佈曲線，"不會延遲"僅具有有限的概率，所以現實情況可能會像計畫安排的那樣順利。然而大型的編程工作，或多或少包含了很多工，某些任務間還具有前後的次序，從而一切正常的概率變得非常小，甚至接近於無。

## 人月

第二個謬誤的思考方式是在估計和進度安排中使用的工作量單位：人月。成本的確隨開發產品的人數和時間的不同，有著很大的變化，進度卻不是如此。因此我認爲用人月作爲衡量一項工作的規模是一個危險和帶有欺騙性的神話。它暗示著人員數量和時間是可以相互替換的。

人數和時間的互換僅僅適用於以下情況：某個任務可以分解給參與人員，並且他們之間不需要相互的交流（圖2.1）。這在割小麥或收穫棉花的工作中是可行的；而在系統編程中近乎不可能。

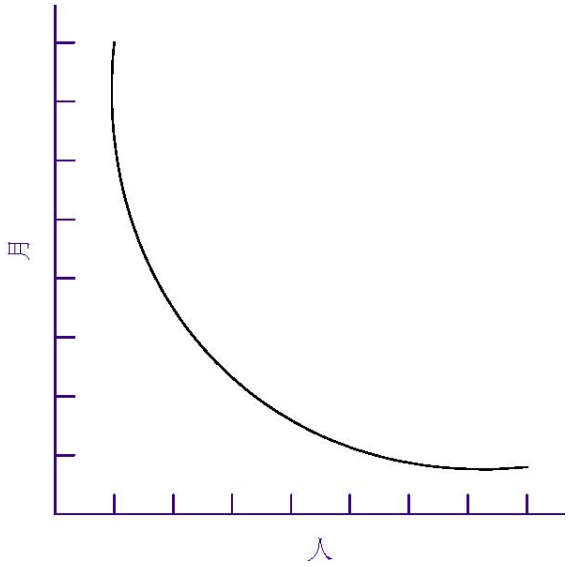



圖2.1：人員和時間之間的關係——完全可以分解的任務

當任務由於次序上的限制不能分解時，人手的添加對進度沒有幫助（圖2.2）。無論多少個母親，孕育一個生命都需要十個月。由於調試、測試的次序特性，許多軟體都具有這種特徵，

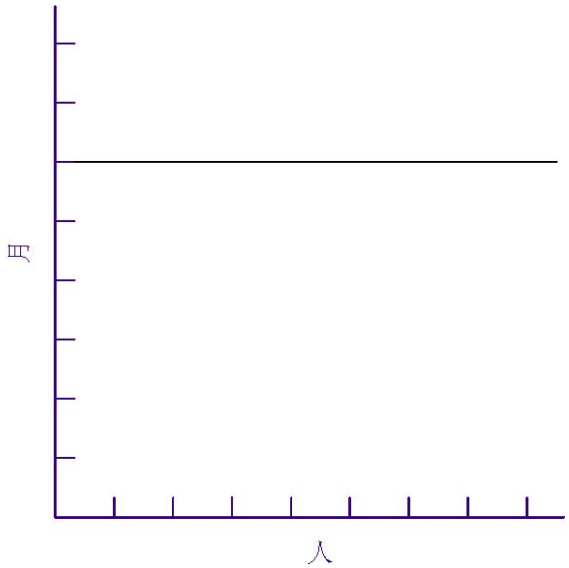



圖2.2：人員和時間之間的關係——無法分解的任務

對於可以分解，但子任務之間需要相互溝通和交流的任務，必須在計畫工作中考慮溝通的

工作量。因此，相同人月的前提下，採用增加人手來減少時間得到的最好情況，也比未調整前要差一些（圖2.3）。

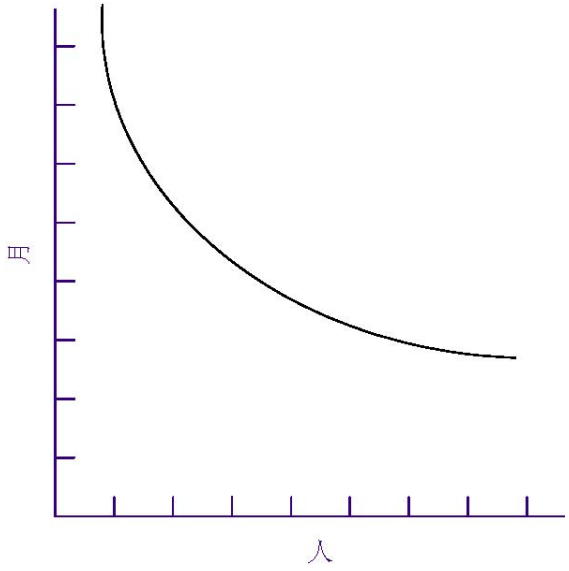



圖2.3：人員和時間之間的關係——需要溝通的可分解任務

溝通所增加的負擔由兩個部分組成，培訓和相互的交流。每個成員需要進行技術、專案目標以及總體策略上的培訓。這種培訓不能分解，因此這部分增加的工作量隨人員的數量呈線性變化[1]。

相互之間交流的情況更糟一些。如果任務的每個部分必須分別和其他部分單獨協作，則工作量按照n(n-1)/2遞增。一對一交流的情況下，三個人的工作量是兩個人的三倍，四個人則是兩個人的六倍。而對於需要在三四個人之間召開會議、進行協商、一同解決的問題，情況會更加惡劣。所增加的用於溝通的工作量可能會完全抵消對原有任務分解所產生的作用，此時我們會被帶到圖2.4的境地。

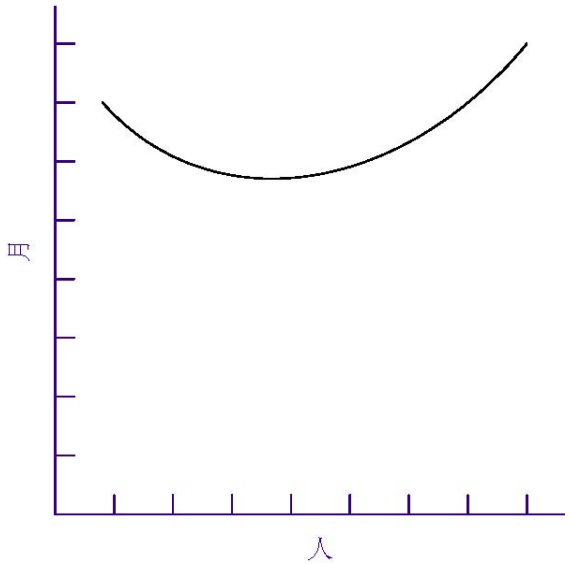



圖2.4：人員和時間之間的關係——關係錯綜複雜的任務

因爲軟體發展本質上是一項系統工作——錯綜複雜關係下的一種實踐——溝通、交流的工作量非常大，它很快會消耗任務分解所節省下來的個人時間。從而，添加更多的人手，實際上是延長了，而不是縮短了時間進度。

## 系統測試

在時間進度中，順序限制所造成的影響，沒有哪個部分比單元調試和系統測試所受到的牽涉更徹底。而且，要求的時間依賴於所遇到的錯誤、缺陷數量以及捕捉它們的程度。理論上，缺陷的數量應該爲零。但是，由於我們的樂觀主義，通常實際出現的缺陷數量比預料的要多得多。因此，系統測試進度的安排常常是編程中最不合理的部分。

對於軟體任務的進度安排，以下是我使用了很多年的經驗法則：

1/3計畫

1/6編碼

1/4構件測試和早期系統測試

1/4系統測試，所有的構件已完成

在許多重要的方面，它與傳統的進度安排方法不同：

1．分配給計畫的時間比尋常的多。即便如此，仍不足以產生詳細和穩定的計畫規格說明，也不足以容納對全新技術的研究和摸索。

2．對所完成代碼的調試和測試，投入近一半的時間，比平常的安排多很多。

3．容易估計的部分，即編碼，僅僅分配了六分之一的時間。

通過對傳統專案進度安排的研究，我發現很少項目允許爲測試分配一半的時間，但大多數項目的測試實際上是花費了進度中一半的時間。它們中的許多專案，在系統測試之前還能保持進度。或者說，除了系統測試，進度基本能保證[2]。

特別需要指出的是，不爲系統測試安排足夠的時間簡直就是一場災難。因爲延遲發生在項目快完成的時候。直到專案的發佈日期，才有人發現進度上的問題。因此，壞消息沒有任何預兆，很晚才出現在客戶和專案經理面前。

另外，此時此刻的延遲具有不尋常的、嚴重的財務和心理上的反應。在此之前，專案已經配置了充足的人員，每天的人力成本也已經達到了最大的限度。更重要的是，當軟體用來支援其他的商業活動（電腦硬體到貨，新設備、服務上線等等）時，這些活動延誤出現即將發佈前，那麼將付出相當高的商業代價。

實際上，上述的二次成本遠遠高於其他開銷。因此，在早期進度策劃時，允許充分的系統測試時間是非常重要的。

## 空泛的估算

觀察一下編程人員，你可能會發現，同廚師一樣，某項任務的計畫進度，可能受限於顧客要求的緊迫程度，但緊迫程度無法控制實際的完成情況。就像約好在兩分鐘內完成一個煎蛋，看上去可能進行得非常好。但當它無法在兩分鐘內完成時，顧客只能選擇等待或者生吃煎蛋。軟體顧客的情況類似。

廚師還有其他的選擇：他可以把火開大，不過結果常常是無法“挽救”的煎蛋——一面已經焦了，而另一面還是生的。

現在，我並不認爲軟體經理內在的勇氣和堅持不如廚師，或者不如其他工程經理。但爲了滿足顧客期望的日期而造成的不合理進度安排，在軟體領域中卻比其他的任何工程領域要普遍得多。而且，非階段化方法的採用，少得可憐的資料支援，加上完全借助軟體經理的直覺，這樣的方式很難生產出健壯可靠和規避風險的估計。

顯然我們需要兩種解決方案。開發並推行生產率圖表、缺陷率、估算規則等等，而整個組織最終會從這些資料的共用上獲益。

或者，在基於可靠基礎的估算出現之前，專案經理需要挺直腰杆，堅持他們的估計，確信自己的經驗和直覺總比從期望派生出的結果要強得多。

# 重複產生的進度災難

當一個軟體專案落後于進度時，通常的做法是什麼呢？自然是加派人手。如圖2.1至2.4所示，這可能有所幫助，也可能無法解決問題。

我們來考慮一個例子[3]。設想一個估計需要12個人月的任務，分派給3個成員4個月時間，在每個月的末尾安排了可測量的里程碑A、B、C、D（圖2.5）。

現在假定兩個月之後，第一個里程碑沒有達到（圖2.6）。項目經理面對的選擇方案有哪些呢？

1．假設任務必須按時完成。假設僅僅是任務的第一個部分估計不得當，即如圖2.6所示，則剩餘了9個人月的工作量，時間還有兩個月，即需要4.5個開發人員，所以需要在原來3個人的基礎上增加2個人。

2．假設任務必須按時完成。假設整個任務的估計偏低，即如圖2.7所示，那麼還有18個人月的工作量以及2個月的時間，需要將原來的3個人增至9個人。

3．重新安排進度。我喜歡P.Fagg，一個具有豐富經驗的硬體工程師的忠告：“避免小的偏差（Take no small slips）”。也就是說，在新的進度安排中分配充分的時間，以確保工作能仔細、徹底地完成，從而無需重新確定時間進度表。

4．削減任務。在現實情況中，一旦開發團隊觀察到進度的偏差，總是傾向於對任務進行削減。當項目延期所導致的後續成本非常高時，這常常是唯一可行的方法。專案經理的相應措施是仔細、正式地調整專案，重新安排進度；或者是默默地注視著任務項由於輕率的設計和不完整的測試而被剪除。

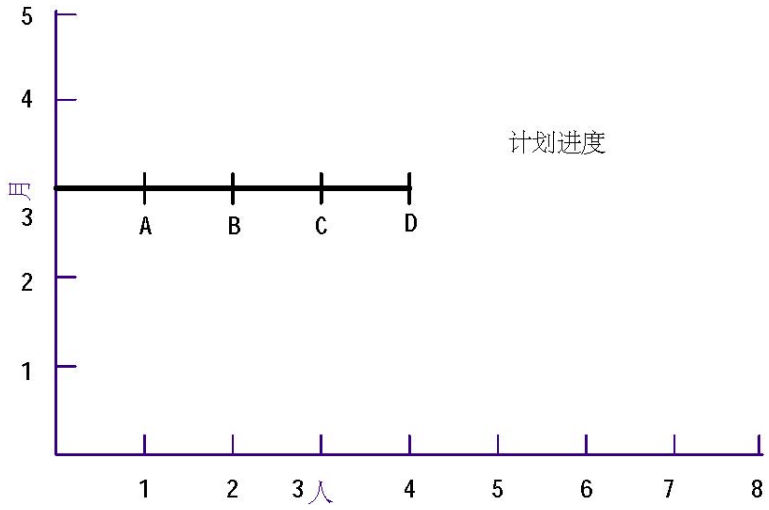



圖2.5

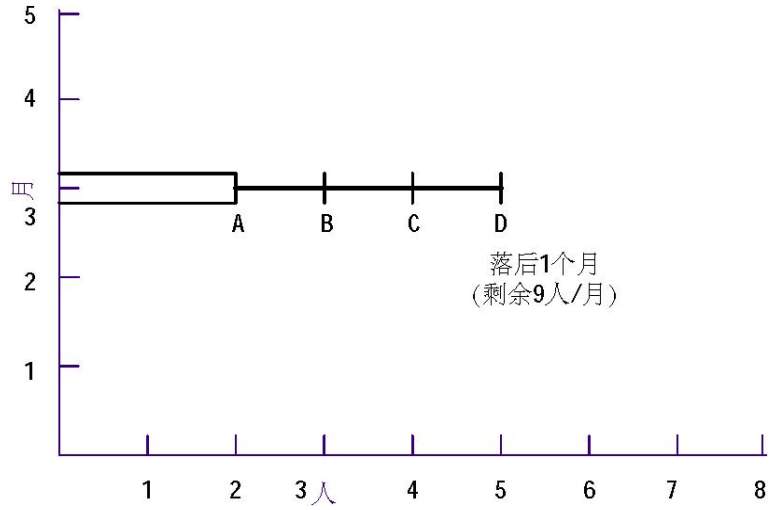

5
4
月
3
2
1
A
B
C
D
落后1个月
（剩余9人/月）
1
2
3人
4
5
6
7
8

圖2.6

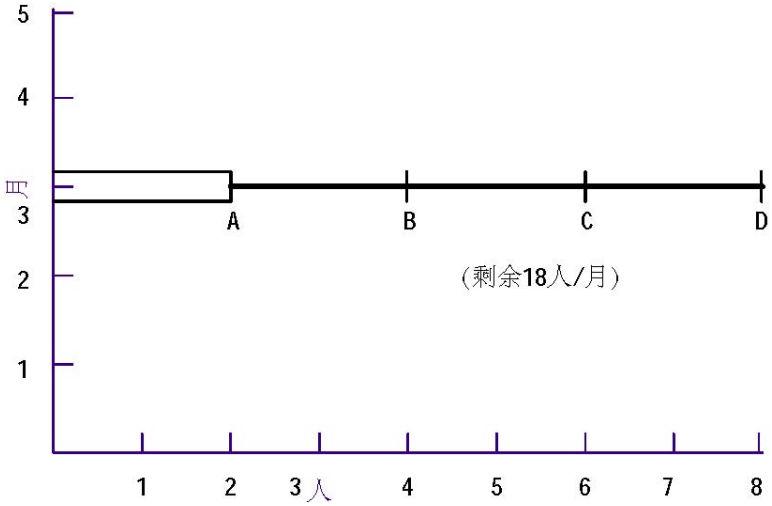

5
4
月
3
2
1
A
B
C
D
（剩余18人/月）
1
2
3人
4
5
6
7
8

圖2.7

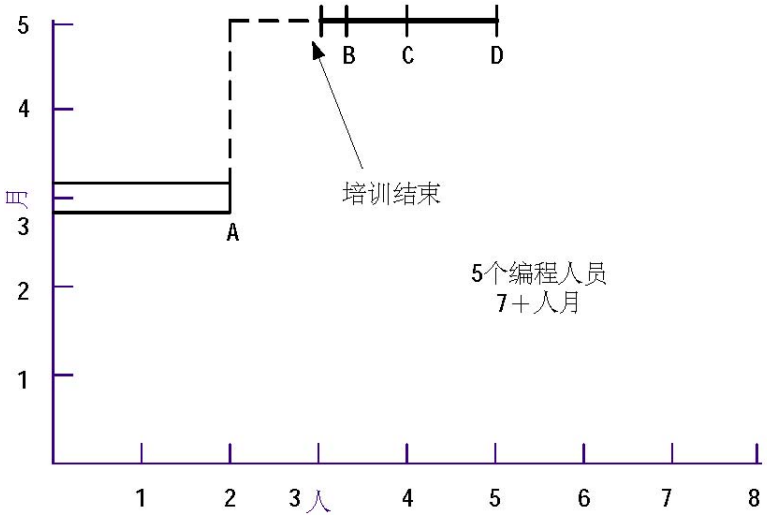



圖2.8

前兩種情況中，堅持把不經調整的任務在四個月內完成將是災難性的。考慮到重複生成的工作量，以第一種爲例（圖2.8）——不論在多短的時間內，聘請到多麼能幹的兩位新員工，他們都需要接受一位元有經驗的職員的培訓。如果培訓需要一個月的時間，那麼三個人月將會投入到原有進度安排以外的工作中。另外，原先劃分爲三個部分的工作，會重新分解成五個部分；某些已經完成的工作必定會丟失，系統測試必須被延長。因此，在第三個月的月末，仍然殘留著7個人月的工作，但此時只有5個有效的人月。如同圖2.8所示，產品還是會延期，如同沒有增加任何人手（圖2.6）。

期望四個月內完成項目，僅僅考慮培訓的時間，不考慮任務的重新劃分和額外的系統測試，在第二個月末需要增添4個，而不是2個人員。如果考慮任務劃分和系統測試的工作量，則還需要繼續增加人手。到那時所擁有的就不是3人的隊伍，而是7人以上的團隊；並且小組的組織和任務的劃分在類型上都不盡相同，這已經不是程度上的差異問題。

注意在第三個月的結尾時，情況看上去還是很糟。除去管理的工作不談，3月1日的里程碑仍未達到。此時，對專案經理而言，仍然存在著很強的誘惑——添加更多人力，結果往往會是上述迴圈的重複。這簡直就是一種瘋狂、愚蠢的做法。

前面的討論僅僅是第一個里程碑估計不當的情況。如果在3月1日，項目經理做出了比較保守的假設，即整個估計過於樂觀了，如圖2.7所示。6個人手需要添加到原先的任務中。培訓、任務的重新分配、系統測試工作量的計算作爲練習留給讀者。但是毫無疑問，重現“災難”所開發出的產品，比沒有增加人手，而是重新安排開發進度所產生的產品更差。

簡單、武斷地重複一下Brooks法則：

向進度落後的專案中增加人手，只會使進度更加落後。（Adding manpower to a late software project makes it later）

這就是除去了神話色彩的人月。專案的時間依賴於順序上的限制，人員的數量依賴於單個

子任務的數量。從這兩個數值可以推算出進度時間表，該表安排的人員較少，花費的時間較長（唯一的風險是產品可能會過時）。相反，分派較多的人手，計畫較短的時間，將無法得到可行的進度表。總之，在眾多軟體專案中，缺乏合理的時間進度是造成專案滯後的最主要原因，它比其他所有因素加起來的影響還要大。

# 外科手術隊伍（*The Surgical Team*）

這些研究表明，效率高和效率低的實施者之間具體差別非常大，經常達到了數量級的水準。

*- SACKMAN, ERIKSON和GRANT*[1]

*These studies revealed large individual differences between high and low performers, often by an order of magnitude.*

*- SACKMAN, ERIKSON, AND GRANT*[1]

在電腦領域的會議中，常常聽到年輕的軟體經理聲稱他們喜歡由頭等人才組成的小型、精幹的隊伍，而不是那些幾百人的大型團隊，這裏的"人"當然暗指平庸的程式師。其實我們也經常有相同的看法。

但這種幼稚的觀點回避了一個很困難的問題——如何在有意義的時間進度內創建大型的系統？那麼就讓我們現在來仔細討論一下這個問題的每一個方面。

## 問題

軟體經理很早就認識到優秀程式師和較差的程式師之間生產率的差異，但實際測量出的差異還是令我們所有的人吃驚。在他們的一個研究中，Sackman、Erikson和Grand曾對一組具有經驗的程式人員進行測量。在該小組中，最好的和最差的表現在生產率上平均爲10:1；在運行速度和空間上具有5:1的驚人差異！簡言之，$20,000/年的程式師的生產率可能是$10,000/年程式師的10倍。資料顯示經驗和實際的表現沒有相互聯繫（我懷疑這種現象是否普遍成立。）

我常常重複這樣的一個觀點，需要協作溝通的人員的數量影響著開發成本，因爲成本的主

要組成部分是相互的溝通和交流，以及更正溝通不當所引起的不良結果（系統調試）。這一點，也暗示系統應該由盡可能少的人員來開發。實際上，絕大多數大型編程系統的經驗顯示出，一擁而上的開發方法是高成本的、速度緩慢的、不充分的，開發出的是無法在概念上進行集成的產品。OS/360、Exec 8、Scope 6600、Multics、TSS、SAFE等等——這個列表可以不斷地繼續下去。

得出的結論很簡單：如果一個200人的專案中，有25個最能幹和最有開發經驗的專案經理，那麼開除剩下的175名程式師，讓專案經理來編程開發。

現在我們來驗證一下這個解決方案。一方面，原有的開發隊伍不是理想的小型強有力的團隊，因爲通常的共識是不超過10個人，而該團隊規模如此之大，以至於至少需要兩層的管理，或者說大約5名管理人員。另外，它需要額外的財務、人員、空間、文秘和機器操作方面的支持。

另一方面，如果採用一擁而上的開發方法，那麼原有200人的隊伍仍然不足以開發真正的大型系統。例如，考慮OS/360項目。在頂峰時，有超過1000人在爲它工作——程式師、文檔編制人員、操作人員、職員、秘書、管理人員、支援小組等等。從1963年到1966年，設計、編碼和文檔工作花費了大約5000人年。如果人月可以等量置換的話，我們所假設的200人隊伍需要25年的時間，才能使產品達到現有的水準。

這就是小型、精幹隊伍概念上的問題：對於真正意義上的大型系統，它太慢了。設想OS/360的工作由一個小型、精幹的團隊來解決。譬如10人隊伍。作爲一個尺度，假設他們都非常厲害，比一般的編程人員在編程和文檔方面的生產率高7倍。假定OS/360原有開發人員是一些平庸的編程人員（這與實際的情況相差很遠）。同樣，假設另一個生產率的改進因數提高了7倍，因爲較小的隊伍所需較少的溝通和交流。那麼，5000/（10x7x7）= 10，他們需要10年來完成5000人年的工作。一個產品在最初設計的10年後才出現，還有人會對它感興趣嗎？或者它是否會隨著軟體發展技術的快速進步，而顯得過時呢？

這種進退兩難的境地是非常殘酷的。對於效率和概念的完整性來說，最好由少數幹練的人員來設計和開發，而對於大型系統，則需要大量的人手，以使產品能在時間上滿足要求。如何調和這兩方面的矛盾呢？

## Mills的建議

Harlan Mills的提議提供了一個嶄新的、創造性的解決方案[2,3]。Mills建議大型專案的每一個部分由一個團隊解決，但是該隊伍以類似外科手術的方式組建，而並非一擁而上。也就是說，同每個成員截取問題某個部分的做法相反，由一個人來進行問題的分解，其他人給予他所需要的支援，以提高效率和生產力。

簡單考慮一下，如果上述概念能夠實施，似乎它可以滿足迫切性的需要。很少的人員被包含在設計和開發中，其他許多人來進行工作的支援。它是否可行呢？誰是編程隊伍中的麻醉醫生和護士，工作如何劃分？讓我們繼續使用醫生的比喻：如果考慮所有可能想到的工作，這樣的隊伍應該如何運作。

外科醫生。Mills稱之爲首席程式師。他親自定義功能和性能技術說明書，設計程式，編制源代碼，測試以及書寫技術文檔。他使用例如PL/I的結構化編程語言，擁有對電腦系統的訪問能力；該電腦系統不僅僅能進行測試，還存儲程式的各種版本，以允許簡單的檔更新，並對他的文檔提供文本編輯能力。首席程式師需要極高的天分、十年的經驗和應用數學、業務資料處理或其他方面的大量系統和應用知識。

副手。他是外科醫生的後備，能完成任何一部分工作，但是相對具有較少的經驗。他的主要作用是作爲設計的思考者、討論者和評估人員。外科醫生試圖和他溝通設計，但不受到他建議的限制。副手經常在與其他團隊的功能和介面討論中代表自己的小組。他需要詳細瞭解所有的代碼，研究設計策略的備選方案。顯然，他充當外科醫生的保險機制。他甚至可能編制代碼，但針對代碼的任何部分，不承擔具體的開發職責。

管理員。外科醫生是老闆，他必須在人員、加薪等方面具有決定權，但他決不能在這些事務上浪費任何時間。因而，他需要一個控制財務、人員、工作地點安排和機器的專業管理人員，該管理員充當與組織中其他管理機構的介面。Baker建議僅在專案具有法律、合同、報表和財務方面的需求時，管理員才具有全職責任。否則，一個管理員可以爲兩個團隊服務。

編輯。外科醫生負責產生文檔——出於最大清晰度的考慮，他必須書寫文檔。對內部描述和外部描述都是如此。而編輯根據外科醫生的草稿或者口述的手稿，進行分析和重新組織，提供各種參考資訊和書目，對多個版本進行維護以及監督文檔生成的機制。

兩個秘書。管理員和編輯每個人需要一個秘書。管理員的秘書負責專案的協作一致和非產品檔。

程式職員。他負責維護編程產品庫中所有團隊的技術記錄。該職員接受秘書性質的培訓，承擔機器碼檔和可讀文件的相關管理責任。

所有的電腦輸入彙集到這個職員處。如果需要，他會對它們進行記錄或者標識。輸出列表會提交給程式職員，由他進行歸檔和編制索引。另外，他負責將任何模型的最新運行情況記錄在狀態日誌中，而所有以前的結果則按時間順序進行歸檔保存。

> Mills概念的真正關鍵是“從個人藝術到公共實踐”的編程觀念轉換。它向所有的團隊成員展現了所有電腦的運作和產物，並將所有的程式和資料看作是團隊的所有物，而非私人財產。

程式職員的專業化分工，使程式師從書記的雜事中解放出來，同時還可以對那些雜事進行系統整理，確保了它們的品質，並強化了團隊最有價值的財富——工作產品。上述概念顯然考慮的是批次處理程式。當使用互動式終端，特別是在沒有紙張輸出的情況下，程式職員的職責並未消失，只是有所更改。他會記錄小組程式和私有工作拷貝之間的更新，依然控制所有程式的運行，並使用自己的互動式工具來控制產品逐步增長的完整性和有效性。

工具維護人員。現在已經有很多檔編輯、文本編輯和互動式調試等工具，因此團隊很少再需要自己的機器和機器操作人員。但是這些工具使用起來必須毫無疑問地令人滿意，而且需要具備較高的可靠性。外科醫生則是這些工具、服務可用性的唯一評判人員。他需要一個工具維護人員，保證所有基本服務的可靠性，以及承擔團隊成員所需要的特殊工具（特別是互動式電

腦服務）的構建、維護和升級責任。即使已經擁有非常卓越的、可靠的集中式服務，每個團隊仍然要有自己的工具人員。因爲他的工作是檢查他的外科醫生所需要的工具。工具維護人員常常要開發一些實用程式、編制具有目錄的過程庫以及巨集庫。

測試人員。外科醫生需要大量合適的測試用例，用來對他所編寫的工作片段，以及對整個工作進行測試。因此，測試人員既是爲他的各個功能設計系統測試用例的對頭，同時也是爲他的日常調試設計測試資料的助手。他還負責計畫測試的步驟和爲測試搭建測試平臺。

語言專家。隨著Algol語言的出現，人們開始認識到大多數電腦專案中，總有一兩個樂於掌握複雜編程語言的人。這些專家非常有幫助，很快大家會向他諮詢。這些天才不同于外科醫生，外科醫生主要是系統設計者以及考慮系統的整體表現。而語言專家則尋找一種簡潔、有效的使用語言的方法來解決複雜、晦澀或者棘手的問題。他通常需要對技術進行一些研究（兩到三天）。通常一個語言專家可以爲兩個到三個外科醫生服務。

以上就是如何參照外科手術隊伍，以及如何對10人的編程隊伍進行專業化的角色分工。

## 如何運作

文中定義的開發團隊在很多方面滿足了迫切性的需要。十個人，其中七個專業人士在解決問題，而系統是一個人或者最多兩個人思考的產物，因此客觀上達到了概念的一致性。

要特別注意傳統的兩人隊伍與外科醫生——副手隊伍架構之間的區別。首先，傳統的團隊將工作進行劃分，每人負責一部分工作的設計和實現。在外科手術團隊中，外科醫生和副手都瞭解所有的設計和全部的代碼。這節省了空間分配、磁片訪問等的勞動量，同時也確保了工作概念上的完整性。

第二，在傳統的隊伍中大家是平等的，出現觀點的差異時，不可避免地需要討論和進行相互的妥協和讓步。由於工作和資源的分解，不同的意見會造成策略和介面上的不一致，例如誰的空間會被用作緩衝區，然而最終它們必須整合在一起。而在外科手術團隊中，不存在利益的差別，觀點的不一致由外科醫生單方面來統一。這兩種團隊組建上的差異——對問題不進行分解和上下級的關係——使外科手術隊伍可以達到客觀的一致性。

另外，團隊中剩餘人員職能的專業化分工是高效的關鍵，它使成員之間採用非常簡單的交流模式成爲可能。

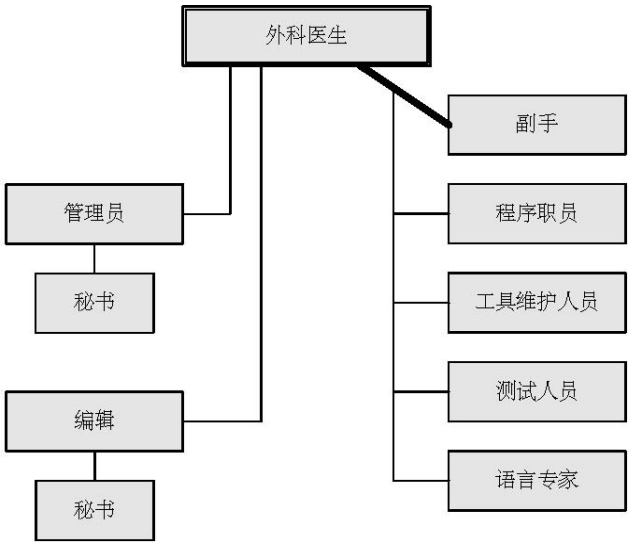



圖3.1：10人程式開發隊伍的溝通模式

Baker的文章[3]提出了專一的、小規模的測試隊伍。在那種情況下，它能按照所預期的進行運作，並具有良好的效果。

## 團隊的擴建

就目前情況而言，還不錯。然而，現在所面臨的問題是如何完成5000人年的項目，而不是20或30人年規模的系統。如果整個工作能控制在範圍之內，10人的團隊無論如何組織，總是比較高效的。但是，當我們需要面對幾百人參與的大型任務時，如何應用外科手術團隊的概念呢？

擴建過程的成功依賴於這樣一個事實，即每個部分的概念完整性得到了徹底的提高——決定設計的人員是原來的七分之一或更少。所以，可以讓200人去解決問題，而僅僅需要協調20個人，即那些“外科醫生”的思路。

對於協調的問題，還是需要使用分解的技術，這在後續的章節中會繼續進行討論。在這裏，可以認爲整個系統必須具備概念上的完整性，要有一個系統結構師從上至下地進行所有的設計。要使工作易於管理，必須清晰地劃分體系結構設計和實現之間的界線，系統結構師必須一絲不苟地專注於體系結構。總的說來，上述的角色分工和技術是可行的，在實際工作中，具有非常高的效率。

# 貴族專制、民主政治和系統設計（***Aristocracy,***

# *Democracy, and System Design*）

大教堂是藝術史上無與倫比的成就。它的原則既不乏味也不混亂□□真正達到了風格上的極致，完成這件作品的藝術家們，完全領會和吸收了以往的成功經驗，也完全掌握了他們那個時代的技術，而且在應用的時候做到了恰如其分，絕不輕率，也絕不花哨。

正是Jean d ‘Orbais構思了建築的整體設計，這個設計得到了後繼者的認同，至少在本質上如此。這也是這個建築如此和諧統一的原因之一。

- 蘭斯大教堂指南[1]

*This great church is an incomparable work of art. There is neither aridity nor confusion in the tenets it sets forth. . . . It is the zenith of a style, the work of artists who had understood and assimilated all their predecessors ’ successes, in complete possession of the techniques of their times, but using them without indiscreet display nor gratuitous feats of skill.*

*It was Jean d ‘Orbais who undoubtedly conceived the general plan of the building, a plan which was respected, at least in its essential elements, by his successors. This is one of the reasons for the extreme coherence and unity of the edifice.*

*- REIMS CATHEDRAL GUIDEBOOK*[1]

## 概念一致性

絕大多數歐洲的大教堂中，由不同時代、不同建築師所建造的各個部分之間，在設計或結構風格上都存在著許多差異。後來的建築師總是試圖在原有建築師的基礎上有所“提高”，以反映他們在設計風格和個人品味上的改變。所以，在雄偉的哥特式的教堂上，依附著祥和的諾曼第風格十字架，它在顯示上帝榮耀的同時，展示了同樣屬於建築師的驕傲。

與之對應的是，法國城市蘭斯（Reims）在建築風格上的一致性和上面所說的大教堂形成了鮮明的對比。設計的一致性和那些獨到之處一樣，同樣讓人們讚歎和喜悅。如同旅遊指南所述，風格的一致和完整性來自8代擁有自我約束和犧牲精神的建築師們，他們每一個人犧牲了自己的一些創意，以獲得純粹的設計。同樣，這不僅顯示了上帝的榮耀，同時也體現了他拯救那些沉醉在自我驕傲中的人們的力量。

對於電腦系統而言，儘管它們通常沒有花費幾個世紀的時間來構建，但絕大多數系統體現出的概念差異和不一致性遠遠超過歐洲的大教堂。這通常並不是因爲它由不同的設計師們開

發，而是由於設計被分成了由若干人完成的若干任務。

我主張在系統設計中，概念完整性應該是最重要的考慮因素。也就是說爲了反映一系列連貫的設計思路，寧可省略一些不規則的特性和改進，也不提倡獨立和無法整合的系統，哪怕它們其實包含著許多很好的設計。在本章和以下的兩章裏，我們將解釋在編程系統設計中，這個主題的重要性。

如何得到概念的完整性？

這樣的觀點是否要有一位傑出的精英，或者說是結構設計師的貴族專制，和一群創造性天分和構思被壓制的平民編程實現人員？

如何避免結構設計師產出無法實現、或者是代價高昂的技術規格說明，使大家陷入困境？

如何才能與實現人員就技術說明的瑣碎細節充分溝通，以確保設計被正確地理解，並精確地整合到產品中？

## 獲得概念的完整性

編程系統（軟體）的目的是使電腦更加容易使用。爲了做到這一點，電腦裝備了語言和各種工具，這些工具實際上也是被調用的程式，受到編程語言的控制。使用這些工具是有代價的：軟體外部描述的規模大小是電腦系統本身說明的十倍。用戶會發現尋找一個特定功能是很容易的，但相應卻有太多的選擇，要記住太多的選項和格式。

只有當這些功能說明節約下來的時間，比用在學習、記憶和搜索手冊上的時間要少時，易用性才會得到提高。現代編程系統節省的時間的確超過了花費的時間，但是近年來，隨著越來越多的功能添加，收益和成本的比率正逐漸地減少。而IBM 650使用的容易程度總縈繞在我的腦際，即使該系統沒有使用彙編和任何其他的軟體。

由於目標是易用性，功能與理解上複雜程度的比值才是系統設計的最終測試標準。單是功能本身或者易於使用都無法成爲一個好的設計評判標準。

然而這一點被廣泛地誤解了。作業系統OS/360由於其複雜強大的功能被它的開發者廣爲推崇。功能，而非簡潔，總是被用來衡量設計人員工作的出色程度。而另一方面，PDP-10的時分系統由於它的簡潔和概念的精幹被建造它的人員所稱道。當然，無論使用任何測量標準，後者的功能與OS/360都不在一個數量級。但是，一旦以易用性作爲衡量標準，單獨的功能和易於使用都是不均衡的，都只達到了真正目標的一半。

對於給定級別的功能，能用最簡潔和直接的方式來指明事情的系統是最好的。只有簡潔（simplicity）是不夠的，Mooers的TRAC語言和Algol 68用很多獨特的基本概念達到了所需的簡潔特性，但它們並不直白（straightforward）。要表達一件待完成的事情，常常需要對基本元素進行意料不到的複雜組合。而且，僅僅瞭解基本要素和組合規則還不夠，

還需要學習晦澀的用法，以及在實際工作中如何進行組合。簡潔和直白來自概念的完整性。每個部分必須反映相同的原理、原則和一致的折衷機制。在語法上，每個部分應使用相同的技巧；在語義上，應具有同樣的相似性。因此，易用性實際上需要設計的一致性和概念上的完整性。

## 貴族專制統治和民主政治

概念的完整性要求設計必須由一個人，或者非常少數互有默契的人員來實現。

而進度壓力卻要求很多人員來開發系統。有兩種方法可以解決這種矛盾。第一種是仔細地區分設計方法和具體實現。第二種是前一章節中所討論的、一種嶄新的組建編程開發團隊的方法。

對於非常大型的專案，將設計方法、體系結構方面的工作與具體實現相分離是獲得概念完整性的強有力方法。我親眼目睹了它在IBM的Stretch電腦和360電腦產品線上的巨大成功。但同時我也看到了這種方法在360作業系統的開發中，由於缺乏廣泛應用所遭受的失敗。

系統的體系結構（architecture）指的是完整和詳細的用戶介面說明。對於電腦，它是編程手冊；對於編譯器，它是語言手冊；對於控制程式，它是語言和函數調用手冊；對於整個系統，它是用戶要完成自己全部工作所需參考的手冊的集合[2]。

因此，系統的結構師，如同建築的結構師一樣，是用戶的代理人。結構師的工作，是運用專業技術知識來支援用戶的真正利益，而不是維護銷售人員所鼓吹的利益。

體系結構同實現必須仔細地區分開來。如同Blaauw所說的，“體系結構陳述的是發生了什麼，而實現描述的是如何實現[3]。”他舉了一個簡單的例子——時鐘。它的結構包括表面、指標和上發條的旋鈕。當一個小孩知道了時鐘的外表結構，他很容易從手錶或者教堂上的時鐘辨認時間。而時鐘的實現，描述了表殼中的事物——很多種動力提供裝置中的一種，以及眾多控制精度方案的一種。

例如，在System/360中，一個電腦的結構可以用9種不同的模型來實現；而單個實現——Model 30的資料流程、記憶體和微代碼實現——可以用於4種不同的體系結構：System/360電腦、擁有224個獨立邏輯子通道的複雜通道、選擇通道以及1401電腦[4]。

同樣的劃分方法也適用於編程系統。例如，美國的Fortran IV標準，是多種編譯器所遵循的體系結構標準。該體系結構下有多種可能的實現：以文本爲核心、以編譯器爲核心、快速編譯和優化以及側重語法的實現。相類似的，任何組合語言和任務控制語言都允許有多種編譯器或調度程式的實現。

現在讓我們來處理具有濃厚感情色彩的問題——貴族統治和民主政治。結構師難道不是新貴？他們一些智力精英，專門來告訴可憐的實現人員如何工作？是否所有的創造性活動被那些精英單獨佔有，實現人員僅僅是機器中的齒輪？難道不能遵循民主的理論，從所有的員工中搜集好的創意，以得到更好的產品，而不是將技術說明工作僅限定於少數人？

最後一個問題是最簡單的。我當然不認爲只有結構師才有好的創意。新的概念經常來自實現者或者用戶。然而，我一直試圖表達，並且我所有的經驗使我確信，系統的概念完整性決定了使用的容易程度。不能與系統基本概念進行整合的良好想法和特色，最好放到一邊，不予考慮。如果出現了很多非常重要但不相容的構想，就應該拋棄原來的設計，對不同基本概念進行合併，在合併後的系統上重新開始。

至於貴族專制統治的問題，必須回答 “是”或者“否”。就必須只能存在少數的結構師而言，答案是肯定的，他們的工作產物的生命週期比那些實現人員的產物要長，並且結構師一直處在解決用戶問題，實現用戶利益的核心地位。如果要得到系統概念上的完整性，那麼必須控制這些概念。這實際上是一種無需任何歉意的貴族專制統治。

第二個問題的答案是否定的，因爲外部技術說明的編制工作並是比具體設計實現更富有創造性，它只是一項性質不同的創造工作而已。在給定體系結構下的設計實現，同樣需要同編制技術說明一樣的創造性、同樣新的思路和卓越的才華。實際上，產品的成本性能比很大程度上依靠實現人員，就如同易用性很大程度上依賴結構師一樣。

有很多行業和領域中的案例讓人相信紀律和規則對行業是有益的。實際上，如同某藝術家的格言所述，“沒有規矩，不成方圓。”最差的建築往往是那些預算遠遠超過起始目標的專案。巴赫曾被要求每週創作一篇形式嚴格的歌劇，但這似乎並沒有被壓制他的創造性。並且，我確信如果Stretch電腦有更嚴格的限制，那麼該電腦會擁有更好的體系結構。就我個人意見而言，System/360 Model 30預算上的限制，完全獲益於Model 75的體系結構。

類似的，我觀察到外部的體系結構規定實際上是增強，而不是限制實現小組的創造性。一旦他們將注意力集中在沒有人解決過的問題上，創意就開始奔湧而出。在毫無限制的實現小組中，在進行結構上的決策時，會出現大量的想法和爭議，對具體實現的關注反而會比較少[5]。

我曾見過很多次這樣的結果，R.W.Conway也證實了這一點。他在Cornell的小組曾編制PL/I語言的編譯器。他說：“最後我們的編譯器決定支持不經過改進和增強的語言，因爲關於語言的爭議已經耗費了我們所有的時間和精力。[6]”

## 在等待時，實現人員應該做什麼？

幾百萬元的失誤是非常令人慚愧的經驗，但同時也是讓人記憶深刻的教訓。當年我們計畫和組織編寫OS/360外部技術說明的那個夜晚，常常重現在我的腦海。我和體系結構經理、程式實現經理一起制訂計畫進度，並確認責任分工。

體系結構經理擁有10個很好的員工，他聲稱他們可以書寫規格說明，並出色地完成任務。該任務需要10個月，比所允許的進度多了3個月。

程式實現經理有150人。他認爲在體系結構隊伍的協助下，他們可以準備技術說明，並且能按照時間進度，完成高品質的、切合實際的說明。此外，如果光是由體系結構的團隊承擔該工作，他的150人只能坐在那兒幹等10個月，無所事事。

對此，體系結構的經理的反應是，如果讓程式實現隊伍來負責該工作，結果不會按時完成，仍將推遲3個月，而且品質更加低劣。我將工作分派給了程式實現隊伍，其結果也確實如此。體系結構經理的兩個結論都得到了證實。另外，概念完整性的缺乏導致系統開發和修改上要付出更昂貴的代價，我估計至少增加了一年的調試時間。

當然，很多因素造成了那個錯誤的決策，但決定性因素是時間進度和讓150名編程人員進行工作的願望。而它也正是我想強調的致命危險。

當建議由體系結構的團隊來編寫電腦和編程系統的所有外部技術說明時，編程人員提出了三個反對意見：

該說明中的功能過於繁多，而對實際情況中的成本考慮比較少

結構師獲得了所有創造發明的快樂，剝奪了實現人員的創造力

當體系結構的隊伍緩慢工作時，很多實現人員只能空閒地坐著等待

這些問題中的第一個確實是一項危險，在下一章中我們將討論這個問題，但其他的兩個問題都是一些簡單而純粹的誤解。正如我們前面所看到的，實現同樣是一項高級別的創造性活動。具體實現中創造和發明的機會，並不會因爲指定了外部技術說明而大爲減少，相反創造性活動會因爲規範化而得到增強，整個產品也一樣。

最後一個反對意見是時間順序和階段性上的問題。問題的簡要回答是，在說明完成的時候，才雇用編程實現人員。這也正是在搭建一座建築時所採用的方法。

在電腦這個行業中，節奏非常快，而且常常想盡可能地壓縮時間進度，那麼技術說明和開發實現能有多少重疊呢？

如同Blaauw所指出的，整個創造性活動包括了三個獨立的階段：體系結構（architecture）、設計實現（implementation）、物理實現（realization）。在實際情況中，它們往往可以同時開始和併發地進行。

例如，在電腦的設計中，一旦設計實現人員有了對手冊的模糊設想，對技術有了相對清晰的構思以及擁有了定義良好的成本和目標時，工作就可以開始了。他可以開始設計資料流程、控制序列、大體的系統劃分等等。同時，還需要選用工具以及進行相應的調整，特別是記錄存檔系統和設計自動化系統。

同時，在物理實現的級別，電路、板卡、線纜、機箱、電源和記憶體必須分別設計、細化和編制文檔。這項工作與體系結構及設計實現並行進行。。

在編程系統的開發中，這個原理同樣適用。在外部說明完成之前，設計實現人員有很多的事情可以做。只要有一些最終將併入外部說明的系統功能雛形，他就可以開始了。首先，必須設定良好定義的時間和空間目標，瞭解產品運行的平臺配置。接著，他可以開始設計模組的邊界、表結構、演算法以及所有的工具。另外，還需要花費一些時間和體系結構師溝通。

同時，在物理實現的級別，也有很多可以著手的工作。編程也是一項技術，如果是新型的機器，則在庫的調整、系統管理以及搜索和排序演算法上，有許多事情需要處理[7]。

概念的完整性的確要求系統只反映唯一的設計理念，用戶所見的技術說明來自少數人的思想。實際工作被劃分成體系結構、設計實現和物理實現，但這並不意味著該開發模式下的系統需要更長的時間來創建。經驗顯示恰恰相反，整個系統將會開發得更快，所需要的測試時間將更少。同工作的水準分割相比，垂直劃分從根本上大大減少了勞動量，結果是使交流徹底地簡化，概念完整性得到大幅提高。

# 畫蛇添足（*The Second-System Effect*）

聚沙成塔，集腋成裘。

- 奧維德

*Adde parvum parvo magnus acervus erit.*

*[Add little to little and there will be a big pile.]*

*- OVID*

如果將制訂功能規格說明的責任從開發快速、成本低廉的產品的責任中分離出來，那麼有什麼樣的準則和機制來約束結構師的創造性熱情呢？

基本回答是結構師和建築人員之間徹底、仔細和諧的交流。另外，還有很多值得關注的、更細緻的答案。

## 結構師的交互準則和機制

建築行業的結構設計師使用估算技術來編制預算，該估算技術會由後續的承包商報價來驗證和修正。承包商的報價總會超過預算。接下來，設計師會重新改進他的預算或修訂設計，調整到下一期工程。他也可能會向承包商建議，使用更加便宜的方法來實現設計。

類似的過程也支配著電腦系統和電腦編程系統的結構師。相比之下，他有能在設計早期從

承包商處得到報價的優勢，幾乎是只要他詢問，就能得到答案。他的不利之處常常是只有一個承包商，後者可以增高或降低前者的估計，來反映對設計的好惡。實際情況中，儘早交流和持續溝通能使結構師有較好的成本意識，以及使開發人員獲得對設計的信心，並且不會混淆各自的責任分工。

面對估算過高的難題，結構師有兩個選擇：削減設計或者建議成本更低的實現方法——挑戰估算的結果。後者是固有的主觀感性反應。此時，結構師是在向開發人員的做事方式提出挑戰。想要成功，結構師必須

牢記是開發人員承擔創造性和發明性的實現責任，所以結構師只能建議，而不能支配；

時刻準備著爲所指定的說明建議一種實現的方法，同樣準備接受其他任何能達到目標的方法；

對上述的建議保持低調和平靜；

準備放棄堅持所作的改進建議；

一般開發人員會反對體系結構上的修改建議。通常他是對的　　當正在實現產品時，某些特性的修改會造成意料不到的成本開銷。

## 自律　　開發第二個系統所帶來的後果

在開發第一個系統時，結構師傾向於精煉和簡潔。他知道自己對正在進行的任務不夠瞭解，所以他會謹慎仔細地工作。

在設計第一個專案時，他會面對不斷產生的裝飾和潤色功能。這些功能都被擱置在一邊，作爲“下一個”專案的內容。第一個專案遲早會結束，而此時的結構師，對這類系統充滿了十足的信心，熟練掌握了相應的知識，並且時刻準備開發第二個系統。

第二個系統是設計師們所設計的最危險的系統。而當他著手第三個或第四個系統時，先前的經驗會相互驗證，得到此類系統通用特性的判斷，而且系統之間的差異會幫助他識別出經驗中不夠通用的部分。

一種普遍傾向是過分地設計第二個系統，向系統添加很多修飾功能和想法，它們曾在第一個系統中被小心謹慎地推遲了。結果如同Ovid所述，是一個“大餡餅”。例如，後來被嵌入到7090的IBM 709系統，709是對非常成功和簡潔的704系統進行升級的二次開發專案。709的操作集合被設計得如此豐富和充沛，以至於只有一半操作被常規使用。

讓我們來看看更嚴重的例子——Stretch電腦的結構（architecture）、設計實現（implementation）、甚至物理實現（realization），它是很多人被壓抑創造力的宣洩出口。如果Strachey在評審時所述：

對於Stretch系統，我的印象是從某種角度而言，它是一個產品線的終結。如同早期的電腦程式一樣，它極富有創造性，極端複雜，非常高效。但不知爲什麼，同時也感覺到粗糙、浪費、不優雅，以及讓人覺得必定存在某種更好的方法[1]。

作業系統360對於大多數設計者來說，是第二個系統。它的設計小組成員來自1410-7010磁片作業系統、Stretch作業系統、Mercury即時系統專案和7090的IBSYS。幾乎沒有人有兩個以上早期作業系統的經驗[2]。因此，OS/360是典型的第二次開發（second-system effect）的例子，是軟體行業的Stretch系統。Strachey的讚譽和批評可以毫無更改地應用在其中。

例如，OS/360開發了26位元組的常駐日期翻轉常式來正確地處理閏年的12月31日的問題，其實它完全可以留給操作員來完成。

開發第二個系統所引起的後果（second-system effect）與純粹的功能修飾和增強明顯不同，也就是說存在對某些技術進行細化、精煉的趨勢。由於基本系統設想發生了變化，這些技術已經顯得落後。OS/360中有很多這樣的例子。

例如，鏈結編輯器的設計，它用來對分別編譯後的程式進行裝載，解決它們之間的交叉引用。除了這些基本的功能，它還支援程式的覆蓋（overlay）。這是所有實現的覆蓋服務程式中最好的一種。它允許鏈結時在外部完成覆蓋結構，而無需在源代碼中進行設計。它還允許在運行時刻改變覆蓋，而不必重新編譯。它配備了豐富的實用選項和各種功能。某種意義上，它是若干年靜態覆蓋技術開發的頂峰。

然而，它同時也是最後和最優秀的恐龍，因爲它屬於一個基本運行方式爲多道程序，以動態內核分配爲基礎的系統，這直接與靜態覆蓋的概念相衝突。如果我們把投入在覆蓋管理上的工作量，用在提高動態內核分配和動態交叉引用的性能上，那麼系統將會運行得多麼好啊！

另外，鏈結編輯器需要如此大的空間，而且它本身就包含了很多程式庫，以至於即使在不使用覆蓋管理功能，僅僅使用鏈結功能的時候，它也比絕大多數系統的編譯程序還要慢。具有諷刺意味的是，鏈結程式的目的是爲了避免重新編譯。這種情況就像一個挺著大肚子的節食者一樣，直到系統的思想已經十分優越時，才開始對原有技術進行細化和精煉。

TESTRAN調試程式是這個趨勢的另一個例子。它在批調試程式中是出類拔萃的，配備了真正優雅的快照和記憶體資訊轉儲功能。它使用了控制段的概念和卓越的生成技術，從而不需要重新編譯或解釋，就能實現選擇性跟蹤和快照。這種709共用作業系統[3]中魔術般的概念得到了廣泛的使用。

但同時，整個無需重編譯的批調試概念變得落伍了。使用語言解釋器和增量編譯器的互動式計算系統，向它提出了最根本的挑戰。即使是在批次處理系統中，快速編譯/慢速執行編譯器的出現，也使源代碼級別調試和快照技術成爲優先選擇的技術。如果在構建和優化互動式和快速編譯程序之前，就已經著手TESTRAN的開發，那麼系統將是多麼的優秀啊！

還有另外一個例子是調度程式。OS/360的調度程式是非常傑出的，它提供了管理固定批作業的傑出功能。從真正意義上講，該調度程式是作爲1410－7010磁片作業系統後續的二次系統，經過了精煉、改進和增強。它是除了輸入－輸出以外的非多道程序批次處理系統，是一種主要

用於商業應用的系統。但是，它對OS/360的遠端任務項、多道程序、永久駐留互動式子系統，幾乎完全沒有影響和幫助。實際上，OS/360調度程式的設計使它們變得更加困難。

結構師如何避免畫蛇添足——開發第二個系統所引起的後果（second-system effect）？是的，他無法跳過二次系統。但他可以有意識關注那些系統的特殊危險，運用特別的自我約束準則，來避免那些功能上的修飾；根據系統基本理念及目的變更，捨棄一些功能。

一個可以開闊結構師眼界的準則是爲每個小功能分配一個值：每次改進，功能x不超過m位元組的記憶體和n微秒。這些值會在一開始作爲決策的嚮導，在物理實現期間充當指南和對所有人的警示。

專案經理如何避免畫蛇添足（second-system effect）？他必須堅持至少擁有兩個系統以上開發經驗結構師的決定。同時，保持對特殊誘惑的警覺，他可以不斷提出正確的問題，確保原則上的概念和目標在詳細設計中得到完整的體現。

# 貫徹執行（***Passing the Word***）

他只是坐在那裏，嘴裏說：“做這個！做那個！”當然，什麼都不會發生，光說不做是沒有用的。

- 哈裏·杜魯門，關於總統的權力[1]

*He'll sit here and he'll say, "Do this!  Do that!"  And nothing will happen.*

*- HARRY S. TRUMAN, ON PRESIDENTIAL POWER*[1]

假設一個專案經理已經擁有行事規範的結構師和許多編程實現人員，那麼他如何確保每個人聽從、理解並實現結構師的決策？對於一個由1000人開發的系統，一個10個結構師的小組如何保持系統概念上的完整性？在System/360硬體設計工作中，我們摸索出來一套實現上述目標的方法，它們對於軟體專案同樣適用。

## 文檔化的規格說明——手冊

手冊、或者書面規格說明，是一個非常必要的工具，儘管光有文檔是不夠的。手冊是產品的外部規格說明，它描述和規定了用戶所見的每一個細節；同樣的，它也是結構師主要的工作產物。

隨著用戶和實現人員回饋的增加，規格說明中難以使用和難以構建實現的地方不斷被指出，規格說明也不斷地被重複準備和修改。然而對實現人員而言，修改的階段化是很重要的——在進度表上應該有帶日期的版本資訊。

手冊不但要描述包括所有介面在內的用戶可見的一切，它同時還要避免描述用戶看不見的事物。後者是編程實現人員的工作範疇，而實現人員的設計和創造是不應該被限制的。體系結構設計人員必須爲自己描述的任何特性準備一種實現方法，但是他不應該試圖支配具體的實現過程。

規格說明的風格必須清晰、完整和準確。用戶常常會單獨提到某個定義，所以每條說明都必須重複所有的基本要素，所以所有文字都要相互一致。這往往使手冊讀起來枯燥乏味，但是精確比生動更加重要。

System/360 Principles of Operation的一致完整性來自僅有兩名作者的事實：Gerry Blaauw和Andris Padegs。思路是大約十個人的想法，但如果想保持文字和產品之間的一致性，則必須由一個或兩個人來完成將其結論轉換成書面規格說明的工作。而且，將定義書寫成文字，必須對很多原先並不是非常重要的問題進行判斷，並得出結論。例如，System/360需要決定在每次操作後，如何設置返回的條件碼。其實，對於在整個設計中，保證這些看似瑣碎的問題處理原則上的一致性，決不是一件無關緊要的事情。

我想我所見過的最好的一份手冊是System360 Principles of Operation的附錄。它精確仔細地規定了System/360相容性的限制。它定義了相容性，描述了將達到的目標，列舉了很多外部顯示的各個部分：源於某個模型與其他模型差異，帶來變化的部分和保持不變的部分；或者是某個給定模型的拷貝不同於其他拷貝的地方；甚至是工程上的變更引起拷貝自身上的差異。而這正是一個規格說明作者所應該追求的精確程度，他必須在仔細定義規定什麼的同時，定義未規定什麼。

## 形式化定義

英語或者其他任何的人類語言，從根本上說，都不是一種能精確表達上述定義的手段。因此，手冊的作者必須注意自己的思路和語言，達到所需要的精確程度。一種頗具吸引力的作法是對上述定義使用形式化標記方法。畢竟，精確度是我們需要的東西，這也正是形式化標記方法存在的理由和原因。

讓我們來看一看形式化定義的優點和缺點。如文中所示，形式化定義是精確的，它們傾向於更加完整；差異得更加明顯，可以更快地完成。但是形式化定義的缺點是不易理解。記敍性文字則可以顯示結構性的原則，描述階段上或層次上的結構，以及提供例子。它可以很容易地表達異常和強調對比的關係，最重要的是，它可以解釋原因。在表達的精確和簡明性上，目前所提出的形式化定義，具有了令人驚異的效果，增強了我們進行準確表達的信心。但是，它還需要記敍性文字的輔助，才能使內容易於領會和講授。出於這些原因，我想將來的規格說明同

時包括形式化和記敍性定義兩種方式。

一句古老的格言警告說：“決不要攜帶兩個時鐘出海，帶一個或三個。”同樣的原則也適用於形式化和記敍性定義。如果同時具有兩種方式，則必須以一種作爲標準，另一種作爲輔助描述，並照此明確地進行劃分。它們都可以作爲表達的標準，例如，Algol 68採用形式化定義作爲標準，記敍性文字作爲輔助。PL/I使用記敍性定義作爲主要方式，形式化定義用作輔助表述。System/360也將記敍性文字用作標準，以及形式化定義用作派生的論述。

很多工具可以用於形式化定義，例如巴科斯範式在語言定義中很常用，它在書本中有詳細的描述[2]。PL/I的形式化定義使用了抽象語法的新概念，該概念有很確切的解釋[3]。Iverson的APL曾用來描述機器，突出的應用是IBM 7090[4]和System/360[5]。

Bell和Newell建議了能同時描述配置和機器結構的新標注方法，並且在許多機型的應用上得以體現，如DEC PDP-8[6]、7090[8]、System/360。

在規定系統外部功能的同時，幾乎所有的形式化定義均會用來描述和表達硬體系統或軟體系統的某個設計實現。語法和規則的表達可以不需要具體的設計實現，但是特定的語義和意義通常會通過一段實現該功能的程式來定義。理所當然，這是一種實現，不過它過多地限定了體系結構。所以必須特別指出形式化定義僅僅用於外部功能，說明它們是什麼。

如同前面所示，形式化定義可以是一種設計實現。反之，設計實現也可以作爲一種形式化定義的方法。當製造第一批相容性的電腦時，我們使用的正是上述技術：新的機器同原有的機器一致。如果手冊有一些模糊的地方？“問一問機器！”——設計一段程式來得到其行爲，新機器必須按照上述結果運行。

硬體或軟體系統的仿真裝置，可以按照相同的方式完整運用。它是一種實現，可以運行。因此，所有定義的問題可以通過測試來解決。

使用實現來作爲一種定義的方式有一些優點。首先，所有問題可以通過試驗清晰地得到答案，從來不需要爭辯和商討，回答是快捷迅速的。通過定義得出的答案，總是同所要求的一樣精確和正確。但是，相對於這些優點的，是一系列可怕的缺點。實現可能更加過度地規定了外部功能。例如，無效的語法通常會產生某些結果。在擁有錯誤控制的系統中，它通常僅僅導致某種“無效”的指示，而不會產生其他的東西。在無錯誤控制的系統中，會產生各種副作用，它們可能被程式師所使用。例如，當我們著手在System/360上模擬IBM 1401時，有30個不同的“古玩”——被認爲是無效操作的副作用——得到廣泛的應用，並被認爲是定義的一部分。作爲一種定義，實現體現了過多的內容：它不但描述了系統必須做什麼，同時還聲明了自己到底做了些什麼。

因此，當尖銳的問題被提及時，實現有時會給出未在計畫中的意外答案；這些答案中，真正的定義常常是粗糙的，因爲它們從來沒有被仔細考慮過。這些粗糙的功能在其他的設計實現中，往往是低效或者代價高昂的。例如，一些機器在乘法運算之後，將某些運算的垃圾遺留在被乘數寄存器中。該功能確切的特性，即保存運算垃圾，成爲了真正定義的一部分。然而，重複該細節可能會阻止某些快速乘法演算法的使用。

最後，關於實際使用標準是形式化描述還是敍述性文字這一點而言，使用實現作爲形式化定義特別容易引起混淆，特別是在程式化的仿真中。另外，當實現充當標準時，還必須防止對實現的任何修改。

## 直接整合

對軟體系統的體系結構師而言，存在一種更加可愛的方法來分發和強制定義。對於建立模組間接口語法，而非語義時，它特別有用。這項技術是設計被傳遞參數和共用記憶體的聲明，並要求編程實現在編譯時的一些操作（PL/I的宏或%INCLUDE）來包含這些聲明。另外，如果整個介面僅僅通過符號名稱進行引用，那麼需要修改聲明的時候，可以通過增加或插入新變數，或者重新編譯受影響的程式。這種方法不需要修改程式內容。

## 會議和大會

無需多說，會議是必要的。然而，數百人在場的大型磋商會議往往需要大規模和非常正式地召集。因此，我們把會議分成兩個級別：周例會和年度大會——這實際上是一種非常有效的方式。

周例會是每週半天的會議，由所有的結構師，加上硬體和軟體實現人員代表和市場計畫人員參與，由首席系統結構師主持。

會議中，任何人可以提出問題和修改意見，但是建議書通常是以書面形式，在會議之前分發。新問題通常會被討論一些時間。重點是創新，而不僅僅是結論。該小組試圖發現解決問題的新方法，然後少數解決方案會被傳遞給一個和幾個結構師，詳細地記錄到書面的變更建議說明書中。

接著會對詳細的變更建議做出決策。這會經歷幾個反復過程，實現人員和用戶會仔細地進行考慮，正面和負面的意見都會被很好地描述。如果達成了共識，非常好；如果沒有，則由首席結構師來決定。這需要花費時間，最終所發佈的結論是正式和果斷的。

周例會的決策會給出迅捷的結論，允許工作繼續進行。如果任何人對結果過於不高興，可以立刻訴諸于專案經理，但是這種情況非常少見。

這種會議的卓有成效是由於：

1．數月內，相同小組——結構師、用戶和實現人員——每週交流一次。因此，大家對專案相關的內容比較瞭解，不需要安排額外時間對人員進行培訓。

2．上述小組十分睿智和敏銳，深刻理解所面對的問題，並且與產品密切相關。沒有人是“顧問”的角色，每個人都要承擔義務。

3．當問題出現時，在界線的內部和外部同時尋求解決方案。

4．正式的書面建議集中了注意力，強制了決策的制訂，避免了會議草稿紀要方式的不一致。

5．清晰地授予首席結構師決策的權力，避免了妥協和拖延。

隨著時間的推移，一些決定沒有很好地貫徹，一些小事情並沒有被某個參與者真正地接受，其他決定造成了未曾遇到的問題。對於這些問題，有時周例會沒有重新考慮，慢慢地，很多小要求、公開問題或者不愉快會堆積起來。爲解決這些堆積起來的問題，我們會舉行年度大會，典型的年度大會會持續兩周。（如果由我重新安排，我會每六個月舉行一次。）

這些會議在手冊凍結的前夕召開。出席人員不僅僅包括體系結構小組和編程人員、實現人員的結構代表，同時包括編程經理、市場和實現人員，由System/360的專案經理主持。議程典型地包括大約200個條目，大多數條目的規模很小，它們列舉在會議室周圍的圖表上，每個不同的聲音都有機會得到表達。然後，會制訂出決策，加上出色的電腦化文本編輯工作（許多優秀員工的卓越的工作成果）。每天早晨，會議參與人員會在座位上發現更新了的手冊說明，記錄了前一天的各項決定。

這些“收穫的節日”不僅可以解決決策上的問題，而且使決策更容易被接受。每個人都在傾聽，每個人都在參與，每個人對複雜約束和決策之間的相互關係有了更透徹的理解。

## 多重實現

System/360的結構師具有兩個空前有利的條件：充足的工作時間，擁有與實現人員相同的策略影響力。充足時間來自新技術的開發日程；而多重實現的同時開發帶來了策略上的平等性。不同實現之間嚴格要求相互相容，這種必要性是強制規格說明的最佳代言人。

在大多數電腦專案中，機器和手冊之間往往會在某一天出現不一致，人們通常會忽略手冊。因爲與機器相比，手冊更容易改動，並且成本更低。然而，當存在多重實現時，情況就不是這樣。這時，如實地遵從手冊更新機器所造成的延遲和成本的消耗，比根據機器調整手冊要低。

在定義某編程語言的時候，上述概念可以卓有成效地得到應用。可以肯定的是，遲早會有很多編譯器或解釋器被推出，以滿足各種各樣的目標。如果起初至少有兩種以上的實現，那麼定義會更加整潔和規範。

## 電話日誌

隨著實現的推進，無論規格說明已經多麼精確，還是會出現無數結構理解和解釋方面的問題。顯然有很多問題需要文字澄清和解釋，還有一些僅僅是因爲理解不當。

顯然，對於存有疑問的實現人員，應鼓勵他們打電話詢問相應的結構師，而不是一邊自行猜測一邊工作，這是一項很基本的措施。他們還需要認識到的是，上述問題的答案必須是可以

告知每個人的權威性結論。

一種有用的機制是由結構師保存電話日誌。日誌中，他記錄了每一個問題和相應的回答。每週，對若干結構師的日誌進行合併，重新整理，並發佈給用戶和實現人員。這種機制很不正式，但非常快捷和易於理解。

## 產品測試

項目經理最好的朋友就是他每天要面對的敵人——獨立的產品測試機構/小組。該小組根據規格說明檢查機器和程式，充當麻煩的代言人，查明每一個可能的缺陷和相互矛盾的地方。每個開發機構都需要這樣一個獨立的技術監督部門，來保證其公正性。

在最後的分析中，用戶是獨立的監督人員。在殘酷的現實使用環境中，每個細微缺陷都將無從遁形。產品－測試小組則是顧客的代理人，專門尋找缺陷。不時地，細心的產品測試人員總會發現一些沒有貫徹執行、設計決策沒有正確理解或準確實現的地方。出於這方面的原因，設立測試小組是使設計決策得以貫徹執行的必要手段，同樣也是需要儘早著手，與設計同時實施的重要環節。

# 爲什麼巴比倫塔會失敗？（***Why Did the Tower of Babel Fail?***）

□□現在整個大地都採用一種語言，只包括爲數不多的單詞。在一次從東方往西方遷徙的過程中，人們發現了蘇美爾地區，並在那裏定居下來。接著他們奔相走告說：“來，讓我們製造磚塊，並把它們燒好。”於是，他們用磚塊代替石頭，用瀝青代替灰泥（建造房屋）。然後，他們又說：“來，讓我們建造一座帶有高塔的城市，這個塔將高達雲宵，也將讓我們聲名遠揚，同時，有了這個城市，我們就可以聚居在這裏，再也不會分散在廣闊的大地上了。”於是上帝決定下來看看人們建造的城市和高塔，看了以後，他說：“他們只是一個種族，使用一種的語言，如果他們一開始就能建造城市和高塔，那以後就沒有什麼難得倒他們了。來，讓我們下去，在他們的語言裏製造些混淆，讓他們相互之間不能聽懂。”這樣，上帝把人們分散到世界各地，於是他們不得不停止建造那座城市。（創世紀，*11:1-8*）

*Now the whole earth used only one language, with few words. On the occasion of a migration from the east, men discovered a plain in the land of Shinar, and settled there. Then they said to one another, "Come, let us make bricks, burning them well." So they used bricks for stone, and bitumen for mortar. Then they said, "Come, let us build ourselves a city with a tower whose top shall reach the heavens (thus making a name for ourselves), so that we may not be scattered all over the earth." Then the Lord came down to look at the city and tower which human beings had built. The Lord said, "They are just one people and they all have the same language. If this is what they can do as a beginning, then nothing that they resolve to do will be impossible for them. Come, let us go down, and there make*

*such a babble of their language that they will not understand one another's speech." Thus the Lord dispersed them from there all over the earth, so that they had to stop building the city. (Book of Genesis, 11:1-8).*

## 巴比倫塔的管理教訓

據《創世紀》記載，巴比倫塔是人類繼諾亞方舟之後的第二大工程壯舉，但巴比倫塔同時也是第一個徹底失敗的工程。

這個故事在很多方面和不同層次都是非常深刻和富有教育意義的。讓我們將它僅僅作爲純粹的工程項目，來看看有什麼值得學習的教訓。這個項目到底有多好的先決條件？他們是否有：

1．清晰的目標？是的，儘管幼稚得近乎不可能。而且，項目早在遇到這個基本的限制之前，就已經失敗了。

2．人力？非常充足。

3．材料？在美索不達米亞有著豐富的泥土和柏油瀝青。

4．足夠的時間？沒有任何時間限制的跡象。

5．足夠的技術？是的，金字塔、錐形的結構本身就是穩定的，可以很好分散壓力負載。對磚石建築技術，人們有過深刻的研究。同樣，項目遠在達到技術限制之間，就已經失敗了。

那麼，既然他們具備了所有的這些條件，爲什麼項目還會失敗呢？他們還缺乏些什麼？兩個方面——交流，以及交流的結果——組織。他們無法相互交談，從而無法合作。當合作無法進行時，工作陷入了停頓。通過史書的字裏行間，我們推測交流的缺乏導致了爭辯、沮喪和群體猜忌。很快，部落開始分裂——大家選擇了孤立，而不是互相爭吵。

## 大型編程專案中的交流

現在，其實也是這樣的情況。因爲左手不知道右手在做什麼，所以進度災難、功能的不合理和系統缺陷紛紛出現。隨著工作的進行，許多小組慢慢地修改自己程式的功能、規模和速度，他們明確或者隱含地更改了一些有效輸入和輸出結果用法上的約定。

例如，程式覆蓋（program-overlay）功能的實現者遇到了問題，並且統計報告顯示了應用程式很少使用該功能。基於這些考慮，他降低了覆蓋功能的速度。與此同時，整個開發隊伍中，其他同事正在設計監控程序。監控程序在很大程度上依賴於覆蓋功能，它在速度上的變化成爲了主要的規格說明變更。因此需要從系統角度來考慮和衡量該變化，以及公開、廣泛地發佈變更結果。

那麼，團隊如何進行相互之間的交流溝通呢？通過所有可能的途徑。

非正式途徑

清晰定義小組內部的相互關係和充分利用電話，能鼓勵大量的電話溝通，從而達到對所書寫文檔的共同理解。

會議

常規專案會議。會議中，團隊一個接一個地進行簡要的技術陳述。這種方式非常有用，能澄清成百上千的細小誤解。

工作手冊。

在專案的開始階段，應該準備正式的專案工作手冊。理所應當，我們專門用一節來討論它。

# 專案工作手冊

是什麼。專案工作手冊不是獨立的一篇文檔，它是對專案必須產出的一系列文檔進行組織的一種結構。

專案所有的文檔都必須是該結構的一部分。這包括目的、外部規格說明、介面說明、技術標準、內部說明和管理備忘錄。

爲什麼。技術說明幾乎是必不可少的。如果某人就硬體和軟體的某部分，去查看一系列相關的用戶手冊。他發現的不僅僅是思路，而且還有能追溯到最早備忘錄的許多文字和章節，這些備忘錄對產品提出建議或者解釋設計。對於技術作者而言，文章的剪裁粘貼與鋼筆一樣有用。

基於上述理由，再加上“未來產品”的品質手冊將誕生於“今天產品” 的備忘錄，所以正確的文檔結構非常重要。事先將專案工作手冊設計好，能保證文檔的結構本身是規範的，而不是雜亂無章的。另外，有了文檔結構，後來書寫的文字就可以放置在合適的章節中。

使用專案手冊的第二個原因是控制資訊發佈。控制資訊發佈並不是爲了限制資訊，而是確保資訊能到達所有需要它的人的手中。

項目手冊的第一步是對所有的備忘錄編號，從而每個工作人員可以通過標題列表來檢索是否有他所需要的資訊。還有一種更好的組織方法，就是使用樹狀的索引結構。而且如果需要的話，可以使用樹結構中的子樹來維護發佈列表。

處理機制。同許多其他的軟體管理問題一樣，隨著專案規模的擴大，技術備忘錄的問題以非線性趨勢增長。10人的專案，文檔僅僅通過簡單的編號就可以了。100人的專案，若干個線性

索引常常可以滿足要求。1000人的項目，人員無可避免地散佈在多個地點，對結構化工作手冊的需要和手冊規模上的要求都緊迫了許多。那麼，用什麼樣的機制來處理呢？

我認爲OS/360專案做得非常好。O.S.Locken強烈要求制訂結構良好的工作手冊，他本人在他的前一個專案1410-7010作業系統中，看到了工作手冊的效果。

我們很快決定了每一個編程人員應該瞭解所有的材料，即在每間辦公室中應保留一份工作手冊的拷貝。

工作手冊的即時更新是非常關鍵的。工作手冊必須是最新的，如果每次變更都要重新列印所有的文檔，實際上這很難做到。不過，如果採用文件夾的方式，則僅需更換變更頁。我們當時擁有電腦編輯系統，它對即時維護有不可思議的幫助。編輯、排版、列印的工作直接在電腦和印表機上完成，周轉時間少於一天。但即便如此，所有接收的人員還是會面臨消化理解的問題。當他第一次收到更改頁時，他需要知道，“修改了什麼？”遲些時候，當他就問題進行諮詢時，他需要知道，“現在的定義是什麼？”。

理解的問題可以通過持續的文檔維護來解決。文檔變更的強調有若干個步驟。首先，必須在頁面上標記發生改變的文本，例如，使用頁邊上的豎線標記每行變化的文字。第二，分發的變更頁附帶獨立的總結性文字，對變更的重要性以及批註進行記錄。

這種機制在我們項目中碰到別的問題之前，穩定運行了六個月。工作手冊大約厚達1.5米！如果將我們在曼哈頓Time-Life大廈辦公室裏所使用的100份手冊疊在一起，它們比這座大廈還要高。另外，每天分發的變更頁大約5釐米，歸入檔案的頁數大概有150頁。日常工作手冊的維護工作佔據了每個工作日的大量時間。

這個時候，我們換用了微縮膠片，在爲每個辦公室配備了微縮膠片閱讀機之後，節約了大量金錢，工作手冊的體積減少了18倍。更重要的是，對數百頁更新工作的幫助——微縮膠片大量地減輕了歸檔問題。

微縮膠片有它的缺點。從管理的角度而言，笨拙的文字歸檔工作確保了所有變更會被閱讀，這正是工作手冊要達到的目的。微縮膠片使工作手冊的維護工作變得過於簡單，除非列舉變化的文字說明和變更膠片一起分發。

另外，微縮膠片不容易被讀者強調、標記和批註。對作者來說，採用文檔方式與讀者溝通更加有效；對讀者來說，文檔更加容易使用。

總之，我覺得微縮膠片是非常好的一種方法。對於大型專案，我建議把它作爲文字工作手冊的補充。

現在如何入手？在當今很多可以應用的技術中，我認爲一種選擇是採用可以直接訪問的檔。在檔中，記錄修訂日期記錄和標記變更標識條。每個用戶可以從一個顯示終端（印表機太慢了）來查閱。每日維護的變更小結以“後進先出”的方式保存，在一個固定的地方提供訪問。編程人員可能會每天閱讀，但如果錯過了一天，他只需在第二天多花一些時間。在他查看小結的同時，他可以停下來，去查詢變更的文字。

注意工作手冊本身沒有發生變化。它還是所有專案文檔的集合，根據某種經過細緻考慮的規則組織在一起。唯一發生改變的地方是分發機制和查詢方法。斯坦福研究機構的D.C.Engelbart和同事開發了一套系統，並用它在ARPA網路專案中建立和維護文檔。

卡內基－梅隆大學的D.L.Parnas提出了更徹底的解決方法[1]。他認爲，編程人員僅瞭解自己負責的部分，而不是整個系統的開發細節時，工作效率最高。這種方法的先決條件是精確和完整地定義所有介面。這的確是一個徹底的解決方法。如果能處理得好，的確是能解決很多"災難"。一個好的資訊系統不但能暴露介面錯誤，還能有助於改正錯誤。

## 大型編程專案的組織架構

如果專案有n個工作人員，則有（$n^2 - n$）/ 2個相互交流的介面，有將近$2^n$個必須合作的潛在團隊。團隊組織的目的是減少不必要交流和合作的數量，因此良好的團隊組織是解決上述交流問題的關鍵措施。

減少交流的方法是人力劃分（division of labor）和限定職責範圍（specialization of function）。當使用人力劃分和職責限定時，樹狀管理結構所映出對詳細交流的需要會相應減少。

事實上，樹狀組織架構是作爲權力和責任的結構出現。其基本原理——管理角色的非重複性——導致了管理結構是樹狀的。但是交流的結構並未限制得如此嚴格，樹狀結構幾乎不能用來描述交流溝通，因爲交流是通過網狀結構進行的。在很多工程活動領域，樹狀類比結構不能很精確地用於描述一般團隊、特別工作組、委員會，甚至是矩陣結構組織。

讓我們考慮一下樹狀編程隊伍，以及要使它行之有效，每棵子樹所必須具備的基本要素。它們是：

1. 任務（a mission）

2. 產品負責人（a producer）

3. 技術主管和結構師（a technical director or architect）

4. 進度（a schedule）

5. 人力的劃分（a division of labor）

6. 各部分之間的介面定義（interface definitions among the parts）

所有這些是非常明顯和約定俗成的，除了產品負責人和技術主管之間有一些區別。我們先分析一下兩個角色，然後再考慮它們之間的關係。

產品負責人的角色是什麼？他組建團隊，劃分工作及制訂進度表。他要求，並一直要求必要的資源。這意味著他主要的工作是與團隊外部，向上和水準地溝通。他建立團隊內部的溝通和報告方式。最後，他確保進度目標的實現，根據環境的變化調整資源和團隊的構架。

那麼技術主管的角色是什麼？他對設計進行構思，識別系統的子部分，指明從外部看上去的樣子，勾畫它的內部結構。他提供整個設計的一致性和概念完整性；他控制系統的複雜程度。當某個技術問題出現時，他提供問題的解決方案，或者根據需要調整系統設計。用A1 Capp所喜歡的一句諺語，他是"攻堅小組中的獨行俠"（inside-man at the skunk works.）。他的溝通交流在團隊中是首要的。他的工作幾乎完全是技術性的。

現在可以看到，這兩種角色所需要的技能是非常不同的。這些技能可以按不同的方式進行組合。產品負責人和技術主管所擁有的特殊技能可以用不同方式組合，組合結果控制和支配了他們之間的關係。團隊的搭建必須根據參與的人員來組織，而不是將人員純粹地按照理論進行安排。

存在三種可能的關係，它們都在實踐中得到了成功的應用。

產品負責人和技術主管是同一個人。這種方式非常容易應用在很小型的隊伍中，可能是三個或六個開發人員。在大型的項目中則不容易得到應用。原因有兩個：第一，同時具有管理技能和技術技能的人很難找到。思考者很少，實幹家更少，思考者－實幹家太少了。

第二，大型項目中，每個角色都必須全職工作，甚至還要加班。對負責人來說，很難在承擔全部管理責任的同時，還能抽出時間進行技術工作。對技術主管來說，很難在保證設計的概念完整性，沒有任何妥協的前提下，擔任管理工作。

產品負責人作爲總指揮，技術主管充當其左右手。這種方法有一些困難。很難在技術主管不參與任何管理工作的同時，建立在技術決策上的權威。

顯然，產品負責人必須預先聲明技術主管的技術權威，在即將出現的絕大部分測試用例中，他必須支援後者的技術決定。要達到這一點，產品責任人和技術主管必須在基本的技術理論上具有相似觀點；他們必須在主要的技術問題出現之前，私下討論它們；產品責任人必須對技術主管的技術才能表現出尊重。

另外，還有一些技巧。例如，產品責任人可以通過一些微妙狀態特徵暗示來（如，辦公室的大小、地毯、裝修、影印機等等）體現技術主管的威信，儘管決策權力的源泉來自管理。

這種組合可以使工作很有效。不幸的是它很少被應用。不過，它至少有一個好處，即專案經理可以使用並不很擅長管理的技術天才來完成工作。

技術主管作爲總指揮，產品負責人充當其左右手。Robert Heinlein在《出售月球的人》（"The Man Who Sold the Moon"）中，用一幅場景描述了這樣的安排：

Coster低下頭，雙手捂著臉，接著，抬起頭。"我知道。我瞭解需要做什麼——但每次我試圖解決技術問題時，總有些該死的笨蛋要我做一些關於卡車、或者電話、以及其他一些討厭的事情。我很抱歉。Harriman先生，我原以爲我可以處理好。"

Harriman非常溫和的說：“Bob，別讓這些事煩你。近來好像睡眠不大好，是嗎？告訴你吧。我將在你的位子上幹幾天，爲你搭建一個免於這些事情干擾的環境。我需要你的大腦工作在反向量、燃油效率和壓力設計上，而不是卡車的合同。”Harriman走到門邊，掃了一圈，點了一個可能是、也可能不是辦公室主要職員的工作人員。“嘿，你！過來一下。”

那個人看上去有些驚慌，站了起來，走到門邊說道，“什麼事？”

“把角落上的那個桌子和上面所有的東西搬到本層樓的一個空的辦公室去，馬上。”

他監督著Coster和他的桌子移到另一個辦公室，看了看，發現新辦公室的電話沒有接上。接著，想了一下，搬了一個長沙發過來。“今晚我們將安裝一個投影儀、繪圖儀、書架和其他一些東西，”他告訴Coster。“把你工程所需要的東西列一個表。”他回到了原來的總工程師辦公室，愉快地想了想如何進行工作組織，以及是否有什麼不妥。

過了四個小時，他帶Berkeley進來，與Coster會面。這位總工程師正在他的桌子上睡覺，頭枕在臂彎裏。Harriman慢慢地退出去，但Coster醒了過來。“喔，對不起，”他有點不好意思地說，“我肯定是打了個瞌睡。”

“這就是我給你帶來長沙發的原因，”Harriman說道。“它更加舒適。Bob，來見一下Jock Berkeley。他是你的新奴隸，你仍是總工程師，毫無疑問的老闆。Jock是其他一切的主管。從現在起，你不需要擔心其他的任何問題，除了建造登月飛船的一些細節問題。”

他們握了一下手。“Coster先生，我只想問一件事，”Berkeley認真的說，“所有你需要做的事，我都無權過問——你即將進行一個技術演示——但是看在上帝的份上，能否記錄一下，從而讓我瞭解一下。我將會把一個開關放在你的桌上，它會開啓桌上的一個密封的錄影機。”

“好的！”Coster正看著他，Harriman想，夠年輕的。

“如果要做任何非技術的事情，不需要自己動手。只需按個按鈕知會一聲，它們就會被完成！”Berkeley掃了Harriman一眼。“老闆說他想同你談一談實際的工作。我得先走，去忙去了。”他離開了。

Harriman坐了下來，Coster整了整衣服，說道，“喔！”

“感覺好一些了？”

“我喜歡Berkeley這小夥子的樣子。”

“太好了！不用擔心，他現在就是你的學生兄弟。我以前用過他。你可以認爲你正住在一個頭等的療養院裏。”[2]

這個故事幾乎不需要任何的分析解釋，這種安排同樣能使工作非常有效。

我猜測最後一種安排對小型的團隊是最好的選擇，如同在第3章《外科手術隊伍》一文中

所述。對於真正大型專案中的一些開發隊伍，我認爲產品負責人作爲管理者是更合適的安排。

巴比倫塔可能是第一個工程上的徹底失敗，但它不是最後一個。交流和交流的結果——組織，是成功的關鍵。交流和組織的技能需要管理者仔細考慮，相關經驗的積累和能力的提高同軟體技術本身一樣重要。

# 胸有成竹（*Calling the Shot*）

實踐是最好的老師。

*- PUBILIUS*

實踐是最好的老師，但是，如果不能從中學習，再多的實踐也沒有用。

- 《可憐的理查年鑒》

*Practice is the best of all instructors.*

*- PUBILIUS*

*Experience is a dear teacher, but fools will learn at no other.*

*- POOR RICHARD'S ALMANAC*

系統編程需要花費多長的時間？需要多少的工作量？如何進行估計？

先前，我推薦了用於計畫進度、編碼、構件測試和系統測試的比率。首先，需要指出的是，僅僅通過對編碼部分的估計，然後應用上述比率，是無法得到對整個任務的估計的。編碼大約只占了問題的六分之一左右，編碼估計或者比率的錯誤可能會導致不合理的荒謬結果。

第二，必須聲明的是，構建獨立小型程式的資料不適用於編程系統產品。對規模平均爲3200指令的程式，如Sackman、Erikson和Grant的報告中所述，大約單個的程式師所需要的編碼和調試時間爲178個小時，由此可以外推得到每年35,800語句的生產率。而規模只有一半的程式花費

時間大約僅爲前者的四分之一，相應推斷出的生產率幾乎是每年80,000代碼行[1]。計畫、編制文檔、測試、系統集成和培訓的時間必須被考慮在內。因此，上述小型專案資料的外推是沒有意義的。就好像把100碼短跑記錄外推，得出人類可以在3分鐘之內跑完1英里的結論一樣。

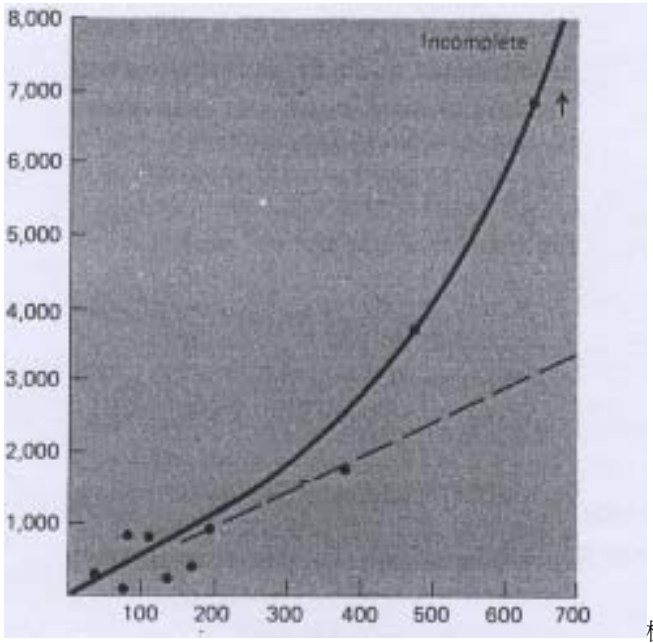
在將上述觀點拋開之前，儘管不是爲了進行嚴格的比較，我們仍然可以留意到一些事情。即使在不考慮相互交流溝通，開發人員僅僅回顧自己以前工作的情況下，這些數位仍然顯示出工作量是規模的冪函數。

圖8.1講述了這個悲慘的故事。它闡述了Nanus和Farr[2]在System Development Corporation公司所做研究，結果表明該指數爲1.5，即，

$$\text{工作量} = (\text{常數}) \times (\text{指令的數量})^{1.5}$$

Weinwurm[3]的SDC研究報告同樣顯示出指數接近於1.5。

現在已經有了一些關於編程人員生產率的研究，提出了很多估計的技術。 Morin對所發佈的資料進行了一些調查研究[4]。這裏僅僅給出了若干特別突出的條目。




人月

機器指令：千

注：

incomplete－未終結的

圖8.1：編程工作量是程式規模的函數

## Portman的數據

曼徹斯特Computer Equipment Organization（Northwest）的ICL軟體部門的經理Charles Portman，提出了另一種有用的個人觀點[5]。

他發現他的編程隊伍落後進度大約1/2，每項工作花費的時間大約是估計的兩倍。這些估計通常是非常仔細的，由很多富有經驗的團隊完成。他們對PERT圖上數百個子任務估算過（用人小時作單位）。當偏移出現時，他要求他們仔細地保存所使用時間的日誌。日誌顯示事實上他的團隊僅用了百分之五十的工作周，來進行實際的編程和調試，估算上的失誤完全可以由該情況來解釋。其餘的時間包括機器的當機時間、高優先順序的無關瑣碎工作、會議、文字工作、公司業務、疾病、事假等等。簡言之，專案估算對每個人年的技術工作時間數量做出了不現實的假設。我個人的經驗也在相當程度上證實了他的結論[6]。

## Aron的數據

Joel Aron，IBM在馬里蘭州蓋茲堡的系統技術主管，在他所工作過的9個大型專案（簡要地說，大型意味著程式師的數目超過25人，將近30,000行的指令）[7]的基礎上，對程式師的生產率進行了研究。他根據程式師（和系統部分）之間的交互劃分這些系統，得到了如下的生產率：

| | |
|---|---|
| 非常少的交互 | 10,000指令每人年 |
| 少量的交互 | 5,000 |
| 較多的交互 | 1,500 |

該人年資料未包括支援和系統測試活動，僅僅是設計和編程。當這些資料採用除以2，以包括系統測試的活動時，它們與Harr的資料非常的接近。

## Harr的數據

John Harr，Bell電話實驗室電子交換系統領域的編程經理，在1969年春季聯合電腦會議[8]的論文中，彙報了他和其他人的經驗。這些資料如圖8.2、8.3和8.4所示。

這些圖中，圖8.2是最資料詳細和最有用的。頭兩個任務是基本的控制程式，後兩個是基本的語言翻譯。生產率以經調試的指令/人年來表達。它包括了編程、構件測試和系統測試。沒有包括計畫、硬體機器支援、文書工作等類似活動的工作量。

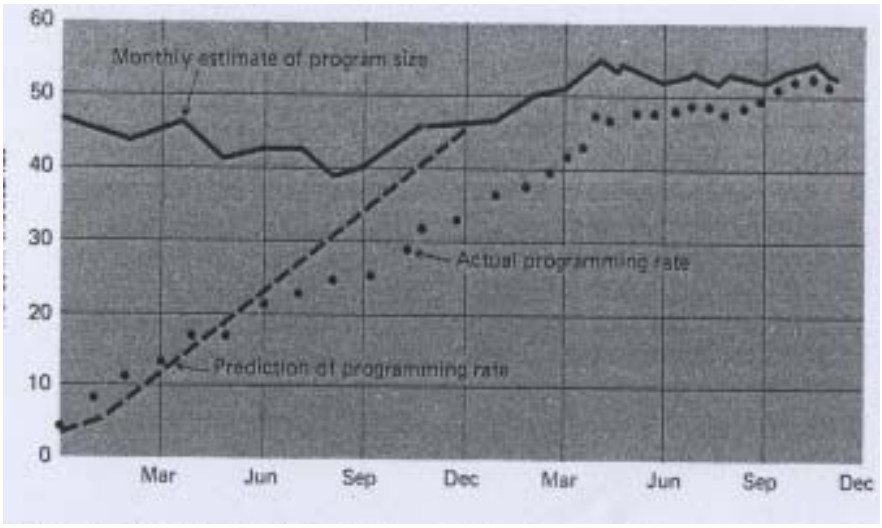
生產率同樣地被劃分爲兩個類別，控制程式的生產率大約是600指令每人年，語言翻譯大約是2200指令每人年。注意所有的四個程式都具有類似的規模——差異在於工作組的大小、時間的長短和模組的個數。那麼，哪一個是原因，哪一個是結果呢？是否因爲控制程式更加複雜，

所以需要更多的人員？或者因爲它們被分派了過多的人員，所以要求有更多的模組？是因爲複雜程度非常高，還是分配較多的人員，導致花費了更長的時間？沒有人可以確定。控制程式確實更加複雜。除開這些不確定性，資料反映了實際的生產率——描述了在現在的編程技術下，大型系統開發的狀況。因此，Harr資料的確是真正的貢獻。

圖8.3和8.4顯示了一些有趣的資料，將實際的編程速度、調試速度與預期做了對比。

| | 程式單元 | 程式師人數 | 年 | 人年 | 程式字數 | 字/人年 |
|---|---|---|---|---|---|---|
| 操作性 | 50 | 83 | 4 | 101 | 52,000 | 515 |
| 維護 | 36 | 60 | 4 | 81 | 51,000 | 630 |
| 編譯器 | 13 | 9 | $2\frac{1}{4}$ | 17 | 38,000 | 2230 |
| 語言解釋器（彙編） | 15 | 13 | $2\frac{1}{2}$ | 11 | 25,000 | 2270 |

圖8.2：4個NO.1的ESS編程工作總結




注：

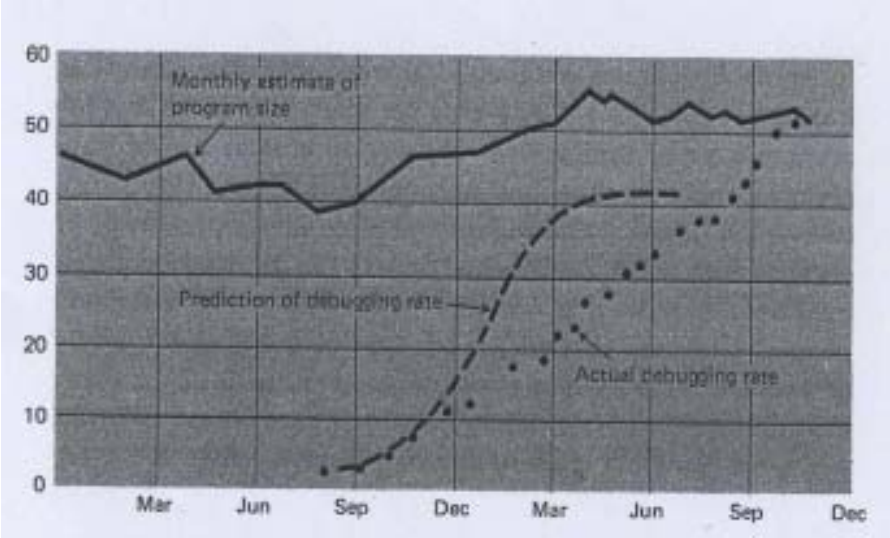
Monthly estimate of program size－程式規模月估計

Actual Programmize rate－實際編程速度

Prediction of programming rate－預計編程速度

圖8.3：ESS預計和實際的編程速度

機器指令：千字




注：

Monthly estimate of program size－程式規模月估計

Actual Programmize rate－實際調試速度

Prediction of programming rate－預計調試速度

圖8.4：ESS預計和實際的調試速度

## OS/360的數據

IBM OS/360的經驗，儘管沒有Harr那麼詳細的資料，但還是證實了那些結論。就控制程式組的經驗而言，生產率的範圍大約是600～800（經過調試的指令）/人年。語言翻譯小組所達到的生產率是2000～3000（經過調試的指令）/人年。這包括了小組的計畫、代碼構件測試、系統測試和一些支援性活動。就我的觀點來說，它們同Harr的資料是可比的。

Aron、Harr和OS/360的資料都證實，生產率會根據任務本身複雜度和困難程度表現出顯著差異。在複雜程度估計這片"沼澤"上的指導原則是：編譯器的複雜度是批次處理程式的三倍，作業系統複雜度是編譯器的三倍[8]。

## Corbato的數據

Harr和OS/360的資料都是關於組合語言編程的，好像使用高階語言系統編程的資料公佈得很少。Corbato的MIT專案MAC報告表示在MULTICS系統上，平均生產率是1200行經調試的PL/I語句（大約在1和2百萬指令之間）/人年。

該數字非常令人興奮。如同其他的專案，MULTICS包括了控制程式和語言翻譯程式。和其他項目一樣，它產出的是經過測試和文檔化的系統編程產品。在所包括的工作類型方面，資料

看上去是可以比較的。該數位是其他專案中控制程式和翻譯器程式生產率的良好平均值。

機器指令：千字

但Corbato的數字是行/人年，不是指令！系統中的每個語句對應於手寫代碼的3至5個指令！這意味著兩個重要的結論。

對常用編程語句而言。生產率似乎是固定的。這個固定的生產率包括了編程中需要注釋，並可能存在錯誤的情況.

使用適當的高階語言，編程的生產率可以提高5倍。

# 削足適履（*Ten Pounds in a Five-Pound Sack*）

他應該瞪大眼睛盯著諾亞，□□好好學習，看他們是怎樣把那麼多東西裝到一個小小的方舟上的。

- 西德尼·史密斯，愛丁堡評論

*the author should gaze at Noah, and ... learn, as they did in the Ark, to crowd*

*a great deal of matter into a very small compass.*

*- SYDENY SMITH. EDINBURGH REVIEW*

## 作爲成本的程式空間

程式有多大？除了運行時間以外，它所佔據的空間也是主要開銷。這同樣適用于專用開發的程式，用戶支付給開發者一筆費用，作爲必要分擔的開發成本。考慮一下IBM APL互動式軟體系統，它的租金爲每月400美金，在使用時，它至少佔用160K位元組的記憶體。在Model 165上，記憶體租金大約是12美金/每月每千位元組。如果程式在全部時間內都可用，他需要支付400美

元的軟體使用費和1920美金的記憶體租用費。如果某個人每天使用APL系統4小時，他每月需要支出400美元的軟體租金和320美元的記憶體租用費。

常常聽到的一個“可怕的”談論是在2M記憶體的機器上，作業系統就需要佔用400K記憶體。這種言論就好像批評波音747飛機，僅僅因爲它耗資兩千七百萬美元一樣無知。我們首先必須問的是“它能幹什麼？”。對於所耗費的資金，獲得的易用性和性能是什麼？投資在記憶體上的每月4800美元的租金能否比用在其他硬體、編程人員、應用程式上更加有效？

當系統設計者認爲對用戶而言，常駐程式記憶體的形式比加法器、磁片等更加有用時，他會將硬體實現中的一部分移到記憶體上。相反的，其他的做法是非常不負責任的。所以，應該從整體上來進行評價。沒有人可以在自始至終提倡更緊密的軟硬體設計集成的同時，又僅僅就規模本身對軟體系統提出批評。

由於規模是軟體系統產品用戶成本中如此大的一個組成部分，開發人員必須設置規模的目標，控制規模，考慮減小規模的方法，就像硬體開發人員會設立元器件數量目標，控制元器件的數量，想出一些減少零件的方法。同任何開銷一樣，規模本身不是壞事，但不必要的規模是不可取的。

## 規模控制

對專案經理而言，規模控制既是技術工作的一部分，也是管理工作的一部分。他必須研究用戶和他們的應用，以設置將開發系統的規模。接著，把這些系統劃分成若干部分，並設定每個部分的規模目標。由於規模－速度權衡方案的結果在很大的範圍內變化，規模目標的設置是一件頗具技巧的事情，需要對每個可用方案有深刻的瞭解。聰明的項目經理還會給自己預留一些空間，在工作推行時分配。

在OS/360項目中，即使所有的工作都完成得相當仔細，我們依然能從中得到一些痛苦的教訓。

首先，僅對核心程式設定規模目標是不夠的，必須把所有的方面都編入預算。在先前的大多數作業系統中，系統駐留在磁帶上，長時間的磁帶搜索意味著它無法自如地運用在程式片段上。OS/360和它的前任產品Stretch作業系統和1410-7010磁片作業系統一樣，是駐留在磁片上的。它的開發者對自由、廉價的磁片訪問感到欣喜。而如果使用磁帶，會給性能帶來災難性的後果。

在爲每個單元設立核心規模的同時，我們沒有同時設置訪問的目標。正如大家能想到的一樣，當程式師發現自己的單元核心未能達到要求時，他會把它分解成程式庫。這個過程本身增加了程式整體的規模，並降低了運行速度。最重要的是，我們的管理控制系統既沒有度量，也沒有捕獲這些問題。每個人都彙報了核心的大小，都在目標範圍之內，所以沒有人發現規模上的問題。

幸運的是，OS/360性能仿真程式投入使用的時間較早。第一次運行的結果反映出很大的麻煩。Fortran H，在帶磁鼓的Moda1 65上，每分鐘模擬編譯5條語句！嵌入的常式顯示控制程式模組進行了很多次磁片訪問。甚至使用頻繁的監控模組也犯了很多同樣的錯誤，結果很類似於

頁面的切換。

第一個道理很清楚：和制訂駐留空間預算一樣，應該制訂總體規模的預算；和制訂規模預算一樣，應該制訂後臺存儲訪問的預算。

下一個教訓十分類似。在每個模組分配功能之前，已編制了空間的預算。其結果是，任何在規模上碰到問題的程式師，會檢查自己的代碼，看是否能將其中一部分扔給其他人。因此，控制程式所管理的緩衝區成爲了用戶空間的一部分。更嚴重的是，所有的控制模組都有相同的問題，徹底影響了系統的穩定和安全性。

所以，第二個道理也很清晰：在指明模組有多大的同時，確切定義模組的功能。

第三個更深刻的教訓體現在以上的經驗中。專案規模本身很大，缺乏管理和溝通，以至於每個團隊成員認爲自己是爭取小紅花的學生，而不是構建系統軟體產品的人員。爲了滿足目標，每個人都在局部優化自己的程式，很少會有人停下來，考慮一下對客戶的整體影響。對大型專案而言，這種導向和缺乏溝通是最大的危險。在整個實現的過程期間，系統結構師必須保持持續的警覺，確保連貫的系統完整性。在這種監督機制之外，是實現人員自身的態度問題。培養開發人員從系統整體出發、面向用戶的態度是軟體編程管理人員最重要的職能。

## 空間技能

空間預算的多少和控制並不能使程式規模減小，爲實現這一目標，它還需要一些創造性和技能。

顯然，在速度保持不變的情況下，更多的功能意味著需要更多的空間。所以，其中的一個技巧是用功能交換尺寸。這是一個較早的、影響較深遠的策略問題：爲用戶保留多少選擇？程式可以有很多的選擇功能，每個功能僅佔用少量的空間。也可以設計成擁有若干選項分組，根據選項組來剪裁程式。任何一系列特殊選項被合併在一起進行分組時，程式需要的空間較少。這很像小汽車。如果把照明燈、點煙器和時鐘作爲整個配件來標明價格，則成本會比單獨提供這些選擇所需要的成本低。所以，設計人員必須決定用戶可選項目的粗細程度。

在記憶體大小一定的情況下進行系統設計時，會出現另外一個基本問題。記憶體受限的後果是即使最小的功能模組，它的適用範圍也難以得到推廣。在最小規模的系統中，大多數模組被覆蓋（overlaid），系統的主幹佔用的空間，會被用作其他部分的交換頁面。它的尺寸決定了所有模組的尺寸。而且將功能分解到很小的模組會耗費空間和降低性能。所以，當可以提供20倍臨時性空間的大型系統使用這些模組時，節省的也僅僅是訪問次數，仍然會因爲模組的規模引起空間和速度上的損失。這樣後果其實是——很難用小型系統的模組構造出非常高效的系統。

第二個技能是考慮空間－時間的折衷。對於給定的功能，空間越多，速度越快。這一點在很大的範圍內都適用。也正是這一點使空間預算成爲可能。

項目經理可以做兩件事來幫助他的團隊取得良好的空間－時間折衷。一是確保他們在編程技能上得到培訓，而不僅僅是依賴他們自己掌握的知識和先前的經驗。特別是使用新語言或者

新機器時，培訓顯得尤其重要。熟練使用往往需要快速的學習和經驗的廣泛共用，也許它應該伴隨特別的新技術獎勵或者表揚。

另外一種方法是認識到編程需要技術積累，需要開發很多公共單元構件。每個專案要有能用於佇列、搜索和排序的常式或者巨集庫。對於每項功能，庫至少應該有兩個程式實現：運行速度較快和短小精煉的。上述的公共庫開發是一件重要的實現工作，它可以與系統設計工作並行進行。

## 資料的表現形式是編程的根本

創造出自精湛的技藝，精煉、充分和快速的程式也是如此。技藝改進的結果往往是戰略上的突破，而不僅僅是技巧上的提高。這種戰略上突破有時是一種新的演算法，如快速傅立葉變換，或者是將比較演算法的複雜度從$n^2$降低到$n \log n$。

更普遍的是，戰略上突破常來自資料或表的重新表達——這是程式的核心所在。如果提供了程式流程圖，而沒有表資料，我仍然會很迷惑。而給我看表資料，往往就不再需要流程圖，程式結構是非常清晰的。

很容易就能找到重新表達所帶來好處的例子。我記得有一個年輕人承擔了爲IBM650開發精細的控制臺解釋器的任務。他發現用戶交互得很慢，並且空間很昂貴。於是，他編寫了一個解釋器的解釋器，使得最後程式所占的空間減少到不可思議的程度。Digitek小而優雅的Fortran編譯器使用了非常密集的、專業化的代碼來表達自己，以至於不再需要外部存儲。對這種表達方式解碼會損失一些時間，但由於避免了輸入－輸出，反而得到了十倍的補償。（Brooks和Iverson第六章結尾的練習以及Knuth的練習[2]—自動資料處理[1]包含了許多類似的例子。）

由於缺乏空間而絞盡腦汁的編程人員，常常能通過從自己的代碼中掙脫出來，回顧、分析實際情況，仔細思考程式的資料，最終獲得非常好的結果。實際上，資料的表現形式是編程的根本。

# 提綱挈領（*The Documentary Hypothesis*）

前提：

在一片檔的汪洋中，少數文檔形成了關鍵的樞紐，每件專案管理的工作都圍繞著它們運轉。它們是經理們的主要個人工具。

*The hypothesis:*

*Amid a wash of paper, a small number of documents become the critical pivots around which every project's management revolves. These are the manager's chief personal tools.*

技術、周邊組織機構、行業傳統等若干因素湊在一起，定義了項目必須準備的一些文書工作。對於一個剛從技術人員中任命的專案經理來說，這簡直是一件徹頭徹尾令人生厭的事情，而且是毫無必要和令人分心的，充滿了被吞沒的威脅。但是，在實際工作中，大多數情況都是這樣的。

慢慢的，他逐漸認識到這些文檔的某些部分包含和表達了一些管理方面的工作。每份文檔的準備工作是集中考慮，並使各種討論意見明朗化的主要時刻。如果不這樣，專案往往會處於無休止的混亂狀態。文檔的跟蹤維護是專案監督和預警的機制。文檔本身可以作爲檢查列表、狀態控制，也可以作爲彙報的資料基礎。

爲了闡明軟體專案如何開展這項工作，我們首先借鑒一下其他行業一些有用的文檔資料，看是否能進行歸納，得出結論。

## 電腦產品的文檔

如果要製造一台機器，哪些是關鍵的文檔呢？

目標：定義待滿足的目標和需要，定義迫切需要的資源、約束和優先順序。

技術說明：電腦手冊和性能規格說明。它是在計畫新產品時第一個產生，並且最後完成的文檔。

進度、時間表

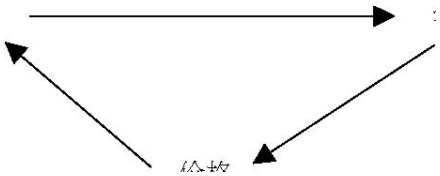
預算：預算不僅僅是約束。對管理人員來說，它還是最有用的文檔之一。預算的存在會迫使技術決策的制訂，否則，技術決策很容易被忽略。更重要的是，它促使和澄清了策略上的一些決定。

組織機構圖

工作空間的分配

報價、預測、價格：這三個因素互相牽制，決定了專案的成敗。

爲了進行市場預測，首先需要制訂產品性能說明和確定假設的價格。從市場預測得出的數值，連同從設計得出的元件單元的數量，決定了生產的估計成本，進而可以得到每個單元的開發工作量和固定的成本。固定成本又決定了價格。



如果價格低於假設值，令人欣慰的迴圈開始了。預測值較高，單元成本較低，因此價格能夠繼續降低。

如果價格高於預測值，災難性的迴圈開始了，所有的人必須努力奮鬥來打破這個迴圈。新應用程式必須提高性能和支援更高的市場預測。成本必須降低，以產出更低的報價。這個迴圈的壓力常常是激勵市場人員和工程師工作的最佳動力。

同時，它也會帶來可笑的躊躇和搖擺。我記得曾經有一個專案，在三年的開發週期中，機器指令計數器的設計每六個月變化一次。在某個階段，需要好一點的性能時，指令計數器採用觸發器來實現；下一個階段，成本降低是主要的焦點，指令計數器採用記憶體來實現。在另一個項目中，我所見過的最好的一個項目經理常常充當一個大型調速輪的角色，他的慣性降低了來自市場和管理人員的起伏波動。

預測
Xerox
Palo Alto
research
Center

報價

價格

## 大學科系的文檔

除了目的和活動上的巨大差異，數量類似、內容相近的各類文檔形成了大學系主任的主要資料集合。校長、教師會議或系主任的每一個決定幾乎都是一個技術說明，或者是對這些文檔的變更。

目標

課程描述

學位要求

研究報告（申請基金時，還要求計畫）

課程表和課程的安排

預算

教室分配

教師和研究生助手的分配

注意這些文檔的組成與電腦專案非常相似：目標、產品說明、時間安排、資金分配、空間分派和人員的劃分。只有價格文檔是不需要的，學校的決策機構完成了這項任務。這種相似性不是偶然的——任何管理任務的關注焦點都是時間、地點、人物、做什麼、資金。

## 軟體專案的文檔

在許多軟體專案中，開發人員從商討結構的會議開始，然後開始書寫代碼。不論專案的規模如何小，專案經理聰明的做法都是：立刻正式生成若干文檔作爲自己的資料基礎，哪怕這些迷你文檔非常簡單。接著，他會和其他管理人員一樣要求各種文檔。

做什麼：目標。定義了待完成的目標、迫切需要的資源、約束和優先順序。

做什麼：產品技術說明。以建議書開始，以用戶手冊和內部文檔結束。速度和空間說明是關鍵的部分。

時間：進度表

資金：預算

地點：工作空間分配

人員：組織圖。它與介面說明是相互依存的，如同Conway的規律所述：“設計系統的組織架構受到產品的約束限制，生產出的系統是這些組織機構溝通結構的映射。[1]”Conway接著指出，一開始反映系統設計的組織架構圖，肯定不會是正確的。如果系統設計能自由地變化，則專案組織架構必須爲變化做準備。

## 爲什麼要有正式的文檔？

首先，書面記錄決策是必要的。只有記錄下來，分歧才會明朗，矛盾才會突出。書寫這項活動需要上百次的細小決定，正是由於它們的存在，人們才能從令人迷惑的現象中得到清晰、確定的策略。

第二，文檔能夠作爲同其他人的溝通管道。項目經理常常會不斷發現，許多理應被普遍認同的策略，完全不爲團隊的一些成員所知。正因爲項目經理的基本職責是使每個人都向著相同的方向前進，所以他的主要工作是溝通，而不是做出決定。這些文檔能極大地減輕他的負擔。

最後，專案經理的文檔可以作爲資料基礎和檢查列表。通過週期性的回顧，他能清楚專案所處的狀態，以及哪些需要重點進行更改和調整。

我並不是很同意銷售人員所吹捧的"完備資訊管理系統"——管理人員只需在電腦上輸入查詢，顯示幕上就會顯示出結果。有許多基本原因決定了上述系統是行不通的。一個原因是只有一小部分管理人員的時間——可能只有20%——用來從自己頭腦外部獲取資訊。其他的工作是溝通：傾聽、報告、講授、規勸、討論、鼓勵。不過，對於基於資料的部分，少數關鍵的文檔是至關重要的，它們可以滿足絕大多數需要。

專案經理的任務是制訂計畫，並根據計畫實現。但是只有書面計畫是精確和可以溝通的。計畫中包括了時間、地點、人物、做什麼、資金。這些少量的關鍵文檔封裝了一些專案經理的工作。如果一開始就認識到它們的普遍性和重要性，那麼就可以將文檔作爲工具友好地利用起來，而不會讓它成爲令人厭煩的繁重任務。通過遵循文檔開展工作，專案經理能更清晰和快速地設定自己的方向。

# 未雨綢繆（***Plan to Throw One Away***）

不變只是願望，變化才是永恆。

*- SWIFT*

普遍的做法是，選擇一種方法，試試看；如果失敗了，沒關係，再試試別的。不管怎麼樣，重要的是先去嘗試。

- 佛蘭克林 *D.* 羅斯福[1]

*There is nothing in this world constant but inconstancy.*

*- SWIFT*

*It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.*

*- FRANKLIN D. ROOSEVELT*[1]

## 試驗性工廠和增大規模

化學工程師很早就認識到，在實驗室可以進行的反應過程，並不能在工廠中一步實現。一個被稱爲“實驗性工廠（pilot planet）”的中間步驟是非常必要的，它會爲提高產量和在缺乏保護的環境下運作提供寶貴經驗。例如，海水淡化的實驗室過程會先在產量爲10,000加侖/每天的試驗場所測試，然後再用於2,000,000加侖/每天的淨化系統。

軟體系統的構建人員也面臨類似的問題，但似乎並沒有吸取教訓。一個接一個的軟體專案都是一開始設計演算法，然後將演算法應用到待發佈的軟體中，接著根據時間進度把第一次開發的產品發佈給顧客。

對於大多數專案，第一個開發的系統並不合用。它可能太慢、太大，而且難以使用，或者三者兼而有之。要解決所有的問題，除了重新開始以外，沒有其他的辦法——即開發一個更靈巧或者更好的系統。系統的丟棄和重新設計可以一步完成，也可以一塊塊地實現。所有大型系統的經驗都顯示，這是必須完成的步驟[2]。而且，新的系統概念或新技術會不斷出現，所以開發的系統必須被拋棄，但即使是最優秀的專案經理，也不能無所不知地在最開始解決這些問題。

因此，管理上的問題不再是“是否構建一個試驗性的系統，然後拋棄它？”你必須這樣做。現在的問題是“是否預先計畫拋棄原型的開發，或者是否將該原型發佈給用戶？”從這個角度看待問題，答案更加清晰。將原型發佈給用戶，可以獲得時間，但是它的代價高昂——對於用戶，使用極度痛苦；對於重新開發的人員，分散了精力；對於產品，影響了聲譽，即使最好的再設計也難以挽回名聲。

因此，爲捨棄而計畫，無論如何，你一定要這樣做。

## 唯一不變的就是變化本身

一旦認識到試驗性的系統必須被構建和丟棄，具有變更思想的重新設計不可避免，從而直面整個變化現象是非常有用的。第一步是接受這樣的事實：變化是與生俱來的，不是不合時宜和令人生厭的異常情況。Cosgrove很有洞察力地指出，開發人員交付的是用戶滿意程度，而不僅僅是實際的產品。用戶的實際需要和用戶感覺會隨著程式的構建、測試和使用而變化[3]。

當然對於硬體產品而言，同樣需要滿足要求，例如新型汽車或者電腦。但物體的客觀存在容納和階段化（量子化）了用戶對變更的要求。軟體產品易於掌握的特性和不可見性，導致它的構建人員面臨永恆的需求變更。

> 我從不建議顧客目標和需求的所有變更必須、能夠、或者應該整合到設計中。專案開始時建立的基準，肯定會隨著開發的進行越來越高，甚至開發不出任何產品。

然而，目標上的一些變化無可避免，事先爲它們做準備總比假設它們不會出現要好得多。不但目標上的變化不可避免，而且設計策略和技術上的變化也不可避免。拋棄原型概念本身就是對事實的接受——隨著學習的過程更改設計[4]。

## 爲變更計畫系統

如何爲上述變化設計系統，是個非常著名的問題，在書本上被普遍討論——可能討論得比實踐還要多得多。它們包括細緻的模組化、可擴展的函數、精確完整的模組間介面設計、完備的文檔。另外，還可能會採用包括調用佇列和表驅動的一些技術。

最重要的措施是使用高階語言和自文檔技術，以減少變更引起的錯誤。採用編譯時的操作來整合標準聲明，在很大程度上幫助了變化的調整。

變更的階段化是一種必要的技術。每個產品都應該有數位版本號，每個版本都應該有自己的日程表和凍結日期，在此之後的變更屬於下一個版本的範疇。

## 爲變更計畫組織架構

Cosgrove主張把所有計劃、里程碑、日程安排都當作是嘗試性的，以方便進行變化。這似乎有些走極端——現在軟體編程小組失敗的主要原因是管理控制得太少，而不是太多。

不過，他提出了一種卓越的見解。他觀察到不願意爲設計書寫文檔的原因，不僅僅是由於惰性或者時間壓力。相反，設計人員通常不願意提交嘗試性的設計決策，再爲它們進行辯解。“通過設計文檔化，設計人員將自己暴露在每個人的批評之下，他必須能夠爲他的每個結果進行辯護。如果團隊架構因此受到任何形式的威脅，則沒有任何東西會被文檔化，除非架構是完全受到保護的。

爲變更組建團隊比爲變更進行設計更加困難。每個人被分派的工作必須是多樣的、富有拓展性的工作，從技術角度而言，整個團隊可以靈活地安排。在大型的專案中，專案經理需要有兩個和三個頂級程式師作爲技術輕騎兵，當工作繁忙最密集的時候，他們能急馳飛奔，解決各種問題。

當系統發生變化時，管理結構也需要進行調整。這意味著，只要管理人員和技術人才的天賦允許，老闆必須對他們的能力培養給予極大的關注，使管理人員和技術人才具有互換性。

這其中的障礙是社會性的，人們必須同頑固的戒心做鬥爭。首先，管理人員自己常常認爲高級人員太“有價值”，而捨不得讓他們從事實際的編程工作；其次，管理人員擁有更高的威信。爲了克服這個問題，如Bell Labs的一些實驗室，廢除了所有的職位頭銜。每個專業人士都是“技術人員中的一員”。而IBM的另外一些實驗室，保持了兩條職位晉升線，如圖11.1所示。相應的級別在概念上是相同的。

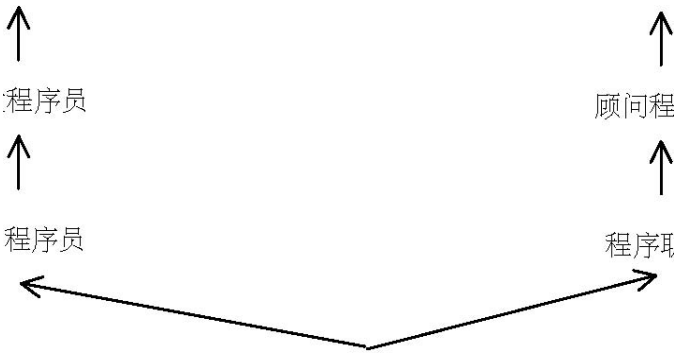



圖11.1：IBM的兩條職位晉升線

很容易爲上述層次建立相互一致的薪水級別。但要建立一致的威信，會困難一些。比如，辦公室的大小和佈局應該相同。秘書和其他支持也必須相同。從技術線向管理同級調動時，不能伴隨著待遇的提升，而且應該以“調動”，而不是“晉升”的名義。相反的調整則應該伴隨著待遇的提高，對於傳統意識進行補償是必要的。

管理人員需要參與技術課程，高級技術人才需要進行管理培訓。專案目標、進展、管理問題必須在高級人員整體中得到共用。

只要能力允許，高層人員必須時刻做好技術和情感上的準備，以管理團隊或者親自參與開發工作。這是件工作量很大的任務，但顯然很值得！

組建外科手術隊伍式的軟體發展團隊，這整個觀念是對上述問題的徹底衝擊。其結果是當高級人才編程和開發時，不會感到自降身份。這種方法試圖清除那些會剝奪創造性樂趣的社會障礙。

另外，上述組織架構的設計是爲了最小化成員間的介面。同樣的，它使系統在最大程度上易於修改。當組織構架必須變化時，爲整個“外科手術隊伍”重新安排不同的軟體發展任務，會變得相對容易一些。這的確是一個長期有效的靈活組織構架解決方案。

管理線

高級程式師

開發程式師

項目程式師

技術線

高級程式師

顧問程式師

程式職員

# 前進兩步，後退一步

在程式發佈給顧客使用之後，它不會停止變化。發佈後的變更被稱爲“程式維護”，但是軟體的維護過程不同於硬體維護。

電腦系統的硬體維護包括了三項活動——替換損壞的器件、清潔和潤滑、修改設計上的缺陷。（大多數情況下——但不是全部——變更修復的是實現上、而不是結構上的一些缺陷。對於用戶而言，這常常是不可見的。）

軟體維護不包括清潔、潤滑和對損壞器件的修復。它主要包含對設計缺陷的修復。和硬體維護相比，這些軟體變更包含了更多的新增功能，它通常是用戶能察覺的。

對於一個廣泛使用的程式，其維護總成本通常是開發成本的40%或更多。令人吃驚的是，該成本受用戶數目的嚴重影響。用戶越多，所發現的錯誤也越多。

麻省理工學院核科學實驗室的Betty Campbell指出特定版本的軟體發佈生命期中一個有趣的迴圈。如圖11.2所示。起初，上一個版本中被發現和修復的bug，在新的版本中仍會出現。新版本中的新功能會產生新的bug。解決了這些問題之後，程式會正常運行幾個月。接著，錯誤率會重新攀升。Campbell認爲這是因爲用戶的使用到達了新的熟練水準，他們開始運用新的功能。這種高強度的考驗查出了新功能中很多不易察覺的問題。[5]

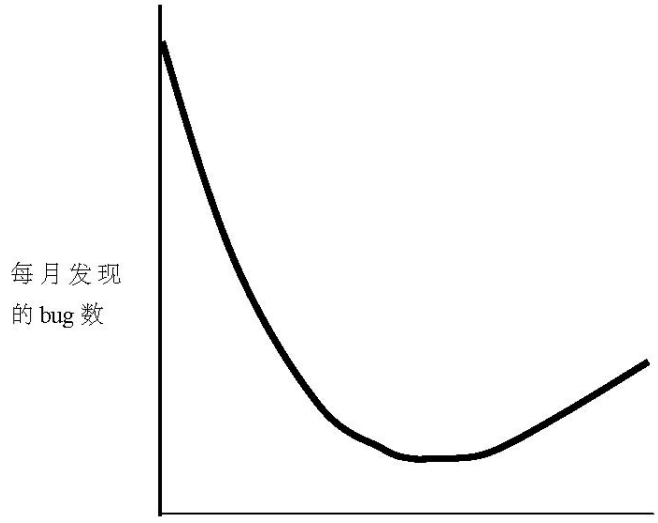



圖11.2：出現的bug數量是發佈時間的函數

每月發現的bug
數

安裝後的時間（月）

程式維護中的一個基本問題是——缺陷修復總會以（20－50）%的機率引入新的bug。所以整個過程是前進兩步，後退一步。

爲什麼缺陷不能更徹底地被修復？首先，看上去很輕微的錯誤，似乎僅僅是局部操作上的失敗，實際上卻是系統級別的問題，通常這不是很明顯。修復局部問題的工作量很清晰，並且往往不大。但是，更大範圍的修復工作常常會被忽視，除非軟體結構很簡單，或者文檔書寫得非常詳細。其次，維護人員常常不是編寫代碼的開發人員，而是一些初級程式師或者新手。

作爲引入新bug的一個後果，程式每條語句的維護需要的系統測試比其他編程要多。理論上，在每次修復之後，必須重新運行先前所有的測試用例，從而確保系統不會以更隱蔽的方式被破壞。實際情況中，回歸測試必須接近上述理想狀況，所以它的成本非常高。

顯然，使用能消除、至少是能指明副作用的程式設計方法，會在維護成本上有很大的回報。同樣，設計實現的人員越少、介面越少，產生的錯誤也就越少。

# 前進一步，後退一步

Lehman和Belady研究了大型作業系統的一系列發佈版本的歷史[6]。他們發現模組數量隨版本號的增加呈線性增長，但是受到影響的模組以版本號指數的級別增長。所有修改都傾向於破壞系統的架構，增加了系統的混亂程度。用在修復原有設計上瑕疵的工作量越來越少，而早期維護活動本身的漏洞所引起修復工作越來越多。隨著時間的推移，系統變得越來越無序，修復工作遲早會失去根基。每一步前進都伴隨著一步後退。儘管理論上系統一直可用，但實際上，整個系統已經面目全非，無法再成爲下一步進展的基礎。而且，機器在變化，配置在變化，用戶的需求在變化，所以現實系統不可能永遠可用。嶄新的、基於原有系統的重新設計是完全必要的。

通過對統計模型的研究，關於軟體系統，Belady和Lehman得到了更具普遍意義、爲所有經驗支持的結論。正如Pascal. C. S. Lewis所敏銳指出的：

這正是歷史的關鍵。使用卓越的能源——構建文明——成立傑出的機構，但是每次總會出現問題。一些致命的缺陷會將自私和殘酷的人帶到塔尖，接著一切開始滑落，回到到痛苦和墮落。實際上，機器失靈了。看上去，就好像是機器正常啓動，跑了幾步，然後垮掉了[7]。

系統軟體發展是減少混亂度（減少熵）的過程，所以它本身是處於亞穩態的。軟體維護是提高混亂度（增加熵）的過程，即使是最熟練的軟體維護工作，也只是放緩了系統退化到非穩態的進程。

# 幹將莫邪（*Sharp Tools*）

巧匠因爲他的工具而出名。

- 諺語

A good workman is known by his tools.

*- PROVERB*

就工具而言，即使是現在，很多軟體專案仍然像一家五金店。每個骨幹人員都仔細地保管自己工作生涯中搜集的一套工具集，這些工具成爲個人技能的直觀證明。正是如此，每個編程人員也保留著編輯器、排序、記憶體資訊轉儲、磁片實用程式等工具。

這種方法對軟體專案來說是愚蠢的。首先，專案的關鍵問題是溝通，個性化的工具妨礙——而不是促進溝通。其次，當機器和語言發生變化時，技術也會隨之變化，所有工具的生命週期是很短的。毫無疑問，開發和維護公共的通用編程工具的效率更高。

不過，僅有通用工具是不夠的。專業需要和個人偏好同樣需要很多專業工具。所以在前面關於軟體發展隊伍的討論中，我建議爲每個團隊配備一名工具管理人員。這個角色管理所有通用工具，能指導他的客戶－老闆使用工具。同時，他還能編制老闆需要的專業工具。

因此，專案經理應該制訂一套策略，並爲通用工具的開發分配資源。與此同時，他還必須意識到專業工具的需求，對這類工具不能吝嗇人力和物力——這種企圖的危害非常隱蔽。可能有人會覺得，將所有分散的人員集結起來，形成一個公共的工具小組，會有更高的效率。但實際上卻不是這樣。

專案經理必須考慮、計畫、組織的工具到底有哪些呢？首先是電腦設施。它需要硬體和使用安排策略；它需要作業系統，提供服務的方式必須明瞭；它需要語言，語言的使用方針必須明確；然後是實用程式、調試輔助程式、測試用例生成工具和處理文檔的字處理系統。接下面我們逐一討論它們[1]。

## 目的機器

機器支援可以有效地劃分成目的機器和輔助機器。目的機器是軟體所服務的物件，程式必須在該機器上進行最後測試。輔助機器是那些在開發系統中提供服務的機器。如果是在爲原有

的機型開發作業系統，則該機器不僅充當目的機器的角色，同時也作爲輔助機器。

目的機器的類型有哪些？團隊開發的監督程式或其他系統核心軟體當然需要它們自己的機器。目的機器系統會需要若干操作員和一兩個系統編程人員，以保證機器上的標準支援是即時更新和即時可用的。

如果還需要其他的機器，那麼將是一件很古怪的東西——運行速度不必非常快，但至少要若干百萬位元組的主存，百百萬位元組的線上硬碟和終端。字元型終端即可滿足要求，但是它必須比15字元/每分的打字機速度要快。大容量記憶體可以進行進程覆蓋（overlay）和功能測試之後的剪裁工作，從而極大地提高生產率。

另外，還需要配備調試機器或者軟體。這樣，在調試過程中，所有類型的程式參數可以被自動計數和測量。例如，記憶體使用模式是非常強大的診斷措施，能查出程式中不可思議的行爲或者性能意外下降的原因。

計畫安排。當目的機器剛剛被研製，或者當它的第一個作業系統被開發時，機器時間是非常匱乏的，時間的調度安排成了主要問題。目的機器時間需求具有特別的增長曲線。在OS/360開發中，我們有很好的System/360模擬器和其他的輔助設施，並根據以前的經驗，我們計畫出System/360的使用時間（小時數），向製造商提前預定了機器。不過，起初它們日復一日地處於空閒狀態。突然有一天，所有16個系統全部上線，這時資源配給成了嚴重問題。實際使用情況如圖12.1所示。每個人在同一時間，開始調試自己的第一個元件，然後團隊大多數成員持續地進行某些調試工作。

我們集中了所有的機器和磁帶庫，並組建了一個富有經驗的專業團隊來操作它們。爲了最大限度地利用S/360的時間，我們在任何系統空閒和可能的時間裏，以批次處理方式運行所有運算任務。我們嘗試了每天運行四次（周轉時間爲兩個半小時），而實際要求的周轉時間爲四小時。我們使用了一台帶有終端的1401輔助機器來進行調度，跟蹤成千上萬的任務，監督時間週期。

Model40的每月使用小時數

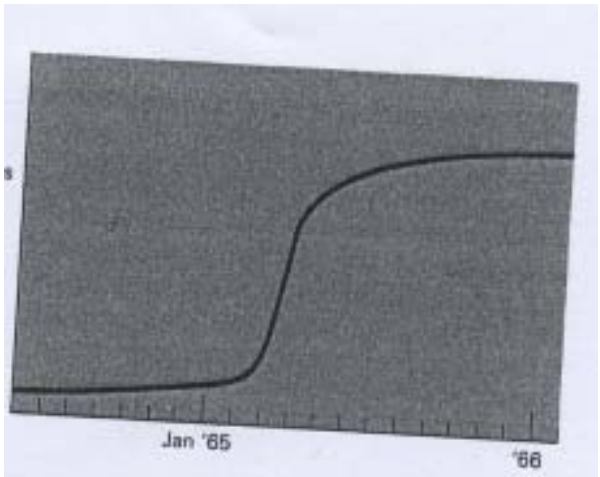



圖12.1：目的機器使用的增長曲線

但是整個開發隊伍實在是過度運轉了。在經過了幾個月的緩慢周轉、相互指責、極度痛苦之後，我們開始把機器時間分配成連續的塊。例如，整個從事排序工作的15人小組，會得到系統4至6小時的使用時間塊，由他們自己決定如何使用。即使沒有安排，其他人也不能使用機器資源。

這種方式，是一種更好的分配和安排方法。儘管機器的利用程度可能會有些降低（常常不是這樣），生產率卻提高了。上述小組中的每個人，6小時中連續10次操作的生產率，比間隔3小時的10次操作要高許多，因爲持續的精力集中能減少思考時間。在這樣的衝刺之後，提出下一個時間塊要求之前，小組通常需要一到兩天的時間來從事書面文檔工作。並且，通常3人左右的小組能卓有成效地安排和共用時間塊。在調試新作業系統時，這似乎是一種使用目的機器的最好方法。

上述方法儘管沒有在任何理論中被提及，在實際情況中卻一直如此。另外，同天文工作者一樣，系統調試總是夜班性質的工作。二十年前，當所有機房負責人在家中安睡時，我正工作在701上。三代機器過去了，技術完全改變了，作業系統出現了，然而大家喜好的工作方式沒有改變。這種工作方式得以延續，是因爲它的生產率最高。現在，人們已開始認識到它的生產力，並且敞開地接受這種富有成效的實踐。

## 輔助機器和資料服務

仿真裝置。如果目的機器是新產品，則需要一個目的機器的邏輯仿真裝置。這樣，在生產出新機器之前，就有輔助的調試平臺可供使用。同樣重要的是——即使在新機器出現之後，仿真裝置仍然可以提供可靠的調試平臺。

可靠並不等於精確。在某些方面，仿真機器肯定無法精確地達到與新型機器一致的實現。但是至少在一段時間內，它的實現是穩定的，新硬體就不會。

現在，我們已經習慣於電腦硬體自始至終能正常工作。除非程式開發人員發現相同運算在運行時會產生不一致的結果，否則出錯時，他都會被建議去檢查自己代碼中的錯誤，而不是去懷疑他的運行平臺。

這樣的經驗，對於支援新型機器的編程工作來說，是不好的。實驗室研製和試製的模型產品和早期硬體不會像定義的那樣運行，不會穩定工作，甚至每天都不會一樣。當一些缺陷被發現時，所有的機器拷貝，包括軟體編程小組所使用的，都會發生修改。這種飄忽不定的開發基礎實在是夠糟的。而硬體失敗，通常是間歇性的，導致情況更加惡劣。不確定性是所有情況中最糟糕的，因爲它剝奪了開發人員查找bug的動力——可能根本就沒有問題。所以，一套運行在穩定平臺上的可靠仿真裝置，提供了遠大於我們所期望的功用。

編譯器和彙編平臺。出於同樣的原因，編譯器和彙編軟體需要運行在可靠的輔助平臺上，爲目的機器編譯目標代碼。接著，可以在模擬器上立刻開始後續的調試。

高階語言的編程開發中，在目的機器上開始全面測試目標代碼之前，編譯器可以在輔助機器上完成很多目標代碼的調試和測試工作。這爲直接運行提供了支援，而不僅僅是穩定機器上的仿真結果。

程式庫和管理。在OS/360開發中，一個非常成功的重要輔助機器應用是維護程式庫。該系統由W. R. Crowley帶領開發，連接兩台7010機器，共用一個很大的磁片資料庫。7010同時還提供System/360組合語言程式。所有經過測試或者正在測試的代碼都保存在該庫中，包括源代碼和彙編裝載模組。這個庫實際上劃分成不同訪問規則下的子庫。

首先，每個組或者編程人員分配了一個區域，用來存放他的程式拷貝、測試用例以及單元測試需要的測試輔助常式和資料。在這個開發庫（playpen）中，不存在任何限制開發人員的規定。他可以自由處置自己的程式，他是它們的擁有者。

當開發人員準備將軟體單元集成到更大的部分時，他向集成經理提交一份拷貝，後者將拷貝放置在系統集成子庫中。此時，原作者不可以再改變代碼，除非得到了集成經理的批准。當系統合併在一起時，集成經理開始進行所有的系統測試工作，識別和修補bug。

有時，系統的一個版本可能會被廣泛應用，它被提升到當前版本子庫。此時，這個拷貝是不可更改的，除非有重大缺陷。該版本可以用於所有新模組的集成和測試。7010上的一個程式目錄對每個模組的每個版本進行跟蹤，包括它的狀態、用途和變更。

這有兩個重要的理念。首先是受控，即程式的拷貝屬於經理，他可以獨立地授權程式的變更。其次是使發佈的進展變得正式，以及開發庫（playpen）與集成、發佈的正式分離。

在我看來，這是OS/360工作中最優秀的成果之一。它實際上是管理技術的一部分，很多大型的專案都獨立地發展了這些技術[2]，包括Bell試驗室、ICL、劍橋大學等。它同樣適用于文檔，是一種不可缺少的技術。

編程工具。隨著調試技術的出現，舊方法的使用減少了，但並沒有消失。因此，還是需要記憶體傾印、原始檔案編輯、快照轉儲、甚至跟蹤等工具。

與之類似，一整套實用程式同樣是必要的，用來實現磁帶走帶、拷貝磁片、列印檔、更改目錄等工作。如果一開始就任命了專案的工具操作和維護人員，那麼這些工作可以一次完成，並且隨時處在待命狀態。

文檔系統。在所有的工具中，最能節省勞動力的，可能是運行在可靠平臺上的、電腦化的文本編輯系統。我們有一套使用非常方便的系統，由J. W. Franklin發明。沒有它，OS/360手冊的進度可能會遠遠落後，而且更加晦澀難懂。另外，對於6英尺的OS/360手冊，很多人認爲它表達的是一大堆口頭垃圾，巨大容量帶來了新的不理解問題——這種觀點有一些道理。

對此，我通過兩種途徑作出了反應。首先，OS/360的文檔規模是不可避免的，需要制訂仔細的閱讀計畫。如果選擇性地閱讀，則可以忽略大部分內容和省下大量時間。人們必須把OS/360的文檔看成是圖書館或者百科全書，而不是一系列強制閱讀的文章。

第二，它比那些刻畫了大多數編程系統特性的短篇文檔更加可取。不過，我也承認，手冊仍有某些需要大量改進的地方，經改進後文檔篇幅會大大減少。事實上，某些部分（“概念和設施”）已經被很好地改寫了。

性能仿真裝置。最好有一個。正如我們將在下章討論到的，徹底地開發一個。使用相同的自頂向下設計方法，來實現性能模擬器、邏輯仿真裝置和產品。盡可能早地開始這項工作，仔細地聽取“它們表達的意見”。

## 高階語言和互動式編程

在十年前的OS/360開發中，並沒有使用現在最重要的兩種系統編程工具。目前，它們也沒有得到廣泛應用，但是所有證據都證明它們的功效和適用。他們是（1）高階語言和（2）互動式編程。我確信只有懶散和惰性會妨礙它們的廣泛應用，技術上的困難很快就不再成爲藉口。

高階語言。使用高階語言的主要原因是生產率和調試速度。我們在前面已討論過生產率的問題（第8章）。其中，並沒有提到大量的數字論據，但是所體現出來的是整體提升，而不僅僅是部分增加。

調試上的改進來自下列事實——存在更少的bug，而且更容易查找。bug更少的原因，是因爲它避免在錯誤面前暴露所有級別的工作，這樣不但會造成語法上的錯誤，還會產生語義上的問題，如不當使用寄存器等。編譯器的診斷機制可以幫助找出這些類似的錯誤，更重要的是，它非常容易插入調試的快照。

就我而言，這些生產率和調試方面的優勢是勢不可擋的。我無法想像使用組合語言能方便地開發出系統軟體。

那麼，上述工具的傳統反對意見有哪些呢？這裏有三點：它無法完成我想做的事情；目標代碼過於龐大；目標代碼運行速度過慢。

就功能而言，我相信反對不再存在。所有證據都顯示了人們可以完成想做的事情，只是需要花費時間和精力找出如何做而已，這可能需要一些討人嫌的技巧[3,4]。

就空間而言，新的優化編譯器已非常令人滿意，並且將持續地改進。

就速度而言，經優化編譯器生成的代碼，比絕大多數程式師手寫代碼的效率要高。而且，在前者被全面測試之後，可以將其中的百分之一至五替換成手寫的代碼，這往往能解決速度方面的問題[5]。

系統編程需要什麼樣的高階語言呢？現在可供合理選擇的語言是PL/I[6]。它提供完整的功能集；它與作業系統環境相吻合；它有各種各樣的編譯器，一些是互動式的，一些速度很快，一些診斷性很好，另一些能產生優化程度很高的代碼。我自己覺得使用APL來解決演算法更快一些，然後，將它們翻譯成某個系統環境下的PL/I語言。

互動式編程。MIT的Multics專案的成果之一，是它對軟體編程系統開發的貢獻。在那些系統編程所關注的方面，Multics（以及後續系統，IBM的TSS）和其他互動式電腦系統在概念上有很大的不同：多個級別上資料和程式的共用和保護，可延伸的庫管理，以及協助終端用戶共同開發的設施。我確信在某些應用上，批次處理系統決不會被互動式系統所取代。但是，我認爲Multics小組是互動式系統開發上最具有說服力的成功案例。

然而，目前還沒有非常明顯的證據來證明這些功能強大的工具的效力。正如人們所普遍認識的那樣，調試是系統編程中很慢和較困難的部分，而漫長的調試周轉時間是調試的禍根。就這一點而言，互動式編程的邏輯合理性是勿庸置疑的[7]。

另外，從很多採用這種方式了開發小型系統和系統某個部分的人那裏，我們聽到了很多好的證據。我唯一見到的關於大型編程系統開發方面的數位，來自Bell實驗室John Harr的論文。它們如圖12.2所示。這些數字分別反映了代碼編寫、彙編裝配和程式調試的情況。第一個大部分是控制程式；其他三個則是語言解釋、編輯等程式。Harr的資料表明了系統軟體發展中，互動式編程的生產率至少是原來的兩倍[8]。

| 程式 | 規模 | 批次處理（B）或互動式（C） | 指令/人年 |
|---|---|---|---|
| ESS代碼 | 800,000 | B | 500-1000 |
| 7094 ESS支持 | 120,000 | B | 2100-3400 |
| 360 ESS支持 | 32,000 | C | 8000 |
| 360 ESS支持 | 8,300 | B | 4000 |

圖12.2：批次處理和互動式編程生產率的對比

由於遠端鍵盤終端無法用於記憶體傾印的調試，大多數互動式工具的有效使用需要採用高階語言來進行開發。有了高階語言，可以很容易地修改代碼和選擇性地列印結果。實際上，它們組成了一對強大的工具。

# 整體部分（*The Whole and the Parts*）

我能召喚遙遠的精靈。

那又怎麼樣，我也可以，誰都可以，問題是你真的召喚的時候，它們會來嗎？

- 莎士比亞，《亨利四世》，第一部分

I can call spirits from the vasty deep.

Why, so can I, or so can any man; but will they come when you do call for them?

*- SHAKESPEARE, KING HENRY IV, Part I*

和古老的神話裏一樣，現代神話裏也總有一些愛吹噓的人：“我可以編寫控制航空貨運、攔截彈道導彈、管理銀行帳戶、控制生產線的系統。”對這些人，回答很簡單，“我也可以，任何人都可以，但是其他人成功了嗎？”

如何開發一個可以運行的系統？如何測試系統？如何將經過測試的一系列構件集成到已測試過、可以依賴的系統？對這些問題，我們以前或多或少地提到了一些方法，現在就來更加系統地考慮一下。

## 剔除**bug**的設計

防範bug的定義。系統各個組成部分的開發者都會做出一些假設，而這些假設之間的不匹配，是大多數致命和難以察覺的bug的主要來源。第4、5、6章所討論的獲取概念完整性的途徑，就是直接面對這些問題。簡言之，產品的概念完整性在使它易於使用的同時，也使開發更容易進行以及bug更不容易產生。

上述方法所意味的詳盡體系結構設計正是出於這個目的。Bell實驗室安全監控系統專案的V.A.Vyssotsky提出，“關鍵的工作是產品定義。許許多多的失敗完全源於那些產品未精確定義的地方。[1]”細緻的功能定義、詳細的規格說明、規範化的功能描述說明以及這些方法的實施，大大減少了系統中必須查找的bug數量。

測試規格說明。在編寫任何代碼之前，規格說明必須提交給測試小組，以詳細地檢查說明的完整性和明確性。如同Vyssotsky所述，開發人員自己不會完成這項工作：“他們不會告訴你他們不懂。相反，他們樂於自己摸索出解決問題和澄清疑惑的辦法。”

自頂向下的設計。在1971年的一篇論文中，Niklaus Wirth把一種被很多最優秀的編程人員所使用的設計流程[2]形式化。儘管他的理念是爲了程式設計，同樣也完全適用於複雜系統的軟體發展設計。他將程式開發劃分成體系結構設計、設計實現和物理編碼實現，每個步驟可以使用自頂向下的方法很好地實現。

簡言之，Wirth的流程將設計看成一系列精化步驟。開始是勾畫出能得到主要結果的，但比較粗略的任務定義和大概的解決方案。然後，對該定義和方案進行細緻的檢查，以判斷結果與期望之間的差距。同時，將上述步驟的解決方案，在更細的步驟中進行分解，每一項任務定義的精化變成了解決方案中演算法的精化，後者還可能伴隨著資料表達方式的精化。

在這個過程中，當識別出解決方案或者資料的模組時，對這些模組的進一步細化可以和其他的工作獨立，而模組的大小程度決定了程式的適用性和可變化的程度。

Wirth主張在每個步驟中，盡可能使用級別較高的表達方法來表現概念和隱藏細節，除非有必要進行進一步的細化。

好的自頂向下設計從幾個方面避免了bug。首先，清晰的結構和表達方式更容易對需求和模組功能進行精確的描述。其次，模組分割和模組獨立性避免了系統級的bug。另外，細節的隱藏使結構上的缺陷更加容易識別。第四，設計在每個精化步驟的層次上是可以測試的，所以測試可以儘早開始，並且每個步驟的重點可以放在合適的級別上。

當遇到一些意想不到的問題時，按部就班的流程並不意味著步驟不能反過來，直到推翻頂層設計，重新開始整個過程。實際上，這種情況經常發生。至少，它讓我們更加清楚在什麼時候和爲什麼拋棄了某個臃腫的設計，並重新開始。一些糟糕的系統往往就是試圖挽救一個基礎很差的設計，而對它添加了很多表面裝飾般的補丁。自頂向下的方法減少了這樣的企圖。

我確信在十年內，自頂向下進行設計將會是最重要的新型形式化軟體發展方法。

結構化編程。另外一系列減少bug數量的新方法很大程度上來自Dijkstra[3]。Bohm和Jacopini的爲其提供了理論證明[4]。

基本上，該方法所設計程式的控制結構，僅包含語句形式的迴圈結構，例如DO WHILE，以及IF...THEN...ELSE的條件判斷結構，而具體的條件部分在IF...THEN...ELSE後的花括弧中描述。Bohm和Jacopini展示了這些結構在理論上是可以證明的。而Dijkstra認爲另外一種方法，即通過GO TO不加限制的分支跳轉，會產生導致自身邏輯錯誤的結構。

這種方法的基本理念非常優秀，但仍有人提出了一些反面的意見。一些附加的控制結構非常有效，例如，在多個條件下的多路分支（CASE、SWITCH語句），異常跳轉等（GO TO ABNORMAL END）。此外，關於完全避免GO TO語句的說法顯得有些教條主義，而且似乎有些吹毛求疵。

關鍵的地方和構建無bug程式的核心，是把系統的結構作爲控制結構來考慮，而不是獨立的跳轉語句。這種思考方法是我們在程式設計發展史上向前邁出的一大步。

## 構件單元調試

程式調試過程在過去的二十年中有過很多反復，甚至在某些方面，它們又回到了出發的起點。整個調試過程有四個步驟，跟隨這個過程來檢驗每個步驟各自的動機是一件很有趣的事情。

本機調試。早期的機器的輸入和輸出設備很差，延遲也很長。典型的情況是，機器採用紙帶或者磁帶的方式來讀寫，採用離線設備來完成磁帶的準備和列印工作。這使得磁帶輸入/輸出對於調試是不可忍受的。因此，在一次機器交互會話中會盡可能多地包含試驗性操作。

在那種情況下，程式師仔細地設計他的調試過程——計畫停止的地點，檢驗記憶體的位

置，需要檢查的東西以及如果沒有預期結果時的對策。花費在編寫調試程式上的時間，可能是程式編制時間的一半。

這個步驟的“重大罪過”是在沒有把程式劃分成測試段，並對執行終止位置進行計畫的前提下，粗暴地按下“開始（START）”。

記憶體傾印。本機調試非常有效。在兩小時的交互過程中可能會發現一打問題，但是電腦的資源非常匱乏，成本很高。想像一下計算機時間的浪費，那實在是一件可怕的事情。

因此，當使用線上高速印表機時，測試技術發生了變化。某人持續地運行程式，直到某個檢測失敗，這時所有的記憶體都被轉儲。接著，他將開始艱苦的桌面工作，考慮每個記憶體位置的內容。桌面工作的時間和本機調試並沒有太大的不同，但它的方式比以前更爲含混，並且發生在測試執行之後。特定用戶調試用的時間更長，因爲測試依賴於批次處理的週期。總之，整個過程的設計是爲了減少電腦的使用時間，從而盡可能滿足更多的用戶。

快照。採用記憶體傾印技術的機器往往配有2000～4000個字（word雙位元組），或者8K～16K位元組的記憶體。但是，隨著記憶體的規模不斷增長，對整個記憶體都進行轉儲變得不大可能。因此，人們開發了有選擇的轉儲、選擇性跟蹤和將快照插入程式的技術。OS/360 TESTRAN允許將快照插入程式，無需重新彙編和編譯，它是快照技術方向的終極產品。

互動式調試。1959年，Codd和他的同事[5]以及Strachey[6]都發表了關於協助分時調試工作的論文，提出了一種兼有本機調試方式即時性和批次處理調試高效使用率的方法。電腦將多個程式載入到記憶體中準備運行，被調試的程式和一個只能由程式控制的終端相關聯，由監督調度程式控制調試過程。當終端前的編程人員停止程式，檢查進展情況或者進行修改時，監督程式可以運行其他程式，從而保證了機器的使用率。

Codd的多道程序系統已經開發出來，但是它的重點是通過有效地利用輸入/輸出來提高吞吐量，並沒有實現互動式的調試。Strachy的想法不斷得到改進，終於在1963年由MIT的Corbato和他的同事在7090的實驗性系統上實現[7]。這個開發結果導致了MULTICS、TSS和現在其他分時系統的出現。

在最初使用的本機調試方法和現在的互動式調試方法之間，用戶可以感覺到的主要差異是工具性軟體、調度監控程序和其他相關語言解釋編譯器的出現。而現在，已經可以用高階語言來編程和調試，高效的編輯工具使修改和快照更爲容易。

互動式調試擁有和本機調試一樣的操作即時性，但前者並沒有象後者要求的那樣，在調試過程中要預先進行計畫。在某種程度上，像本機調試那樣的預先計畫顯得並不是很必要，因爲在調試人員停頓和思考時，電腦的時間並沒有被浪費。

不過，Gold實驗得到一個有趣的結果，這個結果顯示在每次調試會話中，第一次交互取得的工作進展是後續交互的三倍[8]。這強烈地暗示著，由於缺乏對調試會話的計畫，我們沒有發掘互動式調試的潛力，原有本機調試技術中那段高效率的時間消失了。

我發現對良好終端系統的正確使用，往往要求每兩小時的終端會話對應於兩小時的桌面工

作。一半時間用於上次會話的清理工作：更新調試日誌，把更新後的程式列表加入到專案檔夾中，研究和解釋調試中出現的奇怪現象。剩餘一半時間用於準備：爲下一次操作設計詳細的測試，進行計畫的變更和改進。如果沒有這樣的計畫，則很難保持兩個小時的高生產率；而沒有事後的清理工作，則很難保證後續終端會話的系統化和持續推進。

測試用例。關於實際調試過程和測試用例的設計，Grunberger提出了特別好的對策[9]，在其他的文章中，也有較爲簡便的方法[10,11]。

## 系統集成調試

軟體系統開發過程中出乎意料的困難部分是系統集成測試。前面我已經討論了一些困難產生和困難不確定的原因。其中需要再次確認的兩件事是：系統調試花費的時間會比預料的更長，需要一種完備系統化和可計畫的方法來降低它的困難程度。下面來看看這樣的方法所包括的內容[12]。

使用經過調試的構件單元。儘管並不是普遍的實際情況——不過通常的看法是——系統集成調試要求在每個部分都能正常運行之後開始。

實際工作中，存在著與上面看法不同的兩種情況。一種是“合在一起嘗試”的方法，這種方法似乎是基於這樣的觀點：除了構件單元上的bug之外，還存在系統bug（如介面），越早將各個部分合攏，系統bug出現得越早。另一種觀念則沒有這麼複雜：使用系統的各個部分進行相互測試，避免了大量測試輔助平臺的搭建工作。這兩種情況顯然都是合理的，但經驗顯示它們並不完全正確——使用完好的、經過調試的構件，能比搭建測試平臺和進行全面的構件單元測試節省更多的時間。

更微妙的一種方法是“文檔化的bug”。它申明構件單元所有的缺陷已經被發現，還沒有被修復，但已經做好了系統調試的準備。在系統測試期間，依照該理論，測試人員知道這些缺陷造成的後果，從而可以忽略它們，將注意力集中在新出現的問題上。

但是所有這些良好的願望只是試圖爲結果的偏離尋找一些合理理由。實際上，調試人員並不瞭解bug引起的所有後果；不過，如果系統比較簡單，系統測試倒不會太困難。另外，對文檔記錄bug的修復工作本身會注入未知的問題，接下來的系統測試會令人困惑。

搭建充分的測試平臺。這裏所說的輔助測試平臺，指的是供調試使用的所有程式和資料，它們不會整合到最終產品中。測試平臺可能會有相當於測試物件一半的代碼量，但這是合乎情理的。

一種測試輔助的形式是僞構件（dummy component），它僅僅由介面和可能的僞資料或者一些小的測試用例組成。例如，系統包含某種排序程式，但該程式還未完成，這時其他部分的測試可以通過僞構件來實現，該構件讀入輸入資料，對資料格式進行校驗，輸出格式良好、但沒有實際意義的有序數據以供使用。

另一種形式是微縮檔（miniature file）。很常見的一類bug來自對磁帶和磁片檔格式的

錯誤理解。所以，創建一個僅包含典型記錄，但涵蓋全部描述的小型檔是非常值得的。

微縮文件的特例是僞文件（dummy file），實際上並不常見。不過OS/360任務控制語言提供了這種功能，對於構件單元調試非常有用。

還有一種方式是輔助程式（auxiliary program）。用來測試資料發生器、特殊的列印輸出、交叉引用表分析等，這些都是需要另外開發的專用輔助工具的例子[13]。

控制變更。對測試期間進行嚴密控制是硬體調試中一項令人印象深刻的技術，它同樣適用於軟體系統。

首先，必須有人負責。他必須控制和負責各個構件單元的變更或者版本之間的替換。

接著，就像前面所討論的，必須存在系統的受控拷貝：一個是供構件單元測試使用的最終鎖定版本；一個是測試版本的拷貝，用來進行缺陷的修復；以及一個安全版本，其他人員可以在該拷貝上工作，進行各自的程式開發工作，例如修復和擴展自己的模組和子系統等。

在System/360工程模型中，在一大堆常規的黃顏色電線中，常常可以不經意地看到紫色的電線束。在發現bug以後，我們會做兩件事情：設計快速修復電路，並安裝到系統，從而不會妨礙測試的繼續進行。這些更改過的接線使用紫色電線，看上去就像伸著一個受了傷的大拇指。我們需要把更改記錄到日誌中，同時，還要準備一份正式的變更文檔，並啓動設計自動化流程。最後，在電路圖或者黃色線路中會實現該設計的調整——更新相應的電路圖和接線表，以及開發一個新的電路板。現在，物理模型和電路圖重新吻合了，紫色的線束也就不再需要了。

軟體發展也需要用到“紫色線束”的手法。對於最後成爲產品的程式碼，它更迫切地需要進行嚴密控制和深層次的關注。上述技巧的關鍵因素是對變更和差異的記載，即在一個日誌中記錄所有的變更，而在源代碼中顯著標記快速補丁和正式修改之間的區別，正式修改是完備並經過測試的，而且需要文檔化。

一次添加一個構件。這樣做的好處同樣是顯而易見的，但是樂觀主義和惰性常常誘使我們破壞這個規則。因爲離散構件的添加需要調試僞程式和其他測試平臺，有很多工作要做。畢竟，可能我們不需要這些額外工作？可能不會出現什麼bug？

不！拒絕誘惑！這正是系統測試所關注的方面。我們必須假設系統中存在著許多錯誤，並需要計畫一個有序的過程把它們找出來。

注意必須擁有完整的測試用例，在添加了新構件之後，用它們來測試子系統。因爲那些原來可以在子系統上成功運行的用例，必須在現有系統上重新運行，對系統進行回歸測試。

階段（量子）化、定期變更。隨著專案的推進，系統構件的開發者會不時出現在我們面前，帶著他們工作的最新版本——更快、更卓越、更完整，或者公認bug更少的版本。將使用中的構件替換成新版本，仍然需要進行和構件添加一樣的系統化測試流程。這個時候通常已經具備了更完整有效的測試用例，因此測試時間往往會減少很多。

專案中，其他開發團隊會使用經過測試的最新集成系統，作爲調試自己程式的平臺。測試平臺的修改，會阻礙他們的工作。當然，這是必須的。但是，變更必須被階段化，並且定期發佈。這樣，每個用戶擁有穩定的生產週期，其中穿插著測試平臺的改變。這種方法比持續波動所造成的混亂無序要好一些。

Lehman和Belady出示了證據，階段（量子）要麼很大，間隔很寬；要麼小而頻繁[14]。根據他們的模型，小而頻繁的階段很容易變得不穩定，我的經驗也同樣證實了這一點——因此我決不會在實踐中冒險採用後一種策略。

量子（階段）化變更方法非常優美地容納了紫色線束技術：直到下一次系統構件的定期發佈之前，都一直使用快速補丁；而在當前的發佈中，把已經通過測試並進行了文檔化的修補措施整合到系統平臺。

# 禍起蕭牆（*Hatching a Catastrophe*）

帶來壞消息的人不受歡迎。

- 索福克裏斯

項目是怎樣延遲了整整一年的時間？…一次一天。

*None love the bearer of bad news.*

*- SOPHOCLES*

*How does a project get to be a year late? ... One day at a time.*

當人們聽到某個專案的進度發生了災難性偏離時，可能會認爲專案一定是遭受了一系列重大災難。然而，通常災禍來自白蟻的肆虐，而不是龍捲風的侵襲。同樣，專案進度經常以一種難以察覺，但是殘酷無情的方式慢慢落後。實際上，重大災害是比較容易處理的，它往往和重大的壓力、徹底的重組、新技術的出現有關，整個專案組通常可以應付自如。

但是一天一天的進度落後是難以識別、不容易防範和難以彌補的。昨天，某個關鍵人員生病了，無法召開某個會議。今天，由於雷擊打壞了公司的供電變壓器，所有機器無法啓動。明天，因爲工廠磁片供貨延遲了一周，磁片常式的測試無法進行。下雪、應急任務、私人問題、同顧客的緊急會議、管理人員檢查——這個列表可以不斷地延長。每件事都只會將某項活動延

遲半天或者一天，但是整個進度開始落後了，儘管每次只有一點點。

## 里程碑還是沉重的負擔？

如何根據一個嚴格的進度表來控制專案？第一個步驟是制訂進度表。進度表上的每一件事，被稱爲“里程碑”，它們都有一個日期。選擇日期是一個估計技術上的問題，在前面已經討論過，它在很大程度上依賴以往的經驗。

里程碑的選擇只有一個原則，那就是，里程碑必須是具體的、特定的、可度量的事件，能夠進行清晰定義。以下是一些反面的例子，例如編碼，在代碼編寫時間達到一半的時候就已經“90%完成”了；調試在大多時候都是“99%完成”的；“計畫完畢”是任何人只要願意，就可以聲明的事件[1]。

然而，具體的里程碑是百分之百的事件。“結構師和實現人員簽字認可的規格說明”，“100%源代碼編制完成，紙帶打孔完成並輸入到磁片庫”，“測試通過了所有的測試用例”。這些切實的里程碑澄清了那些劃分得比較模糊的階段——計畫、編碼、調試。

里程碑有明顯邊界和沒有歧義，比它容易被老闆核實更爲重要。如果里程碑定義得非常明確，以致于無法自欺欺人時，很少有人會就里程碑的進展弄虛作假。但是如果里程碑很模糊，老闆就常常會得到一份與實際情況不符的報告。畢竟，沒有人願意承受壞消息。這種做法只是爲了起到緩和的作用，並沒有任何蓄意的欺騙。

對於大型開發專案中的估計行爲，政府的承包商做了兩項有趣的研究。研究結果顯示：

1．如果在某項活動開始之前就著手估計，並且每兩周進行一次仔細的修訂。這樣，隨著開始時間的臨近，無論最後情況會變得如何的糟糕，它都不會有太大的變化。

2．活動期間，對時間長短的過高估計，會隨著活動的進行持續下降。

3．過低估計在活動中不會有太大的變化，一直到計畫的結束日期之前大約三周左右。

好的里程碑對團隊來說實際上是一項服務，可以用來向專案經理提出合理要求的一項服務，而不確切的里程碑是難以處理的負擔。當里程碑沒有正確反映損失的時間，並對人們形成誤導，以致事態無法挽回的時候，它會徹底碾碎小組的士氣。慢性進度偏離同樣也是士氣殺手。

## “其他的部分反正會落後”

進度落後了一天，那又怎麼樣呢？誰會關心一天的滯後？我們可以跟上進度。何況，和我們有關的其他部分已經落後了。

棒球隊隊長知道，進取這種心理素質，是很多優秀隊員和團隊不可缺少的。它表現爲“要求

跑得更快”，“要求移動得更加迅速”，“更加努力嘗試”。對軟體發展隊伍，進取同樣是非常必要的。進取提供了緩衝和儲備，使開發隊伍能夠處理常規的異常事件，可以預計和防止小的災禍。而對任務進行計算和對工作量進行度量，會對進取超前會造成一些消極的影響——這時，人們往往會比較樂觀地放緩工作節奏。就這一點來說，它們是令人掃興的事情。不過，如同我們看到的，必須關心每一天的滯後，它們是大災禍的基本組成元素。

並不是每一天的滯後都等於災難。儘管會如上文所述，事先估計會給工作進度的超前帶來影響，但對活動的一些計算和考慮還是必要的。那麼，如何判斷哪些偏離是關鍵的呢？只有採用PERT或者關鍵路徑技術才能判斷。它顯示誰需要什麼樣的東西，誰位於關鍵路徑上，他的工作滯後會影響最終的完成日期。另外，它還指出一個任務在成爲關鍵路徑時，可以落後的時間。

嚴格地說，PERT技術是關鍵路徑計畫的細化，如果使用PERT圖，它需要對每個事件估計三次，每次對應於滿足估計日期的不同可能性。我覺得不值得爲這樣的精化產生額外的工作量，但爲了方便，我把任何關鍵路徑法都稱爲PERT圖。

PERT的準備工作是PERT圖使用中最有價值的部分。它包括整個網狀結構的展開、任務之間依賴關係的識別、各個任務鏈的估計。這些都要求在項目早期進行非常專業的計畫。第一份PERT圖總是很恐怖的，不過人們總是不斷地進行努力，運用才智制訂下一份PERT圖。

隨著項目的推進，PERT圖爲前面那個洩氣的藉口，“其他的部分反正會落後”，提供了答案。它展示某人爲了使自己的工作遠離關鍵路徑，需要超前多少，也建議了補償其他部分失去的時間的方法。

## 地毯的下麵

當一線經理發現自己的隊伍出現了計畫偏離時，他肯定不會馬上趕到老闆那裏去彙報這個令人沮喪的消息。團隊可以彌補進度偏差，他可以想出應對方法或者重新安排進度以解決問題，爲什麼要去麻煩老闆呢？從這個角度來看，好像還不錯。解決這類問題的確是一線經理的職責。老闆已經有很多需要處理的真正的煩心事了，他不想被更多的問題打攪。因此，所有的污垢都被隱藏在地毯之下。

但是每個老闆都需要兩種資訊：需要採取行動的計畫方面的問題，用來進行分析的狀態資料[3]。出於這個目的，他需要瞭解所有開發隊伍的情況，但得到狀態的真相是很困難的。

一線經理的利益和老闆的利益是內在衝突的。一線經理擔心如果彙報了問題，老闆會採取行動,這些行動會取代經理的作用，降低自己的威信，搞亂了其他計畫。所以，只要專案經理認爲自己可以獨立解決問題，他就不會告訴老闆。

有兩種掀開毯子把污垢展現在老闆面前的方法，它們必須都被採用。一種是減少角色衝突和鼓勵狀態共用，另一種是猛地拉開地毯。

減少角色的衝突。首先老闆必須區別行動資訊和狀態資訊。他必須規範自己，不對專案經理可以解決的問題做出反應，並且決不在檢查狀態報告的時候做安排。我曾經認識一個老闆，

他總是在狀態報告的第一個段落結束之前，拿起電話發號施令。這樣的反應肯定壓制資訊的完全公開。

不過，當專案經理瞭解到老闆收到專案報告之後不會驚慌，或者不會越俎代庖時，他就逐漸會提交真實的評估結果。

如果老闆把會見、評審、會議明顯標記爲狀態檢查（status-meeting）和問題－行動（problem-action）會議，並且相應控制自己的行爲，這對整個過程會很有幫助。當然，事態發展到無法控制時，狀態檢查會議會演變成問題－行動會議。不過，至少每個人知道"當時遊戲的分數是多少"，老闆在接過"皮球"之前也會三思。

猛地拉開地毯。不論協作與否，擁有能瞭解狀態真相的評審機制是必要的。PERT圖以及頻繁的里程碑是這種評審的基礎。大型專案中，可能需要每週對某些部分進行評審，大約一個月左右進行整體評審。

有報告顯示關鍵的文檔是里程碑和實際的完成情況。圖14.1是上述報告中的一段摘錄。它顯示了一些問題：手冊（SLR）的批准時間有所衝突，其中一個的時間比獨立產品測試（Alpha）的開始時間還要遲。這樣一份報告將作爲2月1號會議的議程，使得每個人都知道問題的所在，而產品構件經理應準備解釋延遲的原因，什麼時候結束，採取的步驟和需要的任何幫助——老闆提供的，或者是其他小組間接提供的。

SYSTEM/360 SUMMARY STATUS REPORT
OS/360 LANGUAGE PROCESSORS + SERVICE PROGRAMS
AS OF FEBRUARY 01,1965

A=APPROVAL
C=COMPLETED

*=REVISED PLANNED DATE
NE=NOT ESTABLISHED

| PROJECT | LOCATION | COMMITMNT ANNOUNCE RELEASE | OBJECTIVE AVAILABLE APPROVED | SPECS AVAILABLE APPROVED | SRL AVAILABLE APPROVED | ALPHA TEST ENTRY EXIT | COMP TEST START COMPLETE | SYS TEST START COMPLETE | BULLETIN AVAILABLE APPROVED | BETA TEST ENTRY EXIT |
|---|---|---|---|---|---|---|---|---|---|---|
| OPERATING SYSTEM | | | | | | | | | | |
| 12K DESIGN LEVEL (E) | | | | | | | | | | |
| ASSEMBLY | SAN JOSE | 04/--/4 C<br>12/31/5 | 10/28/4 C | 10/13/4 C<br>01/11/5 | 11/13/4 C<br>11/18/4 A | 01/15/5 C<br>02/22/5 | | | | 09/01/5<br>11/30/5 |
| FORTRAN | POK | 04/--/4 C<br>12/31/5 | 10/28/4 C | 10/21/4 C<br>01/22/5 | 12/17/4 C<br>12/19/4 A | 01/15/5 C<br>02/22/5 | | | | 09/01/5<br>11/30/5 |
| COBOL | ENDICOTT | 04/--/4 C<br>12/31/5 | 10/28/4 C | 10/15/4 C<br>01/20/5 A | 11/17/4 C<br>12/08/4 A | 01/15/5 C<br>02/22/5 | | | | 09/01/5<br>11/30/5 |
| RPG | SAN JOSE | 04/--/4 C<br>12/31/5 | 10/28/4 C | 09/30/4 C<br>01/05/5 A | 12/02/4 C<br>01/18/5 A | 01/15/5 C<br>02/22/5 | | | | 09/01/5<br>11/30/5 |
| UTILITIES | TIME/LIFE | 04/--/4 C<br>12/31/5 | 06/24/4 C | | 11/20/4 C<br>11/30/4 A | | | | | 09/01/5<br>11/30/5 |
| SORT 1 | POK | 04/--/4 C<br>12/31/5 | 10/28/4 C | 10/19/4 C<br>01/11/5 | 11/12/4 C<br>11/30/4 A | 01/15/5 C<br>03/22/5 | | | | 09/01/5<br>11/30/5 |
| SORT 2 | POK | 04/--/4 C<br>06/33/6 | 10/28/4 C | 10/19/4 C<br>01/11/5 | 11/12/4 C<br>11/30/4 A | 01/15/5 C<br>03/22/5 | | | | 03/01/6<br>05/30/6 |
| 44K DESIGN LEVEL (F) | | | | | | | | | | |
| ASSEMBLY | SAN JOSE | 04/--/4 C<br>12/31/5 | 10/28/4 C | 10/13/4 C<br>01/11/5 | 11/13/4 C<br>11/18/4 A | 02/15/5<br>03/22/5 | | | | 09/01/5<br>11/30/5 |
| COBOL | TIME/LIFE | 04/--/4 C<br>06/33/6 | 10/28/4 C | 10/15/4 C<br>01/20/5 A | 11/17/4 C<br>12/08/4 A | 02/15/5<br>03/22/5 | | | | 03/01/6<br>05/30/6 |
| NPL | HURSLEY | 04/--/4 C<br>03/31/6 | 10/28/4 C | | | | | | | |
| 2250 | KINGSTON | 03/39/4 C<br>03/31/6 | 11/05/4 C | 12/08/4 C<br>01/04/5 | 01/12/5 C<br>01/29/5 | 01/04/5 C<br>01/29/5 | | | | 01/03/6<br>NE |
| 2280 | KINGSTON | 06/33/4 C<br>09/33/6 | 11/05/4 C | | | 04/01/5<br>04/30/5 | | | | 01/28/6<br>NE |
| 200K DESIGN LEVEL (H) | | | | | | | | | | |
| ASSEMBLY | TIME/LIFE | | 10/28/4 C | | | | | | | |
| FORTRAN | POK | 04/--/4 C<br>06/30/6 | 10/28/4 C | 10/16/4 C<br>01/11/5 | 11/11/4 C<br>12/10/4 A | 02/15/5<br>03/22/5 | | | | 03/01/6<br>05/30/6 |
| NPL | HURSLEY | 04/--/4 C<br>03/31/7 | 10/28/4 C | | | 07/--/5 | | | | 01/--/7 |
| NPL H | POK | 04/--/4 C | 03/30/4 C | | | 02/01/5<br>04/01/5 | | | | 10/15/5<br>12/15/5 |

注：

SYSTEM/360 SUMMARY STATUS REPORT －SYSTEM/360總結狀態報告

OS/360 LANGUAGE　PROCESSORS + SERVICE PROGAMS－OS/360語言處理器 + 服務程式

AS OF FEBRAURAY 01.1965－1965年2月1號

APPOVEL－批准

COMPLETED－完成

PROJECT－項目

LOCATION－地點

COMMITMNT ANNOUNCE RELEASE－計畫 發佈

OBJECTIVE AVAILABLE APPROVED－目標制訂 批准

SPECS AVAILABLE APPROVED－規格說明提交 批准

SRL AVAILABLE APPROVED－SRL提交 批准

ALPHA TEST ENTRY EXIT－ALPHA測試 進入 退出

COMP TEST START COMPLETE－單元測試 開始 結束

SYS TEST START COMPLETE－系統測試 開始 結束

BULLETIN AVAILABLE APPROVED－公告發佈 批准

BETA TEST ENTRY EXIT－BETA測試 進入 退出

圖14.1

Be11實驗室的V. Vyssotsky添加了以下的觀察意見：

我發現在里程碑報告中很容易記錄“計畫”和“估計”的日期。計畫日期是項目經理的工作產物，代表了經協調後的項目整體工作計畫，它是合理計畫之前的判斷。估計日期是最基層經理的工作產物，基層經理對所討論的工作有著深刻的瞭解，估計日期代表了在現有資源和已得到了作爲先決條件的必要輸入（或得到了相應的承諾）的情況下，基層經理對實際實現日期的最佳判斷。專案經理必須停止對這些日期的懷疑，而將重點放在使其更加精確上、以便得到沒有偏見的估計，而不是那些合乎心意的樂觀估計或者自我保護的保守估計。一旦它們在每個人的腦海中形成了清晰的印象，項目經理就可以預見到將來哪些地方如果他不採取任何措施，就會出現問題[4]。

PERT圖的準備工作是老闆和要向他進行彙報的經理們的職責。需要一個小組（一至三個人）來關注它的更新、修訂和報告，這個小組可以看作是老闆的延伸。對大型項目，這種計畫和控制（Plan and Contro1）小組的價值是非常可貴的。小組的職權僅限於向產品線經理詢問他們什麼時候設定或更改里程碑，以及里程碑是否被達到。計畫和控制小組處理所有的文字工作，因此產品線經理的負擔將會減到最少——僅僅需要作出決策。

我們擁有一個富有熱情的、有經驗的、熟練的計畫和控制小組。這個小組由A. M. Pietrasanta負責，他投入了大量創造天分來設計有效的、謙遜的控制方法。結果，我發現他的小組被廣爲尊重，而不僅僅是被容忍。對於這樣一個本來就十分敏感的角色而言，這的確是一個成功。

對計畫和控制職能進行適度的技術人力投資是非常值得讚賞的。它對專案的貢獻方式和直接開發軟體產品有很大的不同。計畫和控制小組作爲監督人員，明白地指出了不易察覺的延遲，並強調關鍵的因素。他們是早期預警系統，防止專案以一次一天的方式落後一年。

# 另外一面（*The other face*）

不瞭解，就無法真正擁有。

- 歌德

- 克雷布

*What we do not understand we do not possess.*

*- GOETHE*

*O give me commentators plain, Who with no deep researches vex the brain[jypan1]*

*- CRABBE*

電腦程式是從人傳遞到機器的一些資訊。爲了將人的意圖清晰地傳達給不會說話的機器，程式採用了嚴格的語法和嚴謹的定義。

但是書面的電腦程式還有其他的呈現面貌：向用戶訴說自己的"故事"。即使是完全開發給自己使用的程式，這種溝通仍然是必要的。因爲記憶衰退的規律會使用戶－作者失去對程式的瞭解，於是他不得不重拾自己勞動的各個細節。

公共應用程式的用戶在時間和空間上都遠離它們的作者，因此對這類程式，文檔的重要性更是不言而喻！對軟體編程產品來說，程式向用戶所呈現的面貌和提供給機器識別的內容同樣重要。

面對那些文檔"簡約"的程式，我們中的大多數人都不免曾經暗罵那些遠在他方的匿名作者。因此，一些人試圖向新人慢慢地灌輸文檔的重要性：旨在延長軟體的生命期、克服惰性和進度的壓力。但是，很多次嘗試都失敗了，我想很可能是由於我們使用了錯誤的方法。

Thomas J. Watson講述了他年輕時在紐約北部，剛開始做收銀機推銷員的經歷。他帶著一馬車的收銀機，滿懷熱情地動身了。他工作得非常勤奮，但是連一台收銀機也沒有賣出去。他很沮喪地向經理彙報了情況，銷售經理聽了一會兒，說道："幫我抬一些機器到馬車上，收緊韁繩，出發！"他們成功了。在接下來的客戶拜訪過程中，經理身體力行地演示了如何出售收銀機。事實證明，這個方法是可行的。

我曾經非常勤奮地給我的軟體工程師們舉辦了多年關於文檔必要性以及優秀文檔所應具備特點方面的講座，向他們講述——甚至是熱誠地向他們勸誡以上的觀點。不過，這些都行不通。我想他們知道如何正確地編寫文檔，卻缺乏工作的熱情。後來，我嘗試了向馬車上搬一些收銀機，以此演示如何完成這項工作。結果顯示，這種方法的效果要好得多。所以，文章剩餘部分將對那些說教之辭一筆帶過，而把重點放在"如何做（才能產生一篇優秀的文檔）上。

## 需要什麼樣的文檔

不同用戶需要不同級別的文檔。某些用戶僅僅偶爾使用程式，有些用戶必須依賴程式，還有一些用戶必須根據環境和目的的變動對程式進行修改。

使用程式。每個用戶都需要一段對程式進行描述的文字。可是大多數文檔只提供了很少的總結性內容，無法達到用戶要求，就像是描繪了樹木，形容了樹葉，但卻沒有一副森林的圖案。爲了得到一份有用的文字描述，就必須放慢腳步，穩妥地進行。

1．目的。主要的功能是什麼？開發程式的原因是什麼？

2．環境。程式運行在什麼樣的機器、硬體配置和作業系統上？

3．範圍。輸入的有效範圍是什麼？允許顯示的合法範圍是什麼？

4．實現功能和使用的演算法。精確地闡述它做了什麼。

5．輸入－輸出格式。必須是確切和完整的。

6．操作指令。包括控制臺及輸出內容中正常和異常結束的行爲。

7．選項。用戶的功能選項有哪些？如何在選項之間進行挑選？

8．運行時間。在指定的配置下，解決特定規模問題所需要的時間？

9．精度和校驗。期望結果的精確程度？如何進行精度的檢測？

一般來說，三、四頁紙常常就可以容納以上所有的資訊。不過往往需要特別注意的是表達的簡潔和精確。由於它包含了和軟體相關的基本決策，所以這份文檔的絕大部分需要在程式編制之前書寫。

驗證程式。除了程式的使用方法，還必須附帶一些程式正確運行的證明，即測試用例。

每一份發佈的程式拷貝應該包括一些可以例行運行的小測試用例，爲用戶提供信心——他擁有了一份可信賴的拷貝，並且正確地安裝到了機器上。

然後，需要得到更加全面的測試用例，在程式修改之後，進行常規運行。這些用例可以根據輸入資料的範圍劃分成三個部分。

1．針對遇到的大多數常規資料和程式主要功能進行測試的用例。它們是測試用例的主要組成部分。

2．數量相對較少的合法資料測試用例，對輸入資料範圍邊界進行檢查，確保最大可能值、最小可能值和其他有效特殊資料可以正常工作。

3．數量相對較少的非法資料測試用例，在邊界外檢查資料範圍邊界，確保無效的輸入能有正確的資料診斷提示。

修改程式。調整程式或者修復程式需要更多的資訊。顯然，這要求瞭解全部的細節，並且這些細節已經記錄在注釋良好的列表中。和一般用戶一樣，修改者迫切需要一份清晰明瞭的概述，不過這一次是關於系統的內部結構。那麼這份概述的組成部分是什麼呢？

1．流程圖或子系統的結構圖，對此以下有更詳細的論述。

2．對所用演算法的完整描述，或者是對文檔中類似描述的引用。

3．對所有檔規劃的解釋。

4．資料流程的概要描述——從磁片或者磁帶中，獲取資料或程式處理的序列——以及在每個處理過程完成的操作。

5．初始設計中，對已預見修改的討論；特性、功能回調的位置以及出口；原作者對可能會擴充的地方以及可能處理方案的一些意見。另外，對隱藏缺陷的觀察也同樣很有價值。

## 流程圖

流程圖是被吹捧得最過分的一種程式文檔。事實上，很多程式甚至不需要流程圖，很少有程式需要一頁紙以上的流程圖。

流程圖顯示了程式的流程判斷結構，它僅僅是程式結構的一個方面。當流程圖繪製在一張圖上時，它能非常優雅地顯示程式的判斷流向，但當它被分成幾張時，也就是說需要採用經過編號的出口和連接符來進行拼裝時，整體結構的概觀就嚴重地被破壞了。

因此，一頁紙的流程圖，成爲表達程式結構、階段或步驟的一種非常基本的圖示。同樣，它也非常容易繪製。圖15.1展示了一個子程式流程圖的圖樣。

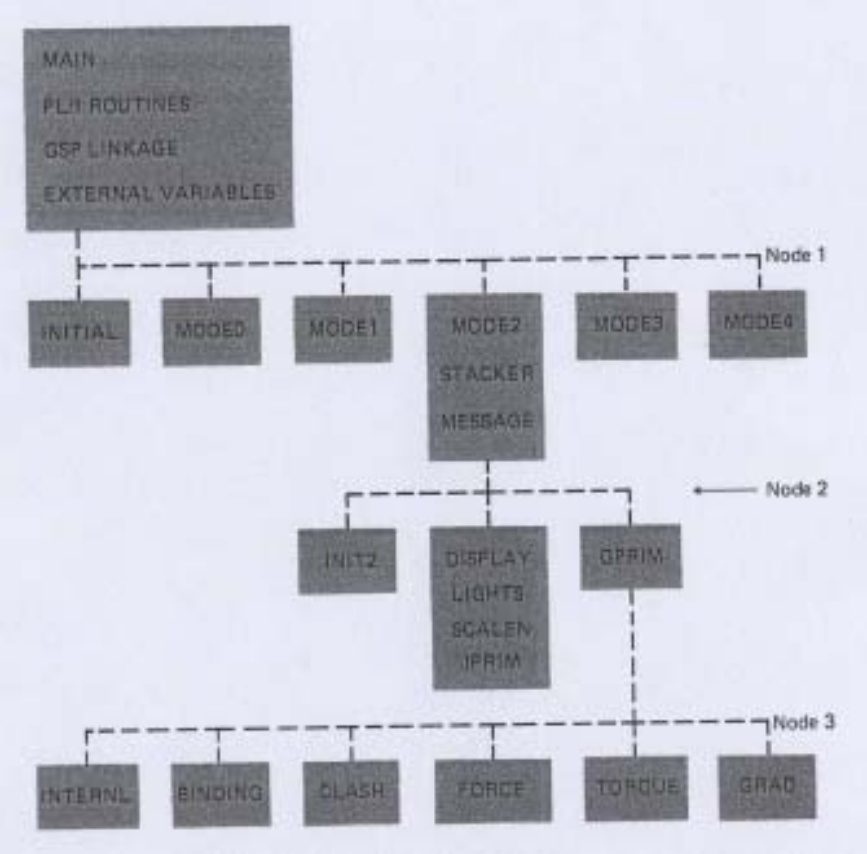



圖15.1：程式結構圖（Courtesy of W. V. Wright）

當然，上述圖紙既沒有，也不需要遵循精心制訂的ANSI流程圖標準。所有圖形元素如方框、連線、編號等，只需要能使這張詳細的流程圖可以理解就行了。

因此，逐一記錄的詳細流程圖過時而且令人生厭，它只適合啓蒙初學者的演算法思維。當Goldstine和Neumann[1]引入這種方法時，框圖和框圖中的內容作爲一種高級別語言，將難以理解的機器語言組合成一連串可理解的步驟。如同早期Iverson所認識到的[2]，在系統化的高階語言中，分組已經完成，每一個方框相應地包含了一條語句（圖15.2）。從而，方框本身變成了一件單調乏味的重複練習，可以去掉它們。這時，剩下的只有箭頭。而連接相鄰後續語句的箭頭也是冗餘的，可以擦掉它們。現在，留下的只有GO TO跳轉。如果大家遵守良好的規則，使用塊結構來消除GO TO語句，那麼所有的箭頭都消失了，儘管這些箭頭能在很大程度上幫助理解。大家完全可以丟掉流程圖，使用文字列表來表達這些內容。

現實中，流程圖被鼓吹的程度遠大於它們的實際作用。我從來沒有看到過一個有經驗的編程人員，在開始編寫程式之前，會例行公事地繪製詳盡的流程圖。在一些要求流程圖的組織中，流程圖總是事後才補上。一些公司則很自豪地使用工具軟體，從代碼中生成這個“不可缺少的設計工具”。我認爲這種普遍經驗並不是令人尷尬和惋惜的對良好實踐的偏離（似乎大家只能對它露出窘迫的微笑），相反，它是對技術的良好評判，向我們傳授了一些流程圖用途方面的知識。

耶穌門徒彼得談到新的異教皈依者和猶太戒律時說道，“爲什麼讓他們背負我們的祖先和我們自己都不能承擔的重負呢？”（《使徒行傳》 15:10 現代英文版本）。對於新的編程人員和陳舊的流程圖方法，我持有相同的觀點。

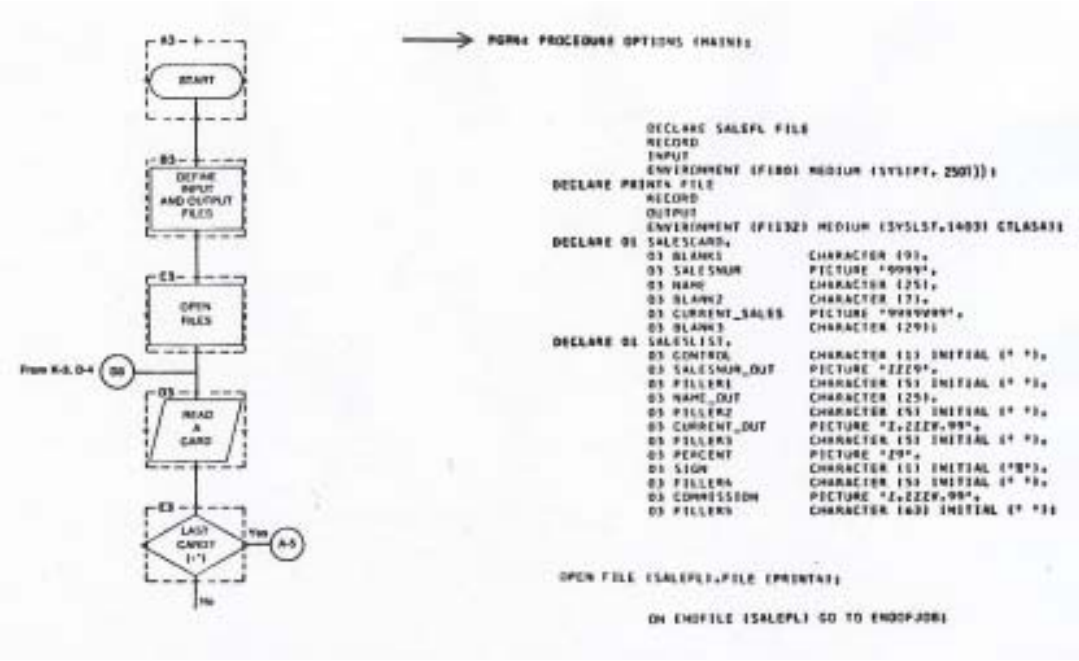
# 自文檔化（**self-documenting**）的程式

資料處理的基本原理告訴我們，試圖把資訊放在不同的檔中，並努力維持它們之間的同步，是一種非常費力不討好的事情。更合理的方法是：每個資料項目包含兩個檔都需要的所有資訊，採用指定的鍵值來區別，並把它們組合到一個檔中。

不過，我們在程式文檔編制的實踐中卻違反了我們自己的原則。典型的，我們試圖維護一份機器可讀的程式，以及一系列包含記敍性文字和流程圖的文檔。

結果和我們自己的認識相吻合。不同檔的資料保存帶來了不良的後果。程式文檔品質聲名狼藉，文檔的維護更是低劣：程式變動總是不能及時精確地反映在文檔中。

我認爲相應的解決方案是“合併檔”，即把文檔整合到源代碼。這對正確維護是直接有力的推動，保證編程用戶能方便、即時地得到文檔資料。這種程式被稱爲自文檔化（self-documenting）。



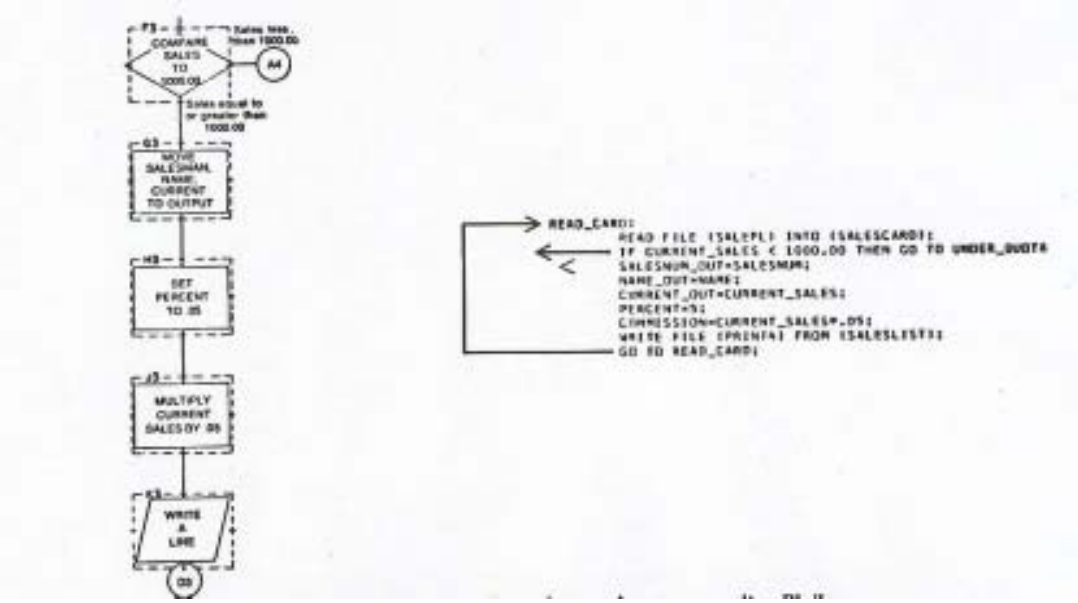



圖15.2：流程圖和對應程式的對比[節選自Thomas J. Cashman和Willian J. Keys（Harper & Row，1971）所著的“Data Processing and Computer Programming: A Modular Approach”中的圖15－41、15－44]

現在看來，在程式中包括流程圖顯然是一種笨拙（但不是不可以）的做法。考慮到流程圖方法的落後和高階語言的使用占統治地位，把程式和文檔放在一起顯然是很合理的。

把根源程式作爲文檔介質強制推行了一些約束。另一方面，對於文檔讀者而言，一行一行的根源程式本身就可以再次利用，使新技術的使用成爲可能。現在，已經到了爲程式文檔設計一套徹底的新方法的時候了。

文檔是我們以及前人都不曾成功背負的重擔。作爲基本目標，我們必須試圖把它的負擔降到最小。

方法。第一個想法是借助那些出於語言的要求而必須存在的語句，來附加盡可能多的“文檔”資訊。因此，標籤、聲明語句、符號名稱均可以作爲工具，用來向讀者表達盡可能多的意思。

第二個方法是盡可能地使用空格和一致的格式提高程式的可讀性，表現從屬和嵌套關係。

第三，以段落注釋的形式，向程式中插入必要的記敍性文字。大多數文檔一般都包括足夠多的逐行注釋，特別是那些滿足公司呆板的“良好文檔”規範的程式，通常就包含了很多注釋。即使是這些程式，在段落注釋方面也常常是不夠的，而段落注釋能提供總體把握和真正加深讀者對整件事情的理解。

因爲文檔是通過程式結構、命名和格式來實現的，所有這些必須在書寫代碼時完成。不過，這也只是應該完成的時間。另外，由於自文檔化的方法減少了很多附加工作，使這件工作遇到的障礙會更少。

一些技巧。圖15.3是一段自文檔化的PL/I程式[3]。圓圈中的數位不是程式的組成部分，而是用來幫助我們進行討論。

```
(1) //QLTA JOB ...

(2) QLTSRT7: PROCEDURE (V);

    /*************************************************************************/
(3) /*A SORT SUBROUTINE FOR 2500 4-BITE FIELDS, PASSED AS THE VECTOR V.  A     */
    /*SEPARATELY COMPILED, NOT-MAIN PROCEDURE, WHICH MUST USE AUTOMATIC CORE  */
    /*ALLOCATION.                                                             */
    /*                                                                        */
(4) /*THE SORT ALGORITHM FOLLOWS BROOKS AND IVERSON, AUTOMATIC DATA PROCESSING,*/
    /*PROGRAM 7.23, P. 350.  THAT ALGORITHM IS REVISED AS FOLLOWS:            */
(5) /*  STEPS 2-12 ARE SIMPLIFIED FOR M=2.                                    */
    /*  STEP 18 IS EXPANDED TO HANDLE EXPLICIT INDEXING OF THE OUTPUT VECTOR. */
    /*  THE WHOLE FIELD IS USED AS THE SORT KEY.                              */
    /*  MINUS INFINITY IS REPRESENTED BY ZEROS.                               */
    /*  PLUS INFINITY IS REPRESENTED BY ONES.                                 */
    /*  THE STATEMENT NUMBERS IN PROG. 7.23 ARE REFLECTED IN THE STATEMENT    */
    /*    LABELS OF THIS PROGRAM.                                             */
    /*  AN IF-THEN-ELSE CONSTRUCTION REQUIRES REPETITION OF A FEW LINES.      */
    /*                                                                        */
    /*TO CHANGE THE DIMENSION OF THE VECTOR TO BE SORTED, ALWAYS CHANGE THE   */
    /*INITIALIZATION OF T.  IF THE SIZE EXCEEDS 4096, CHANGE THE SIZE OF T,TOO.*/
    /*A MORE GENERAL VERSION WOULD PARAMETERIZE THE DIMENSION OF V.           */
    /*                                                                        */
    /*THE PASSED INPUT VECTOR IS REPLACED BY THE REORDERED OUTPUT VECTOR.     */
    /*************************************************************************/

(6) /* LEGEND  (ZERO-ORIGIN INDEXING)                                        */

   DECLARE
     (H,                    /*INDEX FOR INITIALIZING T                       */
      I,                    /*INDEX OF ITEM TO BE REPLACED                   */
      J,                    /*INITIAL INDEX OF BRANCHES FROM NODE I          */
      K) BINARY FIXED,      /*INDEX IN OUTPUT VECTOR                         */

     (MINF,                 /*MINUS INFINITY                                 */
      PINF) BIT (48),       /*PLUS INFINITY                                  */

     V (*)   BIT (*),       /*PASSED VECTOR TO BE SORTED AND RETURNED        */

     T (0:8190) BIT (48); /*WORKSPACE CONSISTING OF VECTOR TO BE SORTED, FILLED*/
                          /*OUT WITH INFINITIES, PRECEDED BY LOWER LEVELS    */
                          /*FILLED UP WITH MINUS INFINITIES                  */

   /* NOW INITIALIZATION TO FILL DUMMY LEVELS, TOP LEVEL, AND UNUSED PART OF TOP*/
   /* LEVEL AS REQUIRED.                                                      */

(7) INIT: MINF= (48) '0'B;
          PINF= (48) '1'B;

          DO L=    0 TO 4094;  T(L) = MINF;        END;
          DO L=    0 TO 2499;  T(L+4095) = V(L);   END;
          DO L=6595 TO 8190;   T(L) = PINF;        END;

(8) K0:  K = -1;                                                        (10)
    K1:  I = 0;                 /*          (11)                      <------| */
    K3:  J = 2*I+1;             /*SET J TO SCAN BRANCHES FROM NODE I.   <-----|| */
    K7:  IF T(J) <= T(J+1)      /*PICK SMALLER BRANCH                   -----> || */
           THEN      (12)       /*                                        ||| */
         (9)DO;                 /*                                        ||| */
    K11:   T(I) = T(J);         /*REPLACE                                 ||| */
    K13:   IF T(I) = PINF THEN GO TO K16; /*IF INFINITY, REPLACEMENT  +=   ||| */
                                /* IS FINISHED                            |||| */
    K12:   I = J;               /*SET INDEX FOR HIGHER LEVEL              |||| */
         END;                   /*                                        |||| */
         ELSE                   /*                                      <---+-|| */
           DO;                  /*                                        | || */
    K11A:    T(I) = T(J+1);  /*                                          | || */
    K13A:    IF T(I) = PINF THEN GO TO K16;       /*                 +=   | || */
    K12A:    I = J+1;         /*                                        | || */
           END;                 /*                                    <   | || */
    K16: IF 2*I < 8191 THEN GO TO K3;   /*GO BACK IF NOT ON TOP LEVEL  ----+-|| */
    K15: T(I) = PINF;           /*IF TOP LEVEL, FILL WITH INFINITY        |  | */
    K16: IF T(0) = PINF THEN RETURN;    /*TEST END OF SORT              <---|  | */
    K17: IF T(0) = MINF THEN GO TO K1;  /*FLUSH OUT INITIAL DUMMIES      -=    | */
    K18: K = K+1;                       /*STEP STORAGE INDEX                  | */
         V(K) = T(0);  GO TO K1; (12)   /*STORE OUTPUT ITEM             -------| */
   END QLTSRT7;
```

圖15.3：一段子文檔化程式

1．爲每次運算使用單獨的任務名稱。維護一份日誌，記錄程式運行的目的、時間和結果。如果名稱由一個助記符（這裏是QLT）和數字尾碼（4）組成，那麼尾碼可以作爲運算編號，把列表和日誌聯繫在一起。這種技術要求爲每次運算準備新的任務卡，不過這項工作可以採用“重複進行公共資訊的批次處理”來完成。

2．使用包含版本號和能幫助記憶的程式名稱。即，假設程式將會有很多版本。例子中使用的是1967年的最低一位元數位。

3．在過程（PROCEDURE）的注釋中，包含記敍性的描述文字。

4．盡可能爲基本演算法提供參考引用，通常它會指向更完備的處理方法。這樣，既節省了空間，同時還允許那些有經驗的讀者能非常自信地略過這一段內容。

5．顯示和演算法書籍中的傳統演算法的關係。

a）更改 b）定制細化 c）重新表達

6．聲明所有的變數。採用助記符，並使用注釋把DECLARE轉化成完整的說明。注意，聲明已經包含了名稱和結構性描述，需要增加的僅僅是對目的的解釋。通過這種方式，可以避免在不同的處理中重複名稱和結構性的描述。

7．用標籤標記出初始化的位置。

8．對程式語句進行分組和標記，以顯示與設計文檔中語句單元的一致性。

9．利用縮進表現結構和分組。

10．在程式列表中，手工添加邏輯箭頭。它們對調試和變更非常有幫助。它們還可以補充在頁面右邊的空白處（注釋區域），成爲機器可讀文字的一部分。

11．使用行注釋來解釋任何不很清楚的事情。如果採用了上述技術，那麼注釋的長度和數量都將小於傳統慣例。

12．把多條語句放置在一行，或者把一條語句拆放在若干行，以吻合邏輯思維，表示和其他演算法描述一致。

爲什麼不？這種方法的缺點在什麼地方？很多曾經遇到的困難，已經隨著技術的進步逐漸解決了。

最強烈的反對來自必須存儲的源代碼規模的增加。隨著編程技術越來越向線上源代碼存儲的方向發展，這成爲了一個主要的考慮因素。我發現自己編寫的APL程式注釋比PL/I程式要少，

這是因爲APL程式保存在磁片上，而PL/I則以卡片的形式存儲。

然而，與此同時文本編輯的訪問和修改，也在朝線上存儲的方向前進。就像前面討論過的，程式和文字的混合使用減少了需要存儲的字元總數。

對於文檔化程式需要更多輸入擊鍵的爭論，也有類似的答案。採用打字方式，每份草稿、每個字元需要至少一次擊鍵。而自文檔化程式的字元總數更少，每個字元需要的擊鍵次數也更少，並且電子草稿不需要重複列印。

那麼流程圖和結構圖的情況又如何呢？如果僅僅使用最高級別的結構圖，那麼另外使用一份文檔的方法可能更安全一些，因爲結構通常不會頻繁變化。它理所當然也可以作爲注釋合併到文檔中。這顯然是一種聰明的作法。

以上討論的用於文檔和軟體彙編的方法到底有多大的應用範圍呢？我認爲“自文檔化”方法的基本思想可以得到大規模的應用。““自文檔化”方法”對空間和格式要求更爲嚴格，這一點的應用可能會受限；而命名和結構化聲明顯然可以利用起來，在這方面，宏可以起到很大的幫助；另外，段落注釋的廣泛使用在任何語言中都是一個很棒的實踐。

自文檔化方法激發了高階語言的使用，特別是用於線上系統的高階語言——無論是對批次處理還是互動式，它都表現出最強的功效和應用的理由。如同我曾經提到的，上述語言和系統強有力地幫助了編程人員。因爲是機器爲人服務，而不是人爲機器服務。因此從各個方面而言，無論是從經濟上還是從以人爲本的角度來說，它們的應用都是非常合情合理的。

# 沒有銀彈－軟體工程中的根本和次要問題（*No Silver Bullet – Essence and Accident in Software Engineering*）

沒有任何技術或管理上的進展，能夠獨立地許諾十年內使生產率、可靠性或簡潔性獲得數量級上的進步。

*There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.*

# 摘要[1]

所有軟體活動包括根本任務——打造由抽象軟體實體構成的複雜概念結構，次要任務——使用編程語言表達這些抽象實體，在空間和時間限制內將它們映射成機器語言。軟體生產率在近年內取得的巨大進步來自對後天障礙的突破，例如硬體的限制、笨拙的編程語言、機器時間的缺乏等等。這些障礙使次要任務實施起來異常艱難，相對必要任務而言，軟體工程師在次要任務上花費了多少時間和精力？除非它占了所有工作的9/10，否則即使全部次要任務的時間縮減到零，也不會給生產率帶來數量級上的提高。

因此，現在是關注軟體任務中的必要活動的時候了，也就是那些和構造異常複雜的抽象概念結構有關的部分。我建議：

仔細地進行市場調研，避免開發已上市的產品。

在獲取和制訂軟體需求時，將快速原型開發作爲迭代計畫的一部分。

有機地更新軟體，隨著系統的運行、使用和測試，逐漸添加越來越多的功能。

不斷挑選和培養傑出的概念設計人員。

# 介紹

在所有恐怖民間傳說的妖怪中，最可怕的是人狼，因爲它們可以完全出乎意料地從熟悉的面孔變成可怕的怪物。爲了對付人狼，我們在尋找可以消滅它們的銀彈。

大家熟悉的軟體專案具有一些人狼的特性（至少在非技術經理看來），常常看似簡單明瞭的東西，卻有可能變成一個落後進度、超出預算、存在大量缺陷的怪物。因此，我們聽到了近乎絕望的尋求銀彈的呼喚，尋求一種可以使軟體成本像電腦硬體成本一樣降低的尙方寶劍。

但是，我們看看近十年來的情況，沒有銀彈的蹤跡。沒有任何技術或管理上的進展，能夠獨立地許諾在生產率、可靠性或簡潔性上取得數量級的提高。本章中，我們試圖通過分析軟體問題的本質和很多候選銀彈的特徵，來探索其原因。

不過，懷疑論者並不是悲觀主義者。儘管我們沒有看見令人驚異的突破，並認爲這種銀彈實際上是與軟體的內在特性相悖，不過還是出現了一些令人振奮的革新。這些方法的規範化、持續地開拓、發展和傳播確實是可以在將來使生產率產生數量級上的提高。雖然沒有通天大道，但是路就在腳下。

解決管理災難的第一步是將大塊的“巨無霸理論”替換成“微生物理論”，它的每一步——希望的誕生，本身就是對一蹴而就型解決方案的衝擊。它告訴工作者進步是逐步取得的，伴隨著

辛勤的勞動，對規範化過程應進行持續不懈的努力。由此，誕生了現在的軟體工程。

## 是否一定那麼困難呢？——根本困難

不僅僅是在目力所及的範圍內，沒有發現銀彈，而且軟體的特性本身也導致了不大可能有任何的發明創新——能夠像電腦硬體工業中的微電子器件、電晶體、大規模集成一樣——提高軟體的生產率、可靠性和簡潔程度。我們甚至不能期望每兩年有一倍的增長。

首先，我們必須看到這樣的畸形並不是由於軟體發展得太慢，而是因爲電腦硬體發展得太快。從人類文明開始，沒有任何其他產業技術的性價比，能在30年之內取得6個數量級的提高，也沒有任何一個產業可以在性能提高或者降低成本方面取得如此的進步。這些進步來自電腦製造產業的轉變，從裝配工業轉變成流水線工業。

其次，讓我們通過觀察預期的軟體技術產業發展速度，來瞭解中間的困難。效仿亞里斯多德，我將它們分成根本的——軟體特性中固有的困難，次要的——出現在目前生產上的，但並非那些與生俱來的困難。

我們在下一章中討論次要問題。首先，來關注內在、必要的問題。

一個相互牽制關聯的概念結構，是軟體實體必不可少的部分，它包括：資料集合、資料條目之間的關係、演算法、功能調用等等。這些要素本身是抽象的，體現在相同的概念構架中，可以存在不同的表現形式。儘管如此，它仍然是內容豐富和高度精確的。

我認爲軟體發展中困難的部分是規格化、設計和測試這些概念上的結構，而不是對概念進行表達和對實現逼真程度進行驗證。當然，我們還是會犯一些語法錯誤，但是和絕大多數系統中的概念錯誤相比，它們是微不足道的。

如果這是事實，那麼軟體發展總是非常困難的。天生就沒有銀彈。

讓我們來考慮現代軟體系統中這些無法規避的內在特性：複雜度、一致性、可變性和不可見性。

複雜度。規模上，軟體實體可能比任何由人類創造的其他實體要複雜，因爲沒有任何兩個軟體部分是相同的（至少是在語句的級別）。如果有相同的情況，我們會把它們合併成供調用的子函數。在這個方面，軟體系統與電腦、建築或者汽車大不相同，後者往往存在著大量重複的部分。

數位電腦本身就比人類建造的大多數東西複雜。電腦擁有大量的狀態，這使得構思、描述和測試都非常困難。軟體系統的狀態又比電腦系統狀態多若干個數量級。

同樣，軟體實體的擴展也不僅僅是相同元素重複添加，而必須是不同元素實體的添加。大多數情況下，這些元素以非線性遞增的方式交互，因此整個軟體的複雜度以更大的非線性級數增長。

軟體的複雜度是必要屬性，不是次要因素。因此，抽掉複雜度的軟體實體描述常常也去掉了一些本質屬性。數學和物理學在過去三個世紀取得了巨大的進步，數學家和物理學家們建立模型以簡化複雜的現象，從模型中抽取出各種特性，並通過試驗來驗證這些特性。這些方法之所以可行——是因爲模型中忽略的複雜度不是被研究現象的必要屬性。當複雜度是本質特性時，這些方法就行不通了。

上述軟體特有的複雜度問題造成了很多經典的軟體產品開發問題。由於複雜度，團隊成員之間的溝通非常困難，導致了產品瑕疵、成本超支和進度延遲；由於複雜度，列舉和理解所有可能的狀態十分困難，影響了產品的可靠性；由於函數的複雜度，函數調用變得困難，導致程式難以使用；由於結構性複雜度，程式難以在不產生副作用的情況下用新函數擴充；由於結構性複雜度，造成很多安全機制狀態上的不可見性。

複雜度不僅僅導致技術上的困難，還引發了很多管理上的問題。它使全面理解問題變得困難，從而妨礙了概念上的完整性；它使所有離散出口難以尋找和控制；它引起了大量學習和理解上的負擔，使開發慢慢演變成了一場災難。

一致性。並不是只有軟體工程師才面對複雜問題。物理學家甚至在非常“基礎”的級別上，面對異常複雜的事物。不過，物理學家堅信必定存在著某種通用原理，或者在誇克中，或者在統一場論中。愛因斯坦曾不斷地重申自然界一定存在著簡化的解釋，因爲上帝不是專橫武斷或反復無常的。

軟體工程師卻無法從類似的信念中獲得安慰，他必須控制的很多複雜度是隨心所欲、毫無規則可言的，來自若干必須遵循的人爲慣例和系統。它們隨介面的不同而改變，隨時間的推移而變化，而且，這些變化不是必需的，僅僅由於它們是不同的人——而非上帝——設計的結果。

某些情況下，因爲是開發最新的軟體，所以它必須遵循各種介面。另一些情況下，軟體的開發目標就是相容性。在上述的所有情況中，很多複雜性來自保持與其他介面的一致，對軟體的任何再設計，都無法簡化這些複雜特性。

可變性。軟體實體經常會遭受到持續的變更壓力。當然，建築、汽車、電腦也是如此。不過，工業製造的產品在出廠之後不會經常地發生修改，它們會被後續模型所取代，或者必要更改會被整合到具有相同基本設計的後續產品系列。汽車的更改十分罕見，電腦的現場調整時有發生。然而，它們和軟體的現場修改比起來，都要少很多。

其中部分的原因是因爲系統中的套裝軟體含了很多功能，而功能是最容易感受變更壓力的部分。另外的原因是因爲軟體可以很容易地進行修改——它是純粹思維活動的產物，可以無限擴展。日常生活中，建築有可能發生變化，但眾所周知，建築修改的成本很高，從而打消了那些想提出修改的人的念頭。

所有成功的軟體都會發生變更。現實工作中，經常發生兩種情況。當人們發現軟體很有用時，會在原有應用範圍的邊界，或者在超越邊界的情況下使用軟體。功能擴展的壓力主要來自那些喜歡基本功能，又對軟體提出了很多新用法的用戶們。

其次，軟體一定是在某種電腦硬體平臺上開發，成功軟體的生命期通常比當初的電腦硬體平臺要長。即使不是更換電腦，則有可能是換新型號的磁片、顯示器或者印表機。軟體必須與各種新生事物保持一致。

簡言之，軟體產品紮根于文化的母體中，如各種應用、用戶、自然及社會規律、電腦硬體等等。後者持續不斷地變化著，這些變化無情地強迫著軟體隨之變化。

不可見性。軟體是不可見的和無法視覺化的。例如，幾何抽象是強大的工具。建築平面圖能幫助建築師和客戶一起評估空間佈局、進出的運輸流量和各個角度的視覺效果。這樣，矛盾變得突出，忽略的地方變得明顯。同樣，機械製圖、化學分子模型儘管是抽象模型，但都起了相同的作用。總之，都可以通過幾何抽象來捕獲物理存在的幾何特性。

軟體的客觀存在不具有空間的形體特徵。因此，沒有已有的表達方式，就像陸地海洋有地圖、矽片有膜片圖、電腦有電路圖一樣。當我們試圖用圖形來描述軟體結構時，我們發現它不僅僅包含一個，而是很多相互關聯、重疊在一起的圖形。這些圖形可能描繪控制流程、資料流程、依賴關係、時間序列、名字空間的相互關係等等。它們通常不是有較少層次的扁平結構。實際上，在上述結構上建立概念控制的一種方法是強制將關聯分割，直到可以層次化一個或多個圖形[2]。

除去軟體結構上的限制和簡化方面的進展，軟體仍然保持著無法視覺化的固有特性，從而剝奪了一些具有強大功能的概念工具的構造思路。這種缺憾不僅限制了個人的設計過程，也嚴重地阻礙了相互之間的交流。

## 以往解決次要困難的一些突破

如果回顧一下軟體領域中取得的最富有成效的三次進步，我們會發現每一次都是解決了軟體構建上的巨大困難，但是這些困難不是本質屬性，也不是主要困難。同樣，我們可以對每一次進步進行外推，來瞭解它們的固有限制。

高階語言。勿庸置疑，軟體生產率、可靠性和簡潔性上最有力的突破是使用高階語言編程。大多數觀察者相信開發生產率至少提高了五倍，同時可靠性、簡潔性和理解程度也大爲提高。

那麼，高階語言取得了哪些進展呢？首先，它減輕了一些次要的軟體複雜度。抽象套裝程式含了很多概念上的要素：操作、資料類型、流程和相互通訊，而具體的機器語言程式則關心位元、寄存器、條件、分支、通道、磁片等等。高階語言所達到的抽象程度包含了（抽象）程式所需要的要素，避免了更低級的元素，它消除了並不是程式所固有的整個級別的複雜度。

高階語言最可能實現的是提供所有編程人員在抽象程式中能想到的要素。可以肯定的是，我們思考資料結構、資料類型和操作的速度穩固提高，不過是以非常緩慢的速度。另外，程式開發方法越來越接近用戶的複雜度。

然而，對於較少使用那些複雜深奧語言要素的用戶，高階語言在某種程度上增加而不是減少了腦力勞動上的負擔。

分時。大多數觀察者相信分時提高了程式師的生產率和產品的品質，儘管它帶來的進步不如高階語言。

分時解決了完全不同的困難。分時保證了及時性，從而使我們能維持對複雜程度的一個總

體把握。批次處理編程的較長周轉時間意味著不可避免會遺忘一些細枝末節，如果我們停下編程，調用編譯程序或者執行程式，思維上的中斷使我們不得不重新進行思考，它在時間上的代價非常高昂。最嚴重的結果可能是失去對複雜系統的掌握。

較長的周轉時間和機器語言的複雜度一樣，是軟體發展過程的次要困難，而不是本質困難。分時所起作用也非常有限。主要效果是縮短了系統的回應時間。隨著它接近於零，到達人類可以辨識的基本能力——大概100毫秒時，所獲得的好處就接近於無了。

統一編程環境。第一個集成開發環境——Unix和Interlisp現在已經得到了廣泛應用，並且使生產率提高了5倍。爲什麼？

它們主要通過提供集成庫、統一檔格式、管道和篩檢程式，解決了共同使用程式的次要困難。這樣，概念性結構理論上的相互調用、提供輸入和互相使用，在現實中可以非常容易地實現。

因爲每個新工具可以通過標準格式在任何一個程式中應用，這種突破接著又激發整個工具庫的開發。

由於這些成功，環境開發是當今軟體工程研究的主要題目。我們將在下章中討論期望達到的目標和限制。

## 銀彈的希望

現在，讓我們來討論一下當今可能作爲潛在銀彈的最先進的技術進步。它們各自針對什麼樣的問題？它們是屬於必要問題，或者依然是解決我們剩下的次要困難？它們是提供了創新，還是僅僅是增量改進？

Ada和其他高級編程語言。近來，最被吹捧的開發進展之一是編程語言Ada，一種80年代的高階語言。Ada實際上不僅僅反映了語言概念上的突破性進展，而且蘊涵了鼓勵現代設計和模組化概念運用的重要特性。由於Ada採用的是抽象資料類型、層次結構的模組化理念，所有Ada理念可能比語言本身更加先進。Ada使用設計來承載需求，作爲這一過程的自然產物，它可能過於豐富了。不過，這並不是致命的，因爲它的辭彙子集可以解決學習問題，硬體的進展能提供更高的MIPS（每秒百萬指令集），減少編譯的成本。軟體系統結構化的先進理念實際上非常好地利用了MIPS上的進展。60年代，曾在記憶體和速度成本上廣受譴責的作業系統，如今已被證明是一種能使用某些MIPS和廉價記憶體的非常優秀的系統。

然而，Ada仍然不是消滅軟體生產率怪獸的銀彈。畢竟，它只是另一種高階語言，這類語言出現最大的回報來自出現時的衝擊，它通過使用更加抽象的語句來開發，降低了機器的次要複雜度。一旦這些難題被解決，就只剩下非常少的問題，解決剩餘部分的獲益必然也要少一些。

我預言在以後的十年中，當Ada的效率被大家評估認可時，它會帶來相當大的變化，這並不是因爲任何特別的語言特性，不是由於這些語言特性被合併在一起，也不是因爲Ada開發環境會不斷發展進步。Ada的最大貢獻在於編程人員培訓方式的轉變，即對開發人員需要進行現代軟體設計技術培訓。

面向物件編程。軟體專業的一些學生堅持面向物件編程是當今若干新潮技術中最富有希望的[3]。我也是其中之一。達特茅斯的Mark Sherman提出，必須仔細地區別兩個不同的概念：抽象資料類型和層次化類型，後者也被稱爲類（class）。抽象資料類型的概念是指物件類型應該通過一個名稱、一系列合適的值和操作來定義，而不是理應被隱藏的存儲結構。抽象資料類型的例子是Ada包（以及私有類型）和Modula的模組。

層次化類型，如Simula-67的類，是允許定義可以被後續子類型精化的通用介面。這兩個概念是互不相干的——可以只有層次化，沒有資料隱藏；也可能是只有資料隱藏，而沒有層次化。兩種概念都體現了軟體發展領域的進步。

它們的出現都消除了開發過程中的非本質困難，允許設計人員表達自己設計的內在特性，而不需要表達大量句法上的內容，這些內容並沒有添加什麼新的資訊。對於抽象資料類型和層次化類型，它們都是解決了高級別的次要困難和允許採用較高層次的表現形式來表達設計。

不過，這些提高僅僅能消除所有設計表達上的次要困難。軟體的內在問題是設計的複雜度，上述方法並沒有對它有任何的促進。除非我們現在的編程語言中，不必要的低層次類型說明佔據了軟體產品設計90%，面向物件編程才能帶來數量級上的提高。對面向物件編程這顆“銀彈”，我深表懷疑。

人工智慧。很多人期望人工智慧上的進展可以給軟體生產率和品質帶來數量級上的增長[4]，但我不這樣認爲。追究其原因，我們必須剖析“人工智慧”意味著什麼，以及它如何應用。

Parnas澄清了術語上的混亂：

現在有兩種差異非常大的AI定義被廣泛使用。AI-1：使用電腦來解決以前只能通過人類智慧解決的問題。AI-2：使用啓發式和基於規則的特定編程技術。在這種方法中，對人類專家進行研究，判斷他們解決方法的啓發性思維或者經驗法則□□。程式被設計成以人類解決問題的方式來運作。

第一種定義的意義容易發生變化□□今天可能適合AI-1定義的程式，一旦我們瞭解了它的運行方式，理解了問題，就不再認爲它是人工智慧□□不幸的是，我無法識別這個領域的特定知識體系□□絕大多數工作是針對問題域的，我們需要一些抽象或者創造性來解決上述問題[5]。

我完全同意這種批評意見。語音識別技術與圖像識別技術的共同點非常少，它們與專家系統中應用的技術不同。例如，我覺得很難去發現圖像識別技術能給編程開發實踐帶來什麼樣的差異。同樣，語音識別也差不多——軟體發展上的困難是決定說什麼，而不是如何說。表達的簡化僅僅能提供少量的促進作用。

至於AI-2專家系統技術，應該用專門的章節來討論。

專家系統。人工智慧領域最先進的、被最大範圍使用的部分，是開發專家系統的技術。很多軟體科學家正非常努力地工作著，想把這種技術應用在軟體的開發環境中[5]。那麼它的概念是什麼，前景如何？

專家系統是包含歸納推論引擎和規則基礎的程式，它接收輸入資料和假設條件，通過從基礎規則推導邏輯結果，提出結論和建議，向用戶展示前因後果，並解釋最終的結果。推論引擎除了處理推理邏輯以外，通常還包括複雜邏輯或者概率資料和規則。

對於解決相同的問題，這種系統明顯比傳統的程式演算法要先進很多。

推導引擎技術的開發獨立于應用程式，因此可以用於多個用戶。在該引擎上花費較大的工作是很合理的。實際上，這種引擎技術非常先進。

基於應用的、可變更的部分，在基礎規則中以一種統一的風格編碼，並且爲規則基礎的開發、更改、測試和文檔化提供了若干工具。這實際上對一些應用程式本身的複雜度進行了系統化。

Edward Feigenbaum指出這種系統的能力不是來自某種前所未有的推導機制，而是來自非常豐富的知識積累基礎，所以更加精確地反映了現實世界。我認爲這種技術提供的最重要進步是具體應用的複雜性與程式本身相分離。

如何把它應用在軟體發展工作中？可以通過很多途徑：建議介面規則、制訂測試策略、記錄各種bug產生的頻率、提供優化建議等等。

例如，考慮一個虛構的測試顧問系統。在最根本的級別，診斷專家系統和飛行員的檢查列表很相似，對可能難以尋找的原因提供基本的建議。建立基礎規則時，可以依據更多的複雜問題徵兆報告，從而使這些建議更加精確。可以想像，該調試輔助程式起初提供的是一般化建議，隨著基礎規則包括越來越多的系統結構資訊時，它產生的推測和推薦的測試也越來越準確。該類型的專家系統可能與傳統系統徹底分離，系統中的規則基礎可能與相應的軟體產品具有相同的層次模組化結構，因此當產品模組化修改時，診斷規則也能相應地進行模組化修改。

產生診斷規則也是在爲模組和系統編制測試用例集時必須完成的任務。如果它以一種適當通用的方式來完成，對規則採用一致的結構，擁有一個良好可用的推測引擎，那麼事實上它就可以減少測試用例設計的總體工作量，以及幫助整個軟體生命週期的維護、修改和測試。同樣，我們可以推測其他的顧問專家系統　　可能是它們中的某一些，或者是較簡單的系統　　能夠用在軟體發展的其他部分。

在較早實現的用於軟體發展的專家顧問系統中，存在著很多困難。在我們假設的例子中，一個關鍵的問題是尋找一種方法，能從軟體結構的技術說明中，自動或者半自動地產生診斷規則。另外，更加重要也是更加困難的任務是：尋覓能夠清晰表達、深刻理解爲什麼的分析專家；開發有效的技術　　抽取專家們所瞭解的知識，把它們精煉成基礎規則。這項工作的工作量是知識獲取工作量的兩倍。構建專家系統的必要前提條件是擁有專家。

專家系統最強有力的貢獻是給缺乏經驗的開發人員提供服務，用最優秀開發者的經驗和知識積累爲他們提供了指導。這是非常大的貢獻。最優秀和一般的軟體工程實踐之間的差距是非常大的，可能比其他工程領域中的差距都要大，一種傳播優秀實踐的工具特別重要。

自動　編程。近四十年中，人們一直在預言和編寫有關　自動編程　的文字，從問題的一段陳述說明自動產生解決問題的程式。現在，仍有一些人期望這樣的技術能夠成爲下一個突破點[7]。

Parnas暗示這是一個用於魔咒的術語，聲稱它本身是語義不完整的。

一句話，自動編程總是成爲一種熱情，使用現在並不可用的更高階語言編程的熱情[8]。

他指出，大多數情況下所給出的技術說明本質上是問題的解決方法，而不是問題自身。

可以找到一些例外情況。例如，資料發生器的開發技術就非常實用，並經常地用於排序程式中。系統評估若干參數，從問題解決方案庫中進行選擇，生成合適的程式。

這樣的應用具有非常良好的特性：

問題通過相對較少的參數迅速地描述出特徵。

存在很多已知的解決方案，提供了可供選擇的庫。

在給定問題參數的前提下，大量廣泛的分析爲選擇具體的解決技術提供了清晰的規則。

具有上述簡潔屬性的系統是一個例外，很難看到該方法能普及到更廣泛的尋常軟體系統，甚至難以想像這種突破如何能夠進行推廣。

圖形化編程。在軟體工程的博士論文中，一個很受歡迎的主題是圖形化和視覺化編程，電腦圖形在軟體設計上的應用[9]。這種方法的推測部分來自VLSI晶片設計的類比，電腦圖形化在設計中扮演了高生產力的角色。部分源於　人們將流程圖作爲一種理想的設計介質，並爲繪製它們提供了很多功能強大的實用程式　這證實了圖形化的可行性。

不過，上述方法中至今還沒有出現任何令人信服和激動的進步。我確信將來也不會出現。

首先，如同我先前所提出的，流程圖是一種非常差勁軟體結構表達方法[10]。實際上，它最好被視爲是馮・諾依曼、戈爾德斯廷和勃克斯試圖爲他們所設計的電腦提供的一種當時迫切需要的高級控制語言。如今的流程圖已經變得複雜，一張圖有若干頁，有很多連接結點。這種表現形式實在令人同情。流程圖已經成爲完全不必要的設計工具　程式師在開發之後，而不是之前繪製描述程式的流程圖。

其次，現在的螢幕非常小，圖元級別，無法同時表現軟體圖形的所有正式、詳細的範圍和細節。現在所謂　類似桌面　的工作站實際上像是　飛機坐艙座椅　。飛機上，任何坐在兩個

肥胖乘客之間，反復挪動一大兜檔的人會意識到這中間的差別　　每次只能看到很少的內容。真正的桌面提供了很多檔的總覽，讓大家可以隨意地使用它們。此外，當人們的創造力一陣陣地湧現時，開發人員大多數都會捨棄工作臺，使用空間更爲廣闊的地板。要使我們面對的工作空間滿足軟體發展工作的需要，硬體技術必須進一步發展。

更加基本的是，如同我們上面所爭論的，軟體非常難以視覺化。即使用圖形表達出了流程圖、變數範圍嵌套情況、變數交叉引用、資料流程、層次化資料結構等等，也只是表達了某個方面，就像盲人摸象一樣。如果我們把很多相關的視圖疊加在所產生的圖形上，那麼很難再抽取出全局的總體視圖。對VLSI晶片設計方法的類推是一種誤導　　晶片設計是對兩維物件的層次設計，它的幾何特性反映了它的本質特性，而軟體系統不是這樣。

程式驗證。現代編程的許多工作是測試和修復bug。是否有可能出現銀彈，能夠在系統設計級別、源代碼級別消除bug呢？是否可以在大量工作被投入到實現和測試之前，通過採用證實設計正確性的　深奧　策略，徹底提高軟體的生產率和產品的可靠性？

我並不認爲在這裏能找到魔法。程式驗證的確是很先進的概念，它對安全作業系統內核等這類應用是非常重要的。不過，這項技術並不能保證節約勞動力。驗證要求如此多的工作量，以致於只有少量的程式能夠得到驗證。

程式驗證不意味著零缺陷的程式。這裏並沒有什麼魔術，數學驗證仍然可能是有錯誤的。因此，儘管驗證可能減少程式測試的工作量，但卻不能省略程式測試。

更嚴肅地說，完美的程式驗證只能建立滿足技術說明的程式。這時，軟體工作中最困難的部分已經接近完成，形成了完整和一致的說明。開發程式的一些必要工作實際上已經變成對技術規格說明進行測試。

環境和工具。向更好的編程開發環境開發中投入，我們可以期待得到多少回報呢？人們的本能反應是首先著手解決高回報的問題：層次化檔系統，統一檔格式以獲得一致的編程介面和通用工具等。特定語言的智慧化編輯器在現實中還沒有得到廣泛應用，不過它們最有希望實現的是消除語法錯誤和簡單的語義錯誤。

開發環境上，現在已經實現的最大成果可能是集成資料庫的使用，用來跟蹤大量的開發細節，供每個程式師精確地查閱資訊，以及在整個團隊協作開發中保持最新的狀態。

顯然，這樣的工作是非常有價值的，它能帶來軟體生產率和可靠性上的一些提高。但是，由於它自身的特性，目前它的回報很有限。

工作站。隨著工作站的處理能力和記憶體容量的穩固和快速提高，我們能期望在軟體領域取得多大的收穫呢？現在的運算速度已經可以完全滿足程式編制和文檔書寫的需要。編譯還需要一些提高，不過一旦機器運算速度提高十倍，那麼程式開發人員的思考活動將成爲日常工作的主要活動。實際上，這已經是現在的情況。

我們當然歡迎更加強大的工作站，但是不能期望有魔術般的提高。

# 針對概念上根本問題的頗具前途的方法

雖然現在軟體上沒有技術上的突破能夠預示我們可以取得像在硬體領域上一樣的進展，但在現實的軟體領域中，既有大量優秀的工作，也有不引人注意的平穩進步。

所有針對軟體發展過程中次要困難的技術工作基本上能表達成以下的生產率公式：

$$\text{任務時間} = \sum (\text{頻率})_i \times (\text{時間})_i$$

如果和我所認爲的一樣，工作的創造性部分佔據了大部分時間，那麼那些僅僅是表達概念的活動並不能在很大程度上影響生產率。

因此，我們必須考慮那些解決軟體上必要困難的活動——即，準確地表達複雜概念結構。幸運的是，其中的一些非常有希望。

購買和自行開發。構建軟體最可能的徹底解決方案是不開發任何軟體。

情況每一天都有些好轉，越來越多的軟體提供商，爲各種眼花繚亂的應用程式提供了品質更好、數量更多的軟體產品。當我們的軟體工程師正忙於生產方法學時，個人電腦的驚天動地的變化爲軟體創造了廣闊的市場。每個報攤上都有大量的月刊，根據機器的類型，刊登著從幾美元到幾百美元的各種產品的廣告和評論。更多專業廠商爲工作站和UNIX市場提供了很多非常有競爭力的產品，甚至很多工具軟體和開發環境軟體都可以隨時購買使用。對於獨立的軟體模組市場，我已在其他的地方提出一些建議。

以上提到的任何軟體，購買都要比重新開發要低廉一些。即使支付100,000美元，購買的軟體也僅僅是一個人年的成本。而且軟體是立即可用的！至少對於現有的產品、對於那些專注于該領域開發者的成果而言，它們是可以立刻投入使用的。並且，它們往往配備了書寫良好的文檔，在某種程度上比自行開發的軟體維護得更加完備。

我相信，這個大眾市場將是軟體工程領域意義最深遠的開發方向。軟體成本一直是開發的成本，而不是複製的成本。所以，即使只在少數使用者之間實現共用，也能在很大程度上減少成本。另一種看法是使用軟體系統的n個拷貝，將會使開發人員的生產率有效地提高n倍。這是一個領域和行業範圍的提高。

當然，關鍵的問題還是可用性。是否可以在自己的開發工作中使用商用的套裝軟體？這裏，有一個令人吃驚的問題。在1950～1960年期間，一個接一個的研究顯示，用戶不會在工資系統、物流控制、帳務處理等系統中使用商用套裝軟體。需求往往過於專業，不同情況之間的差別太大。在80年代，我們發現這些套裝軟體的需求大爲增加，並得到了大規模的使用。什麼導致了這樣的變化？

並不是套裝軟體發生了變化。它們可能比以前更加通用和更加客戶化一些，但並不太多。同樣，也不是應用發生了變化。即使有，今天的商業和學術上的需要也比20年以前更加不同和複雜。

重大的變化在於電腦硬體/軟體成本比率。在1960年，2百萬美元機器的購買者覺得他可以爲定制的薪資系統支付250,000美元。現在，對50,000美元的辦公室機器購買者而言，很難想像能爲定制薪資系統再支付費用。因此，他們把上述系統的模組進行調整，添加到可用的套裝軟體中。電腦現在如此的普遍，上述的改編和調整是發展的必然結果。

我的上述觀點也存在一些戲劇性的例外——套裝軟體的通用化方面並沒有發生什麼變化，除了試算表和簡單的資料庫系統。這些強大的工具，出現得如此之晚和如此醒目，導致無數應用中的一些並不十分規範。大量的文章、甚至書籍講述了如何使用試算表應付很多意想不到的問題。原先作爲客戶程式，使用Cobol或者報表生成程式編寫的大量應用，如今已經被這些工具所取代。

現在很多用戶天天操作電腦，使用著各種各樣的應用程式，但並不編寫代碼。事實上，他們中間很多人無法爲自己的電腦編寫任何程式，不過他們非常熟練地使用電腦來解決新問題。

我認爲，對於現在的很多組織機構來說，最有效的軟體生產率策略是在生產一線配備很多個人電腦，安裝好通用的書寫、作圖、檔管理和試算表程式，以及配備能熟練使用它們的人員，並且把這些人員散佈到各個崗位。類似的策略——通用的數學和統計套裝軟體，以及一些簡單的編程能力，同樣地適用於很多實驗室的科學工作者。

需求精煉和快速原型。開發軟體系統的過程中，最困難的部分是確切地決定搭建什麼樣的系統。概念性工作中，沒有其他任何一個部分比確定詳細的技術需求更加困難，詳細的需求包括了所有的人機界面、與機器和其他軟體系統的介面。需求工作對系統的影響比其他任何一個部分的失誤都大，當然糾正需求的困難也比其他任何一個部分要大。

因此，軟體發展人員爲客戶所承擔的最重要的職能是不斷重複地抽取和細化產品的需求。事實上，客戶不知道他們自己需要什麼。他們通常不知道哪些問題是必須回答的。並且，連必須確定的問題細節常常根本不予考慮，甚至只是簡單地回答——“開發一個類似於我們已有的手工處理過程的新軟體系統”——實際上都過於簡單。客戶決不會僅僅要求這些。複雜的軟體系統往往是活動的、變化的系統。活動的動態部分是很難想像的。所以，在計畫任何軟體活動時，要讓客戶和設計人員之間進行多次廣泛的交流溝通，並將其作爲系統定義的一部分。這是非常必要的。

這裏，我將向前多走一步，下一個定論。在嘗試和開發一些客戶定制的系統之前，即使他們和軟體工程師一起工作，想要完整、精確、正確地抽取現代軟體產品的需求——這，實際上也是不可能的。

因此，現在的技術中最有希望的，並且解決了軟體的根本而非次要問題的技術，是開發作爲迭代需求過程的一部分——快速原型化系統的方法和工具。

軟體系統的快速原型對重要的系統介面進行類比，並演示待開發系統的主要功能。原型不必受到相同硬體速度、規模或者成本約束的限制。原型通常展示了應用程式的功能主線，但不處理任何如無效輸入、退出清除等異常情況。原型的目的是明確實際的概念結構，從而客戶可以測試一致性和可用性。

現在的軟體發展流程基於如下的假設——事先明確地闡述系統，爲系統開發競標，實際進行開發，最後安裝。我認爲這種假設根本上就是不正確的，很多軟體問題就來自這種謬誤。因

此，如果不進行徹底地調整，就無法消除那些軟體問題。其中，一種改進是對產品和原型不斷往復地開發和規格化。

增量開發——增長，而非搭建系統。我現在還記得在1958年，當聽到一個朋友提及搭建（building），而不是編寫（writing）系統時，我所感受到的震動。一瞬間，我的整個軟體發展流程的視野開闊了。這種暗喻是非常有力和精確的。現在，我們已經理解軟體發展是如何類似於其他的建造過程，並開始隨意地使用其他的暗喻，如規格說明、構件裝備、腳手架（測試平臺）（specifications, assembly of components, and scaffolding）。

暗喻"搭建系統"的使用已經有些超出了它的有效期限，是重新換一種表達方式的時候了。如果現在的開發情況和我考慮的一樣，那些概念性的結構非常複雜，以致於難以事先精確地說明和零缺陷地開發，那麼我們必須採用徹底不同的方法。

讓我們轉向自然界，研究一下生物的複雜性，而不是人們的僵硬工作。我們會發現它們的複雜程度令我們敬畏。光是大腦本身，就比任何對它的描述都要複雜，比任何的模擬仿真都要強大，它的多樣性、自我保護和自我更新能力異常豐富和有力。其中的秘密就是逐步發育成長，而不是一次性搭建。

所以，我們的軟體系統也必須如此。很多年前，Harlan Mill建議所有軟體系統都應該以增量的方式開發[11]。即，首先系統應該能夠運行，即使未完成任何有用功能，只能正確調用一系列僞子系統。接著，系統一點一點被充實，子系統輪流被開發，或者是在更低的層次調用程式、模組、子系統的占位元符（僞程式）等。

從我在軟體工程試驗班上開始推動這種方法起，其效果不可思議。在過去幾十年中，沒有任何方法和技術能如此徹底地改變我自己的實踐。這種方法迫切地要求自頂向下設計，因爲它本身是一種自頂向下增長的軟體。增量化開發使逆向跟蹤很方便，並非常容易進行原型開發。每一項新增功能，以及針對更加複雜資料或情況的新模組，從已經規劃的系統中有機地增長。

這種開發模式對士氣的推動是令人震驚的。當一個可運行系統——即使是非常簡單的系統出現時，開發人員的熱情就迸發了出來。當一個新圖形軟體系統的第一副圖案出現在螢幕上時，即使是一個簡單的長方形，工作的動力也會成倍地增長。在開發過程中的每個階段，總有可運行的系統。我發現開發團隊可以在四個月內，培育（grow）出比搭建（building）複雜得多的系統。

大型項目同樣可以得到與我所參與的小型項目相同的好處[12]。

卓越的設計人員。關鍵的問題是如何提高軟體行業的核心，一如既往的是——人員。

我們可以通過遵循優秀而不是拙劣的實踐，來得到良好的設計。優秀的設計是可以傳授的。程式師的周圍往往是最出色的人員，因此他們可以學習到良好的實踐。因此，美國的重大策略是頒佈各種優秀的現代實踐。新課程、新文獻。象軟體工程研究所SEI等新機構的出現都是爲了把我們的實踐從不足提升到更高的水準。其正確性是勿庸置疑的。

不過，我不認爲我們可以用相同的方式取得下一次進步。低劣設計和良好設計之間的區別

可能在於設計方法中的完善性，而良好設計和卓越設計之間的區別肯定不是如此。卓越設計來自卓越的設計人員。軟體發展是一個創造性的過程。完備的方法學可以培養和釋放創造性的思維，但它無法孕育或激發創造性的過程。

其中的差異並不小——就象薩列裏和莫札特。一個接一個的研究顯示，非常卓越的設計者產生的成果更快、更小、更簡單、更優雅，實現的代價更少。卓越和一般之間的差異接近於一個數量級。

簡單地回顧一下，儘管很多傑出、實用的軟體系統是由很多人共同設計開發，但是那些激動人心、擁有廣大熱情愛好者的產品往往是一個或者少數偉大設計師們的思想。考慮一下Unix、APL、Pascal、Modula、Smalltalk的介面、甚至Fortran；與之對應的產品是Cobol、PL/I、Algol、MVS/370和MS/DOS（圖6.1）。

| **YES** | **NO** |
|---|---|
| Unix | Cobol |
| APL | PL/I |
| Pascal | Algol |
| Modula | MVS/370 |
| Smalltalk | MS-DOS |
| Fortran | |

圖16.1：激動人心的產品

因此，儘管我強烈地支援現在的技術轉移和開發技能的傳授，但我認爲我們可以著手的最重要工作是尋求培養卓越設計人員的途徑。

沒有任何軟體機構可以忽視這項挑戰。儘管公司可能缺少良好的管理人員，但決不會比良好設計人員的需求更加迫切，而卓越的管理人員和設計人員都是非常缺乏的。大多數機構花費了大量的時間和精力來尋找和培養管理人員，但據我所知，它們中間沒有任何的一家在尋求和培育傑出的設計人員上投入相同的資源，而產品的技術特色最終依賴于這些設計人員。

我的第一項建議是每個軟體機構必須決定和表明，傑出的設計人員和卓越的管理人員一樣重要，他們應該得到相同的培養和回報。不僅僅是薪資，還包括各個方面的認可——辦公室規模、安排、個人的設備、差旅費用、人員支持等——必須完全一致。

如何培養傑出的設計人員？限於篇幅，不允許進行較長的介紹，但有些步驟是顯而易見的。

盡可能早地、有系統地識別頂級的設計人員。最好的通常不是那些最有經驗的人員。

爲設計人員指派一位職業導師，負責他們技術方面的成長，仔細地爲他們規劃職業生涯。

爲每個方面制訂和維護一份職業計畫，包括與設計大師的、經過仔細挑選的學習過程、正式的高級教育和以及短期的課程　　所有這些都穿插在設計和技術領導能力的培養安排中。

爲成長中的設計人員提供相互交流和學習的機會。

# 再論《沒有銀彈》（“*No Silver Bullet*”*Refired*）

生死有命，富貴在天

- 威廉三世，奧倫治王子

那些想看到完美方案的人，其實在心底裏就認爲它們以前不存在，以後也不可能出現。

- 亞歷山大·波普，批判散文

*Every bullet has its billet.*

*- WILLIAM III OF ENGLAND, PRINCE OF ORANGE*

*Whoever thinks a faultless piece to see, thinks what ne’er was, nor e’er shall be.*

*- ALEXANDER POPE, AN ESSAY ON CRITISIM*

## 人狼和其他恐怖傳說

《沒有銀彈－軟體工程中的根本和次要問題》（第16章）最初是在IFIP'86年都柏林大會的約稿，接著在一系列的刊物上發表[1]。《電腦》雜誌上翻印了該文章，封面是一副類似於《倫敦人狼》[2]影片的恐怖劇照。同時，還有一欄補充報導《殺死人狼》，描述了銀彈將要完成的（現代）神話。在出版以前，我並未注意到補充報導和文字，也沒有料到一篇嚴肅的技術性文字會被這樣潤色。

Computer雜誌的編輯們是取得他們想要的效果的專家，不過，似乎有很多人閱讀了那篇文章。因此，我爲那一章選擇了另一幅人狼插圖，一幅對這種近乎滑稽物種的古老素描。我希望這副並不刺眼的圖案有相同的正面效果。

## 存在著銀彈－就在這裏！

《沒有銀彈》中聲稱和斷定，在近十年內，沒有任何單獨的軟體工程進展可以使軟體生產率有數量級的提高（引自1986年的版本）。現在已經是第九個年頭，因此也該看看是否這些預言得到了應驗。

《人月神話》一文被大量地引用，很少存在異議；相比之下，《沒有銀彈》卻引發了眾多的辯論，編輯收到了很多文章和信件，至今還在延續[3]。他們中的大多數攻擊其核心論點和我的觀點——沒有神話般的解決方案，以及將來也不會有。他們大都同意《沒有銀彈》一文中的多數觀點，但接著斷定實際存在著殺死軟體怪獸的銀彈——由他們所發明的銀彈。今天，當我重新閱讀一些早期的回饋，我不禁發現在1986年～1987年期間，曾被強烈推崇的秘方並沒有出現所聲稱的戲劇性效果。

在購買電腦軟體和硬體時，我喜歡聽取那些真正使用過產品並感到滿意的用戶的推薦。類似的，我很樂意接受銀彈已經出現的觀點，例如，某個名副其實的中立客戶走到面前，並聲稱，“我使用了這種方法、工具或者產品，它使我的軟體生產率提高了十倍。”

很多書信作者進行了若干正確的修訂和澄清，其中一些還提供了很有針對性的分析和辯駁，對此我非常感激。本章中，我將同大家分享這些改進，以及對反面意見進行討論。

## 含糊的表達將會導致誤解

某些作者指出我沒有將一些觀點表達清楚。

次要（Accident）。在第16章的摘要中，我已經盡我所能地清晰表達了《沒有銀彈》一文的主要觀點。然而，仍有些觀點由於術語“accident（偶然）”和“accidental（次要）”而被混淆，這些術語來自亞裏斯多德的古老用法[4]。術語“accidental”，我不是指“偶然發生”，也不是指“不幸的”，而是更接近於“附帶的”或者“從屬的”。

我並不是貶低軟體構建中的次要部分，相反，我認同英國劇作家、偵探小說作者和神學家桃樂絲·賽爾絲看待創造性活動的觀點，創造性活動包括（1）概念性結構的形式規格化，（2）

使用現實的介質來實現，（3）在實際的使用中，與用戶交互[5]。在軟體發展中，我稱爲“必要（essence）”的部分是構思這些概念上的結構；我稱爲“次要（accident）”的部分指它的實現過程。

現實問題。對我而言（儘管不是所有人），關鍵論點的正確與否歸結爲一個現實問題：整個軟體發展工作中的哪些部分與概念性結構的精確和有序表達相關，哪些部分是創造那些結構的思維活動？根據缺陷是概念性的（例如未能識別某些異常），或者是表達上的問題（例如指標錯誤或者記憶體分配錯誤）等，可以將這些缺陷的尋找和修復工作進行相應的劃分。

在我看來，開發的次要或者表達部分現在已經下降到整個工作的一半或一半以下。由於這部分是現實的問題，所以原則上可以應用測量技術來研究[6]。這樣，我的觀點也可以通過來更科學和更新的估計來糾正。值得注意的是，還沒有人公開發表或者寫信告訴我，次要部分的任務佔據了工作的9/10。

《沒有銀彈》無可爭辯地指出，如果開發的次要部分少於整個工作的9/10，那麼即使不佔用任何時間（除非出現奇跡），也不會給生產率帶來數量級的提高。因此，必須著手解決開發的根本問題。

由於《沒有銀彈》，Bruce Blum把我的注意力引向Herzberg、Mausner和Sayderman[7]等人在1959年的研究。他們發現動機因素可以提高生產率。另一方面，環境和次要因素，無論起到多麼積極的作用，仍無法提高生產率。但是在產生負面影響時，它們會使生產率降低。《沒有銀彈》認爲很多軟體發展過程已經消除了以下負面因素：十分笨拙的機器語言、漫長的批次處理周轉時間以及無法忍受的記憶體限制。

因爲是根本困難所以沒有希望？1990年Brad Cox的一篇非常出色的論文《這就是銀彈》（There Is a Silver Bullet），有說服力地指出重用和交互的構件開發是解決軟體根本困難的一種方法[8]。我由衷地表示贊同。

不過，Cox在兩點上誤解了《沒有銀彈》。首先，他斷定軟體困難來自“編程人員缺乏構建當今軟體的技術”。而我認爲根本困難是固有的概念複雜性，無論是任何時間，使用任何方法設計和實現軟體的功能，它都存在。其次，他（以及其他人）閱讀《沒有銀彈》，並認定文中的觀點是沒有任何處理軟體發展中根本困難的希望——這不是我的本意。作爲本質上的困難，構思軟體概念性的結構本身就有複雜性、一致性、可變性及不可見性的特點。不過實際上，每一種困難產生的麻煩都是可以改善的。

複雜性是層次化的。例如，複雜性是最嚴重的內在困難，並不是所有的複雜性都是不可避免的。我們的很多軟體，但不是全部，來自應用本身隨意的複雜特性。來自一家國際管理諮詢公司，MYSIGMA  Lars Sodahl的Lars Sodahl和合作夥伴曾寫道：

就我的經驗而言，在系統工作中所遇到的大多數困難是組織結構上的一些失誤徵兆。試圖爲這些現實建模，建立同等複雜的程式，實際上是隱藏，而不是解決這些雜亂無章的情況。

Northrop的Steve Lukasik認爲即使是組織機構上的複雜性也不是任意的，可能容易受到

策略調整的影響。

我曾作爲物理學家接受過培訓，因此傾向於用更簡單的概念來描述“複雜”事物。現在你可能是正確的，我無法斷定所有的複雜事物都容易用有序的規律表達□□同樣的道理，你不能斷定它們不能。

□□昨天的複雜性是今天的規律。分子的無序性啓迪了氣體動力學理論和熱力學的三大定律。現在，軟體沒有揭示類似的規律性原理，但是解釋爲什麼沒有的重擔在你的身上。我不是遲鈍和好辯的。我相信有一天軟體的“複雜性”將以某種更高級的規律性概念來表達（就像物理學家的不變式）。

我並沒有著手於Lukasik提倡的更深層次的分析。作爲一個學科，我們需要更廣泛的資訊理論，它能夠量化靜態結構的資訊內容，就像針對交互流的香農資訊理論一樣。這已經超越了我的能力。作爲對Lukasik的簡單回應，我認爲系統複雜性是無數細節的函數，這些細節必須精確而且詳細地說明——或者是借助某種通用規則，或者是逐一闡述，但決不僅僅是統計說明。僅靠若干人不相干的工作，是不大可能產生足夠的一致性，能用通用規律進行精確描述。

不過，很多複雜性並不完全是因爲和外部世界保持一致，而是因爲實現的本身，例如資料結構、演算法、互聯性等。而在更高的級別開發（發展）軟體，使用其他人的成果，或者重用自己的程式——都能避免面對整個層次的複雜性。《沒有銀彈》提出了全力解決複雜性問題的方法，這種方法可以在現實中取得十分樂觀的進展。它宣導向軟體系統增加必要的複雜性：

層次化，通過分層的模組或者物件。

增量化，從而系統可以持續地運行。

## **Harel**的分析

David Harel，在1992年的論文《批評銀彈》（Biting the Silver Bullet）中，對已出版的《沒有銀彈》進行了很多最仔細的分析。

悲觀主義 vs. 樂觀主義 vs. 現實主義。Harel同時閱讀了《沒有銀彈》和1984年Parnas的文章《戰略防衛系統的軟體問題》（Software Aspects of Strategic Defense Systems[10]），認爲它們“太過黯淡”。因此，他試圖在論文《走向系統開發的光明未來》（Toward a Brighter Future for System Development）中展現其明亮的一面。Cox同Halel一樣認爲《沒有銀彈》一文過於悲觀，從而他提出“但是，如果從一個新視點去觀察相同的事情，你會得到一個更加樂觀的結論”。他們的論調都有些問題。

首先，我的妻子、同事和我的編輯發現我犯樂觀主義錯誤的幾率遠遠大於悲觀主義。畢竟，我的從業背景是程式師，樂觀主義是這個行業的職業病。

《沒有銀彈》一文明確地指出“我們看看近十年來的情況，沒有銀彈的蹤跡□□懷疑論者並

不是悲觀主義者□□雖然沒有通天大道，但是路就在腳下。”它預言了如果1986年的很多創新能持續開拓和發展，那麼實際上它們的共同作用能使生產率獲得數量級的提高。隨著1986～1996十年過去了，這個預言即使說明了什麼，那也是過於樂觀，而不是過於悲觀。

就算《沒有銀彈》總體看來有些悲觀，那麼到底存在什麼問題？是否愛因斯坦關於任何物體運動的速度無法超過光速的論斷過於“黯淡”或者“令人沮喪”呢？那麼哥德爾關於某些事物無法計算的結論，又如何呢？《沒有銀彈》一文認爲“軟體的特性本身導致了不大可能有任何的銀彈”。Tuski在IFIP大會上發表了一篇論文作爲出色的回應，文中指出：

在所有被誤導的科學探索中，最悲慘的莫過於對一種能夠將一般金屬變成金子的物質，即點金石的研究。這個由統治者不斷地投入金錢，被一代代的研究者不懈追求的、煉金術中至高無上的法寶，是一種從理想化想像和普遍假設中——以爲事情會像我們所認爲的那樣——提取出的精華。它是人類純粹信仰的體現，人們花費了大量的時間和精力來認可和接受這個無法解決的問題。即使被證明是不存在，那種尋找出路和希望能一勞永逸的願望，依然十分的強烈。而我們中的絕大多數總是很同情這些明知不可爲而爲之的人，因此它們總是得以延續。所以，將圓形變方的論文被發表，恢復脫髮的洗液被研製和出售，提高軟體生產率的方法被提出並成功地推銷。

我們太過傾向於遵循我們自己的樂觀主義（或者是發掘我們出資人的樂觀主義）。我們太喜歡忽視真理的聲音，而去聽從萬靈藥販賣者的誘惑[11]。

我和Turski都堅持認爲這個白日夢限制了向前的發展，浪費了精力。

“消極”主題。Hare1認識到《沒有銀彈》中的消極來自三個主題：

根本和次要問題的清晰劃分

獨立地評價每個候選銀彈

僅僅預言了十年，而不是足夠長的時間　出現任何重大的進步。

第一個主題，它是整篇文章的主要觀點。我仍然認爲上述劃分對於理解爲什麼軟體難以開發是絕對關鍵的。對於應該做出哪些方面的改進，它也是十分明確的指南。

至於獨立地考慮不同的候選銀彈，《沒有銀彈》並非如此。各種各樣的技術一個接一個地被提出，每一種都過分宣揚自身的效果。因此，依次獨立的評估它們是非常公平的。我持反對態度的並不是這些技術，而是那種它們能起到魔術般作用的觀點。Glass、Vessey和Conger在他們1992年的論文中提供了充足的證據，指出對銀彈的無謂研究仍未結束[12]。

關於選擇10年還是40年作爲預言的期限，選擇較短的時間是承認我們並沒有足夠強的能力可以預見到十年以後的事情。我們中間有誰可以在1975年預見到80年代的微型電腦革命嗎？

對於十年的期限，還有其他的一些原因：各種銀彈都宣稱它們能夠立刻取得效果。我回顧了一下，發現沒有任何一種銀彈聲稱　向我的秘方投資，在十年後你將獲得成功　。另外，硬體的性能/價格比可能每十年就會有成百倍的增長，儘管這種比較不很合適，但是直覺上的確如此。我們確信會在下一個40年中取得穩步的發展。不過，以40年代價取得數量級的進展，很難被認爲是不可思議的進步。

想像的試驗。Hare1建議了一種想像的試驗，他假設《沒有銀彈》是發表在1952年，而不是1986年，不過表達的論斷相同。他使用反證法來證明將根本和次要問題分開是不恰當的。

這種觀點站不住腳。首先，《沒有銀彈》一開始就聲稱，50年代的程式開發中曾占支配地位的次要困難，如今已經不存在了，並且消除這些困難已經產生了提高若干數量級的效果。將辯論推回到40年前是不合理的，在1952年，甚至很難想像開發的次要問題不會佔據開發工作的主要部分。

其次，Hare1所設想50年代行業所處狀態是不準確的：

當時已經不是構建大型複雜系統的時代，程式師的工作模式已經成爲常規個人程式的開發（在現代的編程語言中，大概是100～200行代碼）。在已有技術和方法學的前提下，這些任務是令人恐怖的，處處都是錯誤、故障和落後的完成期限。

接著，他闡述了在傳統的小型個人程式中，那些假設的錯誤、故障和落後的最終期限如何在接下來的25年中，得到數量級的改進和提高。

但是，50年代該領域的實際情況並不是小型個人程式。在1952年，Univac還在使用大約8人開發的複雜程式處理1950年的人口普查[13]。其他機器則用於化學動力學、中子漫射計算、導彈性能計算等等[14]。組合語言、重定位的鏈結和裝載程式、浮點解釋系統等，還經常被使用[15]。1955年，人們開發50～100人年的商用程式[16]。1956年，通用電氣在路易斯維爾的設備車間使用著超過80,000指令的薪資系統。1957年，SAGE ANFSQ/7防空電腦系統已經運轉了兩年，這個系統分佈在30個不同的地點，是基於通訊、自消除故障的熱備即時系統[17]。因此，幾乎無法堅持說個人程式的技術革命，能夠用來描述1952年以來的軟體工程上的努力。

銀彈就在這裏。Hare1接著提出了他自己的銀彈，一種稱爲　香草（Vanilla）框架　的建模技術。文中並沒有對方法提供足夠評估的詳細描述，不過給出了一些論文和參考資料[18]。建模所針對的確實是軟體發展的根本困難，即概念性要素的設計和調試，因此Vanilla框架有可能是革命性的。我也希望如此。Ken Brooks在報告中提到，在實際工作中應用時，它的確是一種頗有幫助的方法學。

不可見性。Hare1強烈地主張軟體的概念性要素本質上是拓撲的，這些關係可以用空間/圖形方式來自然地表達：

適當使用視覺化圖形可以給工程師和程式師帶來可觀的成效。而且，這種效果並不僅僅局限於次要問題，開發人員思考和探索的品質也得到了改進。未來的成功系統的開發將圍繞在視

覺化圖形表達方式的周圍。首先，我們會使用　合適的　實體和關係來形成概念，然後表達成一系列逐步完善的模型，不斷地系統化闡明和精化設計概念。模型用若干視覺化語言的適當組合來描述，它必須是多種語言的組合，因爲系統模型具有若干方面的內容，每方面象變戲法般產生不同類型的思維圖像。

⇦⇦就使自己成爲良好視覺化表達方式而言，建模過程的某些方面並不會立刻出現改觀。例如，變數和資料結構上的演算法操作可能還是會採用文字性描述。

我和Hare1頗爲一致。我認爲軟體要素並不存在於三維空間中，因此並不存在概念性設計到圖形簡單兩維或三維上的映射。他承認，我也同意　　這需要多種圖形，每種圖形覆蓋某個特定的方面，而且有些方面無法用圖形來表達。

Hare1採用圖形來輔助思考和設計的熱情徹底地感染了我。我一直喜歡向准程式師提問，　下個十一月在哪？　如果覺得問題過於模糊，接著我會問，　告訴我，你自己關於時間曆法的模型。　優秀程式師具有很強的空間想像能力，他們常常有時間的幾何模型，而且無需考慮，就能理解第一個問題。他們往往擁有高度個性化的模型。

## **Jone**的觀點——品質帶來生產率

Capers Jones最開始在一系列備忘錄裏，而後在一本書裏，提出了頗有洞察力的觀點。很多和我有書信往來的人向我提到了他的觀點，《沒有銀彈》如同當時的很多文章，關注於生產率——單位輸入對應的軟體產出。Jone提出，“不。關注品質，生產率自然會隨著提高。19”他認爲，很多代價高昂的後續專案投入了大量的時間和精力來尋找和修復規格說明中、設計和實現上的錯誤。他提供的資料顯示了缺乏系統化品質控制和進度災難之間的密切關係。我認同這些資料。不過，Boehm指出，如果一味追求完美品質，生產率會像IBM的航太軟體一樣再次下降。

Coqui也提出相似的主張：系統化軟體發展方法的發展是爲了解決品質問題（特別是避免大型的災難），而不是出於生產率方面的考慮。

但是注意：70年代，在軟體生產上應用工程原理的目標是提高軟體產品的品質、可測試性、穩定性以及可預見性——而不是軟體產品的開發效率。

在軟體生產上應用工程原理的驅動力是擔心擁有無法控制的“藝術家們”而可能導致的巨大災難，他們往往對異常複雜系統開發承擔責任20。

## 那麼，生產率的情形如何？

生產率數據。生產率的資料非常難以定義、測量和尋找。Capers Jones相信兩個相隔十年、完全等同的COBOL程式，一個採用結構化方法開發，另一個不使用結構化方法，它們之間的差距是3倍。

Ed Yourdon說，“由於工作站和軟體工具，我看到了人們的工作獲得了5倍的提高。”Tom DeMarco認爲“你的期望——十年內，由於所有的技術而使生產率得到數量級的提高——太樂觀了。我沒有看到任何機構取得數量級的進步。”

塑膠薄膜包裝的成品軟體——購買，而非開發。我認爲1986年《沒有銀彈》中的一個估計被證實是正確的：“我相信，這個大眾市場是□□軟體工程領域意義最深遠的開發方向。”從學科的角度說，不管和內部還是外部客戶軟體的開發相比，大眾市場軟體都幾乎是一個嶄新的領域。當套裝軟體的銷量一旦達到百萬或者即使只是幾千，這時關鍵的支配性問題就變成了品質、時機、產品性能和支撐成本，而不再是對於客戶系統異常關鍵的開發成本。

創造性活動的強大工具。提高資訊管理系統（MIS）編程人員生產率最戲劇化的方法是到一家電腦商店去，購買理應由他們開發的商業成品。這並不荒唐可笑。價格低廉、功能強大的薄膜包裝軟體已經能滿足要求，而以前這會要求進行定制套裝軟體的開發。比起複雜的大型產品工具，它們更加像電鋸、電轉和砂磨機。把它們組合成相容互補的集合，像Microsoft Works和集成更好的Claris Works一樣，能夠帶來巨大的靈活性。另外，象供人們使用的組合工具箱，其中的某些工具會經常被使用，以致於熟能生巧。這種工具必須注重常人使用的方便，而不是專業。

Ivan Selin，美國管理系統公司主席，在1987年曾寫信給我：

我有些懷疑你的關於套裝軟體沒有真正地改變很多□□的觀點。我覺得你太過輕易地拋開了你的觀察所蘊涵的事實；你觀察到——[套裝軟體]“可能比以前更加通用和更加容易定制一些，但並不太多。”即使在表面上接受了這種論述，我相信用戶察覺到套裝軟體更加通用和易於本地化，這種感覺使用戶更容易接受套裝軟體。在我公司發現的大多數情況中，是[最終用戶]，而不是軟體人員，不願意使用套裝軟體，因爲他們認爲會失去必要的特性或功能。因此，對他們而言，易於定制是一個非常大的賣點。

我認爲Selin是十分正確的——我低估了套裝軟體客戶化的程度和它的重要性。

## 面向物件編程——這顆銅質子彈可以嗎？

本章一開始的描述提醒我們，當很多零件需要裝配，而且每個零件可能很複雜時，如果它們的介面設計得很流暢，大量豐富的結構就能快速地組合在一起。

使用更大的零件來構建。面向物件編程的第一個特徵是，它強制的模組化和清晰的介面。其次，它強調了封裝，即外界無法看到元件的內部結構；它還強調了繼承和層次化類結構以及虛函數。面向物件還強調了強抽象資料類型化，它確保某種特定的資料類型只能由它自身的相應函數來操作。

現在，無需使用整個Smalltalk或者C++的套裝軟體，就可以獲得這些特點中的任意一個——其中一些甚至出現在面向物件技術之前。面向物件方法吸引人的地方類似于複合維他命藥丸：一次性（即編程人員的培訓）得到所有的好處。面向物件是一種非常有前途的概念。

面向物件技術爲什麼發展緩慢？《沒有銀彈》後的九年中，對面向物件技術的期望穩步增

長。爲什麼增長如此緩慢？理論過多。James Coggins，已經在The C++ Report做了四年 “The best of comp.lang.c++”專欄的作者，他提出了這樣的解釋：

問題是O-O程式師經歷了很多錯綜複雜混亂的應用，他們所關注的是低層次，而不是高層次的抽象。例如，他們開發了很多象鏈表或集合這樣的類，而不是用戶介面、射線束模型或者有限元素模型。不幸的是，C++中幫助程式師避免錯誤的強類型檢查，使得從小型事物中構建大型物體非常困難[21]。

他回歸到基本的軟體問題，主張一種解決軟體不能滿足要求的方法，即通過客戶的參與和協作來提高腦力勞動的規模。他贊同自頂向下的設計：

如果我們設計大粒度的類，關注用戶已經接觸的概念，則在進行設計的時候，他們能夠理解設計並提出問題，並且可以幫助設計測試用例。我的眼科客戶並不關心堆疊，他們關心描述眼角膜形狀的勒讓德多項式。在這方面，小規模的封裝帶來的好處比較少。

David Parnas的論文是面向物件概念的起源之一，他用不同的觀點看這個問題。他寫信給我：

答案很簡單。因爲[O-O]和各種複雜語言的聯繫已經很緊密。人們並沒有被告訴O-O是一種設計的方法，並向他們講授設計方法和原理，大家只是被告知O-O是一種特殊工具。而我們可以用任何工具寫出優質或低劣的代碼。除非我們給人們講解如何設計，否則語言所起的作用非常小。結果是人們使用這種語言做出不好的設計，沒有從中獲得什麼價值。而一旦獲得的價值少，它就不會流行。

資金的先行投入，收益的後期獲得。面向物件技術包含了很多方法學上的進步。面向物件技術的前期投入很多——主要是培訓程式師用很新的方法思考，同時還要把函數打造成通用的類。我認爲它的好處是客觀實在的，並非僅僅是推測。面向物件應用在整個開發週期中，但是真正的獲益只有在後續開發、擴展和維護活動中才能體現出來。Coggin說：“面向物件技術不會加快首次或第二次的開發，產品族中第五個專案的開發才會異乎尋常的迅速。[22]”

爲了預期中的，但是有些不確定的收益，冒著風險投入金錢是投資人每天在做的事情。不過，在很多軟體公司中，這需要真正的管理勇氣，一種比技術競爭力或者優秀管理能力更少有的精神。我認爲極度的前期投入和收益的推後是使O-O技術應用遲緩的最大原因。即使如此，在很多機構中，C++仍毫無疑問地取代了C。

## 重用的情況怎樣？

解決軟體構建根本困難的最佳方法是不進行任何開發。套裝軟體只是達到上述目標的方法之一，另外的方法是程式重用。實際上，類的容易重用和通過繼承方便地定制是面向物件技術最吸引人的地方。

事情常常就是這樣。當某人在新的做事方法上取得了一些經驗，新模式就不再象一開始那麼簡單。

當然，程式師經常重用他們自己的手頭工作。Jone提到：

大多數有豐富經驗的程式師擁有自己的私人開發庫，可以使他們使用大約30%的重用代碼來開發軟體。公司級別的重用能提供70%的重用代碼量，它需要特殊的開發庫和管理支援。公司級別的重用代碼也意味著需要對專案中的變更進行統計和度量，從而提高重用的可信程度[23]。

W.Huang建議用責任專家的矩陣管理來組織軟體工廠，從而培養重用自身代碼的日常工作習慣[24]。

JPL的Van Snyder向我指出，數學軟體領域有著軟體重用的長期傳統：

我們推測重用的障礙不在生產者一邊，而在消費者一邊。如果一個軟體工程師，潛在的標準化軟體構件消費者，覺得尋找能滿足他需要的構件，進行驗證，比自行編寫的代價更加昂貴時，重複的構件就會產生。注意我們上面提到的"覺得"。它和重新開發的真正投入無關。

數學軟體上重用成功的原因有兩個：（1）它很晦澀難懂，每行代碼需要大量高智商的輸入；（2）存在豐富的標準術語，也就是用數學來描述每個構件的功能。因此，重新開發數學軟體構件的成本很高，而查找現有構件功能的成本很低。數學軟體界存在一些長期的傳統——例如，專業期刊和演算法搜集，用適度成本提供演算法，出於商業考慮開發的高品質演算法（儘管成本有些高，但依舊適度）等——使查找和發現滿足某人需要的構件比其他很多領域要容易。其他領域中，有時甚至不可能簡潔地提出明確的要求。這些因素合在一起，使數學軟體的重用比重新開發更有吸引力。

同樣的原因，在很多其他領域中也可以發現相同的重用現象，如那些爲核反應、天氣模型、海洋模型開發軟體的代碼編制工作。這些領域都是在相同的課本和標準概念下逐步地發展起來的。

現在公司級別的重用情況如何？存在著大量的研究。美國國內的實踐相對較少，有報導聲稱國外重用較多[25]。

Jones報告，在他公司的客戶中，所有擁有5000名以上程式師的機構都進行正式的重用研究，而500名以下程式師的組織，只有不到10%著手重用研究[26]。報告指出，最具有重用潛質的企業中，重用性研究（而非部署）"是活躍和積極的，即使沒有完全成功。"Ed Yourdon報告，有一家馬尼拉的軟體公司，200名程式師中有50名從事供其他人使用的重用模組的開發，"我所見到的個案非常少——是由於機構上因素而進行重用研究，而不是技術上的原因"。

DeMarco告訴我，大眾市場套裝軟體提供了資料庫系統等通用功能，充分地減輕了壓力，減少了處在重用模組邊緣的開發。 "不管怎樣，重用的模組一般是一些通用功能。"

Parnas寫道：

重用是一件說起來容易，做起來難的事情。它同時需要良好的設計和文檔。即使我們看到了並不十分常見的優秀設計，但如果沒有好的文檔，我們也不會看到能重用的構件。

Ken Brooks關於預測產品通用化的一些困難的評論：“我不斷地進行修改，即使在第五次使用我自己的個人用戶介面庫的時候。”

真正的重用似乎才剛剛開始。Jones報告，在開放市場上僅有少量的重用代碼模組，它們的價格在常規開發成本的1%～20%[27]。DeMacro說：

對整個重用現象，我變得有些氣餒。對於重用，現有理論幾乎是整體缺乏。時間證明了使模組能夠重用的成本非常高。

Yourdon估計了這個高昂的費用：“一個良好的經驗法則是可重用的構件的工作量是‘一次性’構件的兩倍。[28]”在第一章的討論中，我觀察到了真正產品化構件所需的成本。因此，我對工作量比率的估計是三倍。

顯然，我們正在看到很多重用的形式和變化，但離我們所期望的還遠，還有很多需要學習的地方。

## 學習大量的辭彙——對軟體重用的一個可預見，但還沒有被預言的問題

思索的層次越高，所需要處理的基本思考要素也就越多。因此，編程語言比機器語言更加複雜，而自然語言的複雜程度更高。高階語言有更廣泛的辭彙量、更複雜的語法以及更加豐富的語義。

作爲一個科目，我們並沒有就程式重用的實際情況，仔細考慮它蘊涵的意義。爲了提高品質和生產率，我們需要通過經過調試的大型要素來構建系統，在編程語言中，這些函數的級別遠遠高於語句。所以，無論採用物件類庫還是函數庫的方式，我們必須面對我們編程辭彙規模徹底擴大的事實。對於重用，詞彙學習並不是思維障礙中的一小部分。

現在人們擁有成員超過3000個的類庫。很多物件需要10到20個參數和可選變數的說明。如果想獲得所有潛在的重用，任何使用類庫編程的人員必須學習其成員的語法（外部介面）和語義（詳細的功能行爲）。

這項工作並不是沒有希望的。一般人日常使用的辭彙超過10,000個，受過教育的人遠多於這個數目。另外，我們在自然而然地學習著語法和非常微妙的語義。我們可以正確地區分巨大、大、遼闊、大量和龐大。人們不會說：龐大的沙漠或者遼闊的大象。

對軟體重用問題，我們需要研究適當的學問，瞭解人們如何擁有語言。一些經驗教訓是顯而易見的：

人們在上下文中學習，所以我們需要出版一些複合產品的例子，而不僅僅是零部件的庫。

人們只會記憶背誦單詞。語法和語義是在上下文中，通過使用逐漸地學習。

人們根據語義上的分類對辭彙組合規則進行分組，而不是通過比較對象子集。

## 子彈的本質　　形勢沒有發生改變

現在，我們回到基本問題。複雜性是我們這個行業的屬性，而且複雜性是我們主要的限制。R.L.Glass在1988年的文字精確地總結了我在1995年的看法：

又怎麼樣呢？Parnas和Brooks不是已經告訴我們了嗎？軟體發展是一件棘手的事情，前方並不會有魔術般的解決方案。現在是從業者研究和分析革命性進展的時刻，而不是等待或希望它的出現。

軟體領域中的一些人發現這是一幅使人洩氣的圖畫。他們是那些依然認爲突破近在眼前的人們。

但是我們中的一些——那些非常固執，以致於可以認爲是現實主義者的人——把它看成是清新的空氣。我們終於可以將焦點集中在更加可行的事情上，而不是空中的餡餅。現在，有可能，我們可以在軟體生產率上取得逐步的進展，而不是等待不大可能到來的突破[29]。

# 《人月神話》的觀點：是或非？（***Propositions of* the Mythical Man-Month: *True or False*？）**

我們理解的也好，不理解的也好，描述都應該簡短精練。

撒母耳·巴特勒，諷刺詩

*For brevity is very good, where we are, or are not understood.*

*SAMUEL BUTLER Hudibras*

現在我們對軟體工程的瞭解比1975年要多得多。那麼在1975年版本的人月神話中，哪些觀點得到了資料和經驗的支援？哪些觀點被證明是不正確的？又有哪些觀點隨著世界的變化，顯得陳舊過時呢？爲了幫助判斷，我將1975年書籍中的論斷毫無更改地抽取出來，以摘要的形式列舉在下面——它們是當年我認爲將會是正確的：客觀事實和經驗中推廣的法則。（你也許會問，“如果這些就是那本書講的所有東西，爲什麼要花177頁的篇幅來論述？”）方括號中的評論是新增內容。

所有這些觀點都是可操作驗證的，我將它們表達成刻板的形式是希望能引起讀者的思考、判斷和討論。

# 第1章 焦油坑

1.1 編程系統產品（Programming Systems Product）開發的工作量是供個人使用的、獨立開發的構件程式的九倍。我估計軟體構件產品化引起了3倍工作量，將軟體構件整合成完整系統所需要的設計、集成和測試又強加了3倍的工作量，這些高成本的構件在根本上是相互獨立的。

1.2 編程行業“滿足我們內心深處的創造渴望和愉悅所有人的共有情感”，提供了五種樂趣：

創建事物的快樂

開發對其他人有用的東西的樂趣

將可以活動、相互嚙合的零部件組裝成類似迷宮的東西，這個過程所體現出令人神魂顛倒的魅力

面對不重複的任務，不間斷學習的樂趣

工作在如此易於駕馭的介質上的樂趣　　純粹的思維活動，其存在、移動和運轉方式完全不同於實際物體

1.3 同樣，這個行業具有一些內在固有的苦惱：

將做事方式調整到追求完美，是學習編程的最困難部分

由其他人來設定目標，並且必須依靠自己無法控制的事物（特別是程式）；權威不等同于責任

實際情況看起來要比這一點好一些：真正的權威來自于每次任務的完成

任何創造性活動都伴隨著枯燥艱苦的勞動，編程也不例外

人們通常期望項目在接近結束時，（bug、工作時間）能收斂得快一些，然而軟體專案的情況卻是越接近完成，收斂得越慢

產品在即將完成時總面臨著陳舊過時的威脅

# 第2章 人月神話

2.1 缺乏合理的時間進度是造成專案滯後的最主要原因，它比其他所有因素加起來影響還大。

2.2 良好的烹飪需要時間，某些任務無法在不損害結果的情況下加快速度。

2.3 所有的編程人員都是樂觀主義者："一切都將運作良好"。

2.4 由於編程人員通過純粹的思維活動來開發，所以我們期待在實現過程中不會碰到困難。

2.5 但是，我們的構思是有缺陷的，因此總會有bug。

2.6 我們圍繞成本核算的估計技術，混淆了工作量和專案進展。人月是危險和帶有欺騙性的神話，因爲它暗示人員數量和時間是可以相互替換的。

2.7 在若干人員中分解任務會引發額外的溝通工作量——培訓和相互溝通。

2.8 關於進度安排，我的經驗是爲1/3計畫、1/6編碼、1/4構件測試以及1/4系統測試。

2.9 作爲一個學科，我們缺乏資料估計。

2.10 因爲我們對自己的估計技術不確定，所以在管理和客戶的壓力下，我們常常缺乏堅持的勇氣。

2.11 Brook法則：向進度落後的專案中增加人手，只會使進度更加落後。

2.12 向軟體專案中增派人手從三個方面增加了項目必要的總體工作量：任務重新分配本身和所造成的工作中斷；培訓新人員；額外的相互溝通。

# 第3章 外科手術隊伍

3.1 同樣有兩年經驗而且在受到同樣的培訓的情況下，優秀的專業程式師的工作效率是

較差程式師的十倍。（Sackman、Erikson和Grand）

3.2 Sackman、Erikson和Grand的資料顯示經驗和實際表現之間沒有相互聯繫。我懷疑這種現象是否普遍成立。

3.3 小型、精幹隊伍是最好的——盡可能的少。

3.4 兩個人的團隊，其中一個專案經理，常常是最佳的人員使用方法。[留意一下上帝對婚姻的設計。]

3.5 對於真正意義上的大型系統，小型精幹的隊伍太慢了。

3.6 實際上，絕大多數大型編程系統的經驗顯示出，一擁而上的開發方法是高成本、速度緩慢、不充分的，開發出的產品無法進行概念上的集成。

3.7 一位首席程式師、類似于外科手術隊伍的團隊架構提供了一種方法——既能獲得由少數頭腦產生的產品完整性，又能得到多位協助人員的總體生產率，還徹底地減少了溝通的工作量。

# 第4章 貴族專制、民主政治和系統設計

4.1 “概念完整性是系統設計中最重要的考慮因素”。

4.2 “功能與理解上的複雜程度的比值才是系統設計的最終測試標準”，而不僅僅是豐富的功能。[該比值是對易用性的一種測量，由簡單和複雜應用共同驗證。]

4.3 爲了獲得概念完整性，設計必須由一個人或者具有共識的小型團隊來完成。

4.4 “對於非常大型的專案，將設計方法、體系結構方面的工作與具體實現相分離是獲得概念完整性的強有力方法。”[同樣適用於小型專案。]

4.5 “如果要得到系統概念上的完整性，那麼必須控制這些概念。這實際上是一種無需任何歉意的貴族專制統治。”

4.6 紀律、規則對行業是有益的。外部的體系結構規定實際上是增強，而不是限制實現小組的創造性。

4.7 概念上統一的系統能更快地開發和測試。

4.8 體系結構(architecture)、設計實現(implementation)、物理實現(realization)的許多工作可以併發進行。[軟體和硬體設計同樣可以並行。]

# 第5章 畫蛇添足

5.1 儘早交流和持續溝通能使結構師有較好的成本意識，以及使開發人員獲得對設計的信心，並且不會混淆各自的責任分工。

5.2 結構師如何成功地影響實現：

牢記是開發人員承擔創造性的實現責任；結構師只能提出建議。

時刻準備著爲所指定的說明建議一種實現的方法，準備接受任何其他可行的方法。

對上述的建議保持低調和平靜。

準備對所建議的改進放棄堅持。

聽取開發人員在體系結構上改進的建議。

5.3 第二個系統是人們所設計的最危險的系統，通常的傾向是過分地進行設計。

5.4 OS/360是典型的畫蛇添足（second-system effect）的例子。[Windows NT似乎是90年代的例子。]

5.5 爲功能分配一個位元組和微秒的優先權值是一個很有價值的規範化方法。

# 第6章 貫徹執行

6.1 即使是大型的設計團隊，設計結果也必須由一個或兩個人來完成，以確保這些決定是一致的。

6.2 必須明確定義體系結構中與先前定義不同的地方，重新定義的詳細程度應該與原先的說明一致。

6.3 出於精確性的考慮，我們需要形式化的設計定義，同樣，我們需要記敍性定義來加深理解。

6.4 必須採用形式化定義和記敍性定義中的一種作爲標準，另一種作爲輔助措施；它們都可以作爲表達的標準。

6.5 設計實現，包括模擬仿真，可以充當一種形式化定義的方法；這種方法有一些嚴重

的缺點。

6.6 直接整合是一種強制推行軟體的結構性標準的方法。[硬體上也是如此——考慮內建在ROM中的Mac WIMP介面。]

6.7 “如果起初至少有兩種以上的實現，那麼（體系結構）定義會更加整潔，會更加規範。”

6.8 允許體系結構師對實現人員的詢問做出電話應答解釋是非常重要的，並且必須進行日誌記錄和整理發佈。[電子郵件是一種可選的介質。]

6.9 “項目經理最好的朋友就是他每天要面對的敵人——獨立的產品測試機構/小組。”

# 第7章 爲什麼巴比倫塔會失敗？

7.1 巴比倫塔專案的失敗是因爲缺乏交流，以及交流的結果——組織。

交流

7.2 “因爲左手不知道右手在做什麼，從而進度災難、功能的不合理和系統缺陷紛紛出現。”由於對其他人的各種假設，團隊成員之間的理解開始出現偏差。

7.3 團隊應該以盡可能多的方式進行相互之間的交流：非正式、常規項目會議，會上進行簡要的技術陳述、共用的正式項目工作手冊。[以及電子郵件。]

專案工作手冊

7.4 專案工作手冊“不是獨立的一篇文檔，它是對專案必須產生的一系列文檔進行組織的一種結構。”

7.5 “專案所有的文檔都必須是該（工作手冊）結構的一部分。”

7.6 需要儘早和仔細地設計工作手冊結構。

7.7 事先制訂了良好結構的工作手冊“可以將後來書寫的文字放置在合適的章節中”，並且可以提高產品手冊的品質。

7.8 “每一個團隊成員應該瞭解所有的材料（工作手冊）。”[我想說的是，每個團隊成員應該能夠看到所有材料，網頁即可滿足要求。]

7.9 即時更新是至關重要的。

7.10 工作手冊的使用者應該將注意力集中在上次閱讀後的變更，以及關於這些變更重要性的評述。

7.11 OS/360專案工作手冊開始採用的是紙介質，後來換成了微縮膠片。

7.12 今天[即使在1975年]，共用的電子手冊是能更好達到所有這些目標、更加低廉、更加簡單的機制。

7.13 仍然需要用變更條和修訂日期[或具備同等功能的方法]來標記文字；仍然需要後進先出（LIFO）的電子化變更小結。

7.14 Parnas強烈地認爲使每個人看到每件事的目標是完全錯誤的；各個部分應該被封裝，從而沒有人需要或者允許看到其他部分的內部結構，只需要瞭解介面。

7.15 Parnas的建議的確是災難的處方。[Parnas讓我認可了該觀點，使我徹底地改變了想法。]

組織架構

7.16 團隊組織的目標是爲了減少必要的交流和協作量。

7.17 爲了減少交流，組織結構包括了人力劃分（division of labor）和限定職責範圍（specialization of function）。

7.18 傳統的樹狀組織結構反映了權力的結構原理——不允許雙重領導。

7.19 組織中的交流是網狀，而不是樹狀結構，因而所有的特殊組織機制（往往體現成組織結構圖中的虛線部分）都是爲了進行調整，以克服樹狀組織結構中交流缺乏的困難。

7.20 每個子項目具有兩個領導角色——產品負責人、技術主管或結構師。這兩個角色的職能有著很大的區別，需要不同的技能。

7.21 兩種角色中的任意組合可以是非常有效的：

產品負責人和技術主管是同一個人。

產品負責人作爲總指揮，技術主管充當其左右手。

技術主管作爲總指揮，產品負責人充當其左右手。

# 第8章 胸有成竹

8.1 僅僅通過對編碼部分的估計，然後乘以任務其他部分的相對係數，是無法得出對整項工作的精確估計的。

8.2 構建獨立小型程式的資料不適用於編程系統專案。

8.3 程式開發呈程式規模的指數增長。

8.4 一些發表的研究報告顯示指數約爲1.5。[Boehm的資料並不完全一致，在1.05和1.2之間變化。[1]]

8.5 Portman的ICL資料顯示相對於其他活動開銷，全職程式師僅將50%的時間用於編程和調試。

8.6 IBM的Aron資料顯示，生產率是系統各個部分交互的函數，在1.5K千代碼行/人年至10K千代碼行/人年的範圍內變化。

8.7 Harr的Be11實驗室資料顯示對於已完成的產品，作業系統類的生產率大約是0.6KLOC/人年，編譯類工作的生產率大約爲2.2KLOC/人年。

8.8 Brooks的OS/360S資料與Harr的資料一致：作業系統0.6～0.8KLOC/人年，編譯器2～3 KLOC/人年。

8.9 Corbato的MIT專案MULTICS資料顯示，在作業系統和編譯器混合類型上的生產率是1.2KLOC/人年，但這些是PL/I的代碼行，而其他所有的資料是彙編代碼行。

8.10 在基本語句級別，生產率看上去是個常數。

8.11 當使用適當的高階語言時，程式編制的生產率可以提高5倍。

# 第9章 削足適履

9.1 除了運行時間以外，所佔據的記憶體空間也是主要開銷。特別是對於作業系統，它的很多程式是永久駐留在記憶體中。

9.2 即便如此，花費在駐留程序所佔據記憶體上的金錢仍是物有所值的，比其他任何在配置上投資的效果要好。規模本身不是壞事，但不必要的規模是不可取的。

9.3 軟體發展人員必須設立規模目標，控制規模，發明一些減少規模的方法——就如同硬體開發人員爲減少元器件所做的一樣。

9.4 規模預算不僅僅在佔據記憶體方面是明確的，同時還應該指明程式對磁片的訪問次數。

9.5 規模預算必須與分配的功能相關聯；在指明模組大小的同時，確切定義模組的功能。

9.6 在大型的團隊中，各個小組傾向於不斷地局部優化，以滿足自己的目標，而較少考慮隊用戶的整體影響。這種方向性的問題是大型項目的主要危險。

9.7 在整個實現的過程期間，系統結構師必須保持持續的警覺，確保連貫的系統完整性。

9.8 培養開發人員從系統整體出發、面向用戶的態度是軟體編程管理人員最重要的職能。

9.9 在早期應該制訂策略，以決定用戶可選項目的粗細程度，因爲將它們作爲整體大包能夠節省記憶體空間。[常常還可以節約市場成本。]

9.10 臨時空間的尺寸，以及每次磁片訪問的程式數量是很關鍵的決策，因爲性能是規模的非線性函數。[這個整體決策已顯得過時——起初是由於虛擬記憶體，後來則是成本低廉的記憶體。現在的用戶通常會購買能容納主要應用程式所有代碼的記憶體。]

9.11 爲了取得良好的空間－時間折衷，開發隊伍需要得到特定與某種語言或者機型的編程技能培訓，特別是在使用新語言或者新機器時。

9.12 編程需要技術積累，每個專案需要自己的標準元件庫。

9.13 庫中的每個元件需要有兩個版本，運行速度較快和短小精煉的。[現在看來有些過時。]

9.14 精煉、充分和快速的程式。往往是戰略性突破的結果，而不僅僅技巧上的提高。

9.15 這種突破常常是一種新型演算法。

9.16 更普遍的是，戰略上突破常來自於資料或表的重新表達。資料的表現形式是編程的根本。

# 第10章 提綱挈領

10.1 “前提：在一片檔的汪洋中，少數文檔形成了關鍵的樞紐，每個專案管理的工作都圍繞著它們運轉。它們是經理們的主要個人工具。”

10.2 對於電腦硬體開發專案，關鍵文檔是目標、手冊、進度、預算、組織機構圖、空間分配、以及機器本身的報價、預測和價格。

10.3 對於大學科系，關鍵文檔類似：目標、課程描述、學位要求、研究報告、課程表和課程的安排、預算、教室分配、教師和研究生助手的分配。

10.4 對於軟體專案，要求是相同的：目標、用戶手冊、內部文檔、進度、預算、組織機構圖和工作空間分配。

10.5 因此，即使是小型專案，專案經理也應該在專案早期規範化上述的一系列文檔。

10.6 以上集合中每一個文檔的準備工作都將注意力集中在對討論的思索和提煉，而書寫這項活動需要上百次的細小決定，正是由於它們的存在，人們才能從令人迷惑的現象中得到清晰、確定的策略。

10.7 對每個關鍵文檔的維護提供了狀態監督和預警機制。

10.8 每個文檔本身就可以作爲檢查列表或者資料庫。

10.9 專案經理的基本職責是使每個人都向著相同的方向前進。

10.10 項目經理的主要日常工作是溝通，而不是做出決定；文檔使各項計畫和決策在整個團隊範圍內得到交流。

10.11 只有一小部分管理人員的時間——可能只有20%——用來從自己頭腦外部獲取資訊。

10.12 出於這個原因，廣受吹捧的市場概念——支援管理人員的“完備資訊管理系統”並不基於反映管理人員行爲的有效模型。

# 第11章 未雨綢繆

11.1 化學工程師已經認識到無法一步將實驗室工作臺上的反應過程移到工廠中，需要一個實驗性工廠（pilot planet）來爲提高產量和在缺乏保護的環境下運作提供寶貴經驗。

11.2 對於編程產品而言，這樣的中間步驟是同樣必要的，但是軟體工程師在著手發佈

產品之前，卻並不會常規地進行試驗性系統的現場測試。[現在，這已經成爲了一項普遍的實踐，beta版本。它不同於有限功能的原型，alpha版本，後者同樣是我所宣導的實踐。]

11.3 對於大多數專案，第一個開發的系統並不合用。它可能太慢、太大，而且難以使用，或者三者兼而有之。

11.4 系統的丟棄和重新設計可以一步完成，也可以一塊塊地實現。這是個必須完成的步驟。

11.5 將開發的第一個系統——丟棄原型——發佈給用戶，可以獲得時間，但是它的代價高昂——對於用戶，使用極度痛苦；對於重新開發的人員，分散了精力；對於產品，影響了聲譽，即使最好的再設計也難以挽回名聲。

11.6 因此，爲捨棄而計畫，無論如何，你一定要這樣做。

11.7 “開發人員交付的是用戶滿意程度，而不僅僅是實際的產品。”（Cosgrove）

11.8 用戶的實際需要和用戶感覺會隨著程式的構建、測試和使用而變化。

11.9 軟體產品易於掌握的特性和不可見性，導致了它的構建人員（特別容易）面臨著永恆的需求變更。

11.10 目標上（和開發策略上）的一些正常變化無可避免，事先爲它們做準備總比假設它們不會出現要好得多。

11.11 爲變更計畫軟體產品的技術，特別是細緻的模組介面文檔——非常地廣爲人知，但並沒有相同規模的實踐。盡可能地使用表驅動技術同樣是有所幫助的。[現在記憶體的成本和規模使這項技術越來越出眾。]

11.12 高階語言的使用、編譯時操作、通過引用的聲明整合和自文檔技術能減少變更引起的錯誤。

11.13 採用定義良好的數位化版本將變更量子（階段）化。[當今的標準實踐。]

爲變更計畫組織架構

11.14 程式師不願意爲設計書寫文檔的原因，不僅僅是由於惰性。更多的是源于設計人員的躊躇——要爲自己嘗試性的設計決策進行辯解。（Cosgrove）

11.15 爲變更組建團隊比爲變更進行設計更加困難。

11.16 只要管理人員和技術人才的天賦允許，老闆必須對他們的能力培養給予極大的關注，使管理人員和技術人才具有互換性；特別是希望能在技術和管理角色之間自由地分配人手的時候。

11.17 具有兩條晉升線的高效組織機構，存在著一些社會性的障礙，人們必須警惕和積極地同它做持續的鬥爭。

11.18 很容易爲不同的晉升線建立相互一致的薪水級別，但要同等威信的建立需要一些強烈的心理措施：相同的辦公室、一樣的支援和技術調動的優先補償。

11.19 組建外科手術隊伍式的軟體發展團隊是對上述問題所有方面的徹底衝擊。對於靈活組織架構問題，這的確是一個長期行之有效的解決方案。

前進兩步，後退一步——程式維護

11.20 程式維護基本上不同於硬體的維護；它主要由各種變更組成，如修復設計缺陷、新增功能、或者是使用環境或者配置變換引起的調整。

11.21 對於一個廣泛使用的程式，其維護總成本通常是開發成本的40%或更多。

11.22 維護成本受用戶數目的嚴重影響。用戶越多，所發現的錯誤也越多。

11.23 Campbe11指出了一個顯示產品生命期中每月bug數的有趣曲線，它先是下降，然後攀升。

11.24 缺陷修復總會以（20－50）%的機率引入新的bug。

11.25 在每次修復之後，必須重新運行先前所有的測試用例，從而確保系統不會以更隱蔽的方式被破壞。

11.26 能消除、至少是能指明副作用的程式設計方法，對維護成本有很大的影響。

11.27 同樣，設計實現的人員越少、介面越少，產生的錯誤也就越少。

前進一步，後退一步——系統熵隨時間增加

11.28 Lehman和Be1ady發現模組數量隨大型作業系統（0S/360）版本號的增加呈線性增長，但是受到影響的模組以版本號指數的級別增長。

11.29 所有修改都傾向於破壞系統的架構，增加了系統的混亂程度。即使是最熟練的軟體維護工作，也只是放緩了系統退化到不可修復混亂的進程，從中必須要重新進行設計。[許多程式升級的真正需要，如性能等，尤其會衝擊它的內部結構邊界。原有邊界引發的不足常常在日後才會出現。]

# 第12章 幹將莫邪

12.1 專案經理應該制訂一套策略，以及爲通用工具的開發分配資源，與此同時，他還必須意識到專業工具的需求。

12.2 開發作業系統的隊伍需要自己的目的機器，進行調試開發工作。相對於最快的速度而言，它更需要最大限度的記憶體，還需要安排一名系統程式師，以保證機器上的標準軟體是即時更新和即時可用的。

12.3 同時還需要配備調試機器或者軟體，以便在調試過程中，所有類型的程式參數可以被自動計數和測量。

12.4 目的機器的使用需求量是一種特殊曲線：剛開始使用率非常低，突然出現爆發性的增長，接著趨於平緩。

12.5 同天文工作者一樣，系統調試總是大部分在夜間完成。

12.6 拋開理論不談，一次分配給某個小組連續的目標時間塊被證明是最好的安排方法，比不同小組的穿插使用更爲有效。

12.7 儘管技術不斷變化，這種採用時間塊來安排匱乏電腦資源的方式仍得以延續20年[在1975年]，是因爲它的生產率最高。[在1995年依然如此]

12.8 如果目的機器是新產品，則需要一個目的機器的邏輯仿真裝置。這樣，可以更快地得到輔助調試平臺。即使在真正機器出現之後，仿真裝置仍可提供可靠的調試平臺。

12.9 主程序庫應該被劃分成（1）一系列獨立的私有開發庫；（2）正處在系統測試下的系統集成子庫；（3）發佈版本。正式的分離和進度提供了控制。

12.10 在編制程式的專案中，節省最大工作量的工具可能是文本編輯系統。

12.11 系統文檔中的巨大容量帶來了新的不理解問題[例如，看看Unix]，但是它比大多數未能詳細描述編程系統特性的短小文章更加可取。

12.12 自頂向下、徹底地開發一個性能仿真裝置。盡可能早地開始這項工作，仔細地聽取“它們表達的意見”。

高階語言

12.13 只有懶散和惰性會妨礙高階語言和互動式編程的廣泛應用。[如今它們已經在全世界使用。]

12.14 高階語言不僅僅提升了生產率，而且還改進了調試：bug更少，以及更容易尋找。

12.15 傳統的反對意見——功能、目標代碼的尺寸、目標代碼的速度，隨著語言和編譯器技術的進步已不再成爲問題。

12.16 現在可供合理選擇的語言是PL/I。[不再正確。]

互動式編程

12.17 某些應用上，批次處理系統決不會被互動式系統所替代。[依然成立。]

12.18 調試是系統編程中很慢和較困難的部分，而漫長的調試周轉時間是調試的禍根。

12.19 有限的資料表明了系統軟體發展中，互動式編程的生產率至少是原來的兩倍。

# 第13章 整體部分

13.1 第4、5、6章所意味的煞費苦心、詳盡體系結構工作不但使產品更加易於使用，而且使開發更容易進行以及bug更不容易產生。

13.2 V.A.Vyssotsky提出，“許許多多的失敗完全源於那些產品未精確定義的地方。”

13.3 在編寫任何代碼之前，規格說明必須提交給測試小組，以詳細地檢查說明的完整性和明確性。開發人員自己不會完成這項工作。（Vyssotsky）

13.4 “十年內[1965～1975]，Wirth的自頂向下進行設計[逐步細化]將會是最重要的新型形式化軟體發展方法。”

13.5 Wirth主張在每個步驟中，盡可能使用級別較高的表達方法。

13.6 好的自頂向下設計從四個方面避免了bug。

13.7 有時必須回退，推翻頂層設計，重新開始。

13.8 結構化編程中，程式的控制結構僅由支配代碼塊（相對於任意的跳轉）的給定集合所組成。這種方法出色地避免了bug，是一種正確的思考方式。

13.9 Gold結果顯示了，在互動式調試過程中，第一次交互取得的工作進展是後續交互的三倍。這實際上獲益於在調試開始之前仔細地調試計畫。[我認爲在1995年依然如此。]

13.10 我發現對良好終端系統的正確使用，往往要求每兩小時的終端會話對應於兩小時的桌面工作：1小時會話後的清理和文檔工作；1小時爲下一次計畫變更和測試。

13.11 系統調試（相對於單元測試）花費的時間會比預料的更長。

13.12 系統調試的困難程度證明了需要一種完備系統化和可計畫的方法。

13.13 系統調試僅僅應該在所有部件能夠運作之後開始。（這既不同於爲了查出介面bug所採取 “合在一起嘗試”的方法；也不同於在所有構件單元的bug已知，但未修復的情況下，即開始系統調試的做法。）[對於多個團隊尤其如此。]

13.14 開發大量的輔助調試平臺（scaffolding 腳手架）和測試代碼是很值得的，代碼量甚至可能會有測試物件的一半。

13.15 必須有人對變更進行控制和文檔化，團隊成員應使用開發庫的各種受控拷貝來工作。

13.16 系統測試期間，一次只添加一個構件。

13.17 Lehman和Belady出示了證據，變更的階段（量子）要麼很大，間隔很寬；要麼小和頻繁。後者很容易變得不穩定。[Microsoft的一個團隊使用了非常小的階段（量子）。結果是每天晚上需要重新編譯生成增長中的系統。]

# 第14章 禍起蕭牆

14.1 “項目是怎樣延遲了整整一年的時間？□一次一天。”

14.2 一天一天的進度落後比起重大災難，更難以識別、更不容易防範和更加難以彌補。

14.3 根據一個嚴格的進度表來控制專案的第一個步驟是制訂進度表，進度表由里程碑和日期組成。

14.4 里程碑必須是具體的、特定的、可度量的事件，能進行清晰能定義。

14.5 如果里程碑定義得非常明確，以致于無法自欺欺人時，程式師很少會就里程碑的進展弄虛作假。

14.6 對於大型開發專案中的估計行爲，政府的承包商所做的研究顯示：每兩周進行仔細修訂的活動時間估計，隨著開始時間的臨近不會有太大的變化；期間內對時間長短的過高估計，會隨著活動的進行持續下降；過低估計直到計畫的結束日期之前大約三周左右，才有所變化。

14.7 慢性進度偏離是士氣殺手。[Microsoft的Jim McCarthy說：“如果你錯過了一個最終期限（deadline），確保制訂下一條deadline。[2]”]

14.8 進取對於傑出的軟體發展團隊，同優秀的棒球隊伍一樣，是不可缺少的必要品德。

14.9 不存在關鍵路徑進度的替代品，使人們能夠辨別計畫偏移的情況。

14.10 PERT的準備工作是PERT圖使用中最有價值的部分。它包括了整個網狀結構的展開、任務之間依賴關係的識別、各個任務鏈的估計。這些都要求在項目早期進行非常專業的計畫。

14.11 第一份PERT圖總是很恐怖的，不過人們總是不斷進行努力，運用才智制訂下一份PERT圖。

14.12 PERT圖爲前面那個洩氣的藉口，“其他的部分反正會落後”，提供了答案。

14.13 每個老闆同時需要採取行動的異常資訊以及用來進行分析和早期預警的狀態資料。

14.14 狀態的獲取是困難的，因爲下屬經理有充分的理由不提供資訊共用。

14.15 老闆的不良反應肯定會對資訊的完全公開造成壓制；相反，仔細區分狀態報告、毫無驚慌地接收報告、決不越俎代庖，將能鼓勵誠實的彙報。

14.16 必須有評審的機制，從而所有成員可以通過它瞭解真正的狀態。出於這個目的，里程碑的計畫和完成文檔是關鍵。

14.17 Vyssotsky：我發現在里程碑報告中很容易記錄“計畫（老闆的日期）”和“估計（最基層經理的日期）”的日期。項目經理必須停止對這些日期的懷疑。”

14.18 對於大型項目，一個對里程碑報告進行維護的計畫和控制（Plan and Control）小組是非常可貴的。

# 第15章 另外一面

15.1 對於軟體編程產品來說，程式向用戶所呈現的面貌與提供給機器識別的內容同樣重要。

15.2 即使對於完全開發給自己使用的程式，描述性文字也是必須的，因爲它們會被用戶—作者所遺忘。

15.3 培訓和管理人員基本上沒有能向編程人員成功地灌輸對待文檔的積極態度——文檔能在整個生命週期對克服懶惰和進度的壓力起促進激勵作用。

15.4 這樣的失敗並不都是因爲缺乏熱情或者說服力，而是沒能正確地展示如何有效和經濟地編制文檔。

15.5 大多數文檔只提供了很少的總結性內容。必須放慢腳步，穩妥地進行。

15.6 由於關鍵的用戶文檔包含了跟軟體相關的基本決策，所以它的絕大部分需要在程式編制之前書寫，它包括了9項內容（參見相應章節）。

15.7 每一份發佈的程式拷貝應該包括一些測試用例，其中一部分用於校驗輸入資料，一部分用於邊界輸入資料，另一部分用於無效的輸入資料。

15.8 對於必須修改程式的人而言，他們所需要程式內部結構文檔，同樣要求一份清晰明瞭的概述，它包括了5項內容（參見相應章節）。

15.9 流程圖是被吹捧得最過分的一種程式文檔。詳細逐一記錄的流程圖是一件令人生厭的事情，而且高階語言的出現使它顯得陳舊過時。（流程圖是圖形化的高階語言。）

15.10 如果這樣，很少有程式需要一頁紙以上的流程圖。[在這一點上，MILSPEC軍用標準實在錯得很厲害。]

15.11 即使的確需要一張程式結構圖，也並不需要遵照ANSI的流程圖標準。

15.12 爲了使文檔易於維護，將它們合併至根源程式是至關重要的，而不是作爲獨立文檔進行保存。

15.13 最小化文檔負擔的3個關鍵思路：

借助那些必須存在的語句，如名稱和聲明等，來附加盡可能多的 文檔 資訊。

使用空格和格式來表現從屬和嵌套關係，提高程式的可讀性。

以段落注釋，特別是模組標題的形式，向程式中插入必要的記敍性文字。

15.14 程式修改人員所使用的文檔中，除了描述事情如何以外，還應闡述它爲什麼那樣。對於加深理解，目的是非常關鍵的，但即使是高階語言的語法，也不能表達目的。

15.15 線上系統的高階語言（應該使用的工具）中，自文檔化技術發現了它的絕佳應用和強大功能。

## 原著結束語

E.1 軟體系統可能是人類創造中最錯綜複雜的事物（從不同類型組成部分數量的角度出發）。

E.2 軟體工程的焦油坑在將來很長一段時間內會繼續地使人們舉步維艱，無法自拔。

# 20年後的人月神話（The Mythical Man-Month *after* 20 Years）

只能根據過去判斷將來。

- 派翠克·亨利

然而永遠無法根據過去規劃將來。

- 艾德蒙·伯克

*I know no way of judging the future but by the past.*

*- PATRICK HENRY*

*You can never plan the future by the past.*

*- EDMUND BURKE*

# 爲什麼會出現二十周年紀念版本？

飛機劃破夜空，嗡嗡地飛向紐約的拉瓜迪亞機場。所有的景色都隱藏在雲層和黑暗之中。我正在看一篇平淡無奇的文檔，不過並沒有感到厭煩。緊挨著我的一位陌生人正在閱讀《人月神話》，我在旁邊一直等待著，看他是否會通過文字或者手勢做出反映。最後，當我們向艙門移動時，我無法再等下去了：

“這本書如何？你有什麼評論嗎？”

“噢！這裏面的東西我早就知道。”

此刻，我決定不介紹自己。

爲什麼《人月神話》得以持續？爲什麼看上去它仍然和現在的軟體實踐相關？爲什麼它還擁有軟體工程領域以外的讀者群，律師、醫生、社會學家、心理學家，和軟體人員一樣，不斷地對這本書提出評論意見，引用它，並和我保持通信？20年前的一本關於30年前軟體發展經驗的書，如何能夠依然和現實情況相關？更不用說有所幫助了。

常聽到的一個解釋是軟體發展學科沒有正確地發展，人們經常通過比較電腦軟體發展生產率和硬體製造生產率來支援這個觀點，後者在20年內至少翻了1000倍。正像第16章所解釋的，反常的並不是軟體發展得太慢，而是電腦硬體技術以一種與人類歷史不相配的方式爆發出來。大體上這源於電腦製造從裝配工業向流水線工業、從勞動密集型向資金密集型的逐漸過渡。與生產製造相比，硬體和軟體發展保持著固有的勞動密集型特性。

第二個經常提及的解釋——《人月神話》僅僅是順便提及了軟體，而主要針對團隊中的成員如何創建事物。這種說法的確有些道理，1975年版本的前言中提到，軟體專案管理並不像大多數程式師起初所認爲的那樣，而更加類似於其他類型的管理。現在，我依然認爲這是正確的。人類歷史是一個舞臺，總是上演著相同的故事。隨著文化的發展，這些故事的劇本變化非常緩慢，而舞臺的佈局卻在隨時改變。正是如此，我們發現二十世紀本身會反映在莎士比亞、荷馬的作品和聖經中。因此，某種程度上，《人月神話》是關於人與團隊的書，所以它的淘汰過程會是緩慢的。

不管出於什麼原因，讀者仍然在購買這本書，並且常給我發一些致謝的評論。現在，我常常被問到：“你現在認爲哪些在當時就是錯誤的？哪些是現在才過時的？哪些是軟體工程領域中新近出現的？”這些獨特的問題是完全平等的，我將盡我最大的能力來回答它們。不過，不以上述順序，而是按照一系列主題來答復。首先，讓我們考慮那些在寫作時就正確，現在依然成立的部分。

# 核心觀點：概念完整性和結構師

概念完整性。一個整潔、優雅的編程產品必須向它的每個用戶提供一個條理分明的概念模型，這個模型描述了應用、實現應用的方法以及用來指明操作和各種參數的用戶介面使用策略，用戶所感受到的產品概念完整性是易用性中最重要的因素。（當然還有其他因素。Macintosh上所有應用程式介面的統一就是一個重要的例子。此外，有可能建立統一的介面，儘管它可能

很粗糙，就像MS-DOS。）

有很多由一個或者兩個人設計的優秀軟體產品例子。大多數純智力作品，像書籍、音樂等都是採用這種方式創作出來的。不過，很多產業的產品開發過程無法負擔這種獲取概念完整性的直接方法。競爭帶來了壓力，很多現代工藝的最終產品是非常複雜的，它們的設計需要很多人月的工作量。軟體產品十分複雜，在進度上的競爭也異常激烈。

任何規模很大或者非常緊急，並需要很多人力的專案，都會碰到一個特別的困難：必須由很多人來設計，但與此同時，還需要在概念上保持與單個使用人員的一致。如何組織設計隊伍來獲得上述的概念一致性？這是《人月神話》關注的主要問題。其中一點：由於參與人數的不同，大型編程專案的管理與小型專案在性質上都不同。爲了獲得一致性，經過深思熟慮的，有時甚至是英勇的管理活動是完全必要的。

結構師。從第4到第7章，我一直不斷地在表達一個觀點——委派一名產品結構師是最重要的行動。結構師負責產品所有方面的概念完整性，這些是用戶能實際感受到的。結構師開發用於向用戶解釋使用的產品概念模型，概念模型包括所有功能的詳細說明以及調用和控制的方法。結構師是這些模型的所有者，同時也是用戶的代理。在不可避免地對功能、性能、規模、成本和進度進行平衡時，卓有成效地體現用戶的利益。這個角色是全職工作，只有在最小的團隊中，才能和團隊經理的角色合併。結構師就像電影的導演，而經理類似于製片人。

將體系結構和設計實現、物理實現相分離。爲了使結構師的關鍵任務更加可行，有必要將用戶所感知的產品定義——體系結構，與它的實現相分離。體系結構和實現的劃分在各個設計任務中形成了清晰的邊界，邊界兩邊都有大量的工作。

體系結構的遞迴。對於大型系統，即使所有實現方面的內容都被分離出去，一個人也無法完成所有的體系結構工作。所以，有必要由一位元主結構師把系統分解成子系統，系統邊界應該劃分在使子系統間介面最小化和最容易嚴格定義的地方。每個部分擁有自己的結構師，他必須就體系結構向主結構師彙報。顯然，這個過程可以根據需要重複遞迴地進行。

今天，我比以往更加確信。概念完整性是產品品質的核心。擁有一位元結構師是邁向概念完整性的最重要一步。這個原理決不僅限於軟體系統，它適用於所有的複雜事物，如電腦、飛機、防禦系統、全球定位系統等。在軟體工程試驗室進行20次以上的講授之後，我開始堅持每4個學生左右的小組就選擇不同的經理和結構師。在如此小的隊伍中定義截然不同的角色可能是有點極端，但我仍然發現這種方法即使對小型團隊也運作良好，並且促使了設計的成功。

## 開發第二個系統所引起的後果：盲目的功能和頻率猜測

爲大型用戶群設計。個人電腦革命的一個結果是，至少在商業資料處理領域中，客戶應用程式越來越多地被商用套裝軟體所代替。而且，標準套裝軟體以成百上千，甚至是數百萬拷貝的規模出售。源廠商支援性軟體的系統結構師必須不斷地爲大型的不確定用戶群，而不是爲某個公司的單一、可定義的應用進行設計。許許多多的系統結構師現在面臨著這項任務。

但自相矛盾的是，設計通用工具比設計專用工具更加困難，這是因爲必須爲不同用戶的各種需要分配權重。

盲目的功能（Featuritis）。對於如試算表或字處理等通用工具的結構師，一個不斷困擾他們的誘惑是以性能甚至是可用性的代價，過多地向產品添加邊界實用功能。

功能建議的吸引力在初期階段是很明顯的，性能代價在系統測試時才會出現。而隨著功能一點一點地添加，手冊慢慢地增厚，易用性損失以不易察覺的方式蔓延。[1]

對倖存和發展了若干代的大眾軟體產品，這種誘惑特別強烈。數百萬的用戶需要成百上千的功能特色，任何需求本身就是一種"市場需要它"的證明。而常見的情況是，原有的系統結構師得到了嘉獎，正工作在其他崗位或專案上，現在負責體系結構的結構師，在均衡表達用戶的整體利益方面，往往缺乏經驗。一個對Microsoft Word 6.0的近期評價聲稱"Word 6.0對功能特性進行了打包，通過包緩慢地更新□□Word 6.0同樣是大型和慢速的。"有點令人沮喪的是——Word 6.0需要4MB記憶體，豐富的新增功能意味著"甚至Macintosh IIfx都不能勝任Word 6的任務"[2]。

定義用戶群。用戶群越大和越不確定，就越有必要明確地定義用戶群，以獲得概念完整性。設計隊伍中的每個成員對用戶都有一幅假想的圖像，並且每個設計者的圖像都是不同的。結構師的用戶圖像會有意或者無意地影響每個結構決策，因此有必要使設計隊伍共用一幅相同的用戶圖像。這需要把用戶群的屬性記錄下來，包括：

他們是誰

他們需要（need）什麼

他們認爲自己需要（need）什麼

他們想要（want）的是什麼

頻率。對於軟體產品，任何用戶群屬性實際上都是一種概率分佈，每個屬性具有若干可能的值，每個值有自己的發生頻率。結構師如何成功地得到這些發生頻率？對並未清晰定義的物件進行調查是一種不確定和成本高昂的做法[3]。經過很多年，我現在確信，爲了得到完整、明確和共有的用戶群描述，結構師應該猜測（guess），或者假設（postulate）完整的一系列屬性和頻率值。

這種不是很可靠的過程有很多好處。首先，仔細猜測頻率的過程會使結構師非常細緻地考慮物件用戶群。其次，把它們寫下來一般會引發討論，這能起到解釋的作用，以及澄清不同設計人員對用戶圖像認識上的差異。另外，明確地列舉頻率能幫助大家認識到哪些決策依賴於哪些用戶群屬性。這種非正式的敏感性分析也是頗有價值的。當某些非常重要的決策需要取決於一些特殊的猜測時，很值得爲那些數值花費精力來取得更好的估計。（Jeff Conklin開發的gIBIS提供了一種工具，能精確和正式地跟蹤設計決策和文檔化每個決策的原因[4]。我還沒有機會使用它，但是我認爲它應該非常有幫助。）

總結：爲用戶群的屬性明確地記載各種猜測。清晰和錯誤都比模糊不清好得多。

開發第二系統所引起的後果（second-system effect） 情況怎樣？一位敏銳的學生說，人月神話推薦了一劑對付災難的處方：計畫發佈任何新系統的第二個版本（第11章），第二系統在第5章中被認爲是最危險的系統。

這實際上是語言引起的的差異，現實情況並不是如此。第5章中提到的 第二個 系統是第二個實際系統，它是引入了很多新增功能和修飾的後續系統。第11章中的 第二個 系統指開發第一個實際系統所進行的第二次嘗試。它在所有的進度、人員和範圍約束下開發，這些約束刻畫了專案的特徵，形成了開發準則的一部分。

## 圖形（**WIMP**）介面的成功

在過去20年內，軟體發展領域中，令人印象最深刻的進步是視窗（Windows）、圖示（Icons）、功能表（Menus）、指標選取（Pointing）介面的成功——或者簡稱爲WIMP。這些在今天是如此的熟悉，以致於不需要任何解釋。這個概念首先在1968年西部聯合電腦大會（Western Joint Computer Converence）上，由斯坦福研究機構（Stanford Research Institute）的Doug Englebart公開提出[5]。接著，這種思想被Xerox Palo Alto Research Center所採納，用在了由Bob Taylor和他的團隊所開發的Alto個人工作站中。Steve Jobs在Apple Lisa型電腦中應用了該理念，不過Apple Lisa運行速度太慢，以致於無法承載這個令人激動的易用性概念。後來在1985年，Jobs在取得商業成功的Apple Macintosh機器上體現了這些想法。接下來，它們被IBM PC及其兼容機的Microsoft Windows所採用。我自己的例子則是Mac版本[6]。

通過類比獲得的概念完整性。WIMP是一個充分體現了概念完整性的用戶介面例子，完整性的獲得是通過採用大家非常熟悉的概念模型——對桌面的比喻，以及一致、細緻的擴展，後者充分發揮了電腦的圖形化實現能力。例如，視窗採用覆蓋，而不是排列的方式，這直接來自類比。儘管這種方法成本很高，但卻是正確的決定。電腦圖形介質提供了對視窗尺寸的調整，這是一種保持一致概念的延伸，給用戶提供了新的處理能力，桌面上的檔是無法輕易地調整大小和改變形狀的；拖放功能則直接出自模仿，使用指標來選擇圖示是對人用手拾起東西的直接模擬；圖示和嵌套檔夾源于桌面的文檔，回收站也是如此；剪切、複製和粘貼則完全反映了我們使用文檔的一些習慣；我們甚至可以通過向回收站拖拽磁片的圖示來彈出磁片——象徵手法是如此的貼切，擴展是如此的連貫一致，以致於新用戶常常會被它所體現出的理念打動。

哪些地方使WIMP介面遠遠超越了桌面的比喻？主要是在兩個方面：功能表和單手操作。在真正的桌面上工作時，人們實際上是操作文檔，而不是叫某人來完成這些動作。當要求他人進行某個活動時，常常是新產生，而不是選擇一個口頭或者書面祈使句：“請將這個歸檔。”“請找出前面一致的地方。”“請把這個交給Mary去處理。”

無論是手寫還是口頭的命令，現有處理能力還無法對自由產生的命令形式進行可靠的翻譯和解釋。所以，介面設計人員從用戶對文檔的直接動作中去除了上面提到的兩個步驟。他們非常聰明地從桌面文檔操作中選取了一些常用命令，形成了類似於公文的“便條”，用戶只需從一些語義標準的強制命令功能表中進行選擇。這個概念接著被延伸到水準功能表和垂直的下拉子功能表中。

命令表達和雙游標問題。命令是祈使句，它們通常都有一個動詞和直接賓語。對於任何動作，必須指定一個動詞和一個名詞。對事物選取的直接模仿要求：使用螢幕上不同的兩個游標，同時指定兩件事物。每個游標分別由左右手中的滑鼠來控制。畢竟，在實際的桌面上，我們通常使用兩隻手來操作。（不過，一隻手常常是將東西固定在某處，這一點在電腦桌面是默認情況。）而且，我們當然具備雙手操作的能力，我們習慣上使用雙手來打字、駕駛、烹飪。但是，提供一個滑鼠已經是個人電腦製造商向前邁進的一大步，沒有任何商業系統可以容納由雙手分別控制的兩隻滑鼠同時進行的動作[7]。

介面設計人員接受了現實，爲一隻滑鼠設計。設計人員採用的句法習慣是首先指出（選擇）名詞的，接著指出一個動詞功能表項。這確實犧牲了很多易用性。當我看到用戶、或者用戶錄影、或者游標移動的電腦跟蹤情況。我立刻對一個游標必須完成兩件事而感到驚訝：選擇視窗上桌面部分的一個物件，再選擇功能表部分的一個動詞，尋找或者重新尋找桌面上的一個物件。接著，拉下功能表（常常是同一個）選擇一個動詞。游標來來往往、周而復始地從資料區移到功能表區，每一次都丟棄了一些有用的位置資訊“上次在這個空間的什麼地方”——總而言之，是一個低效的過程。

一個卓越的解決方案。即使軟體和器材可以很容易實現兩個同時活動的游標，也仍然存在一些空間佈局上的困難。WIMP象徵手法中的桌面實際上包括了一個打字機，它必須在實際桌面的物理空間中容納一個物理鍵盤。鍵盤加上兩個滑鼠墊會佔據大量雙手所及的空間。不過，鍵盤問題實際上是一個機會——爲什麼不一只手在鍵盤上指定動詞，另一隻手使用滑鼠來指定名詞，從而使高效的雙手操作成爲可能？這時，游標停留在資料區，爲後續點擊拾取提供了充分的空間活動能力。真正的高效，真正強大的用戶功能。

用戶功能和易用性。不過，這個解決方案捨棄了一些易用性——功能表提供了任何特定狀態下的一些可選的有效動詞。例如，我們可以購買某個商品，將它帶回家，緊記購買的目的，遵照功能表上不同的動詞略爲試驗一下，就可以開始使用，並不需要去查看手冊。

軟體結構師所面臨的最困難問題是如何確切地平衡用戶功能和易用性。是爲初學者或偶爾使用的用戶設計簡單操作，還是爲專業用戶設計強大的功能？理想的答案是通過概念一致的方式把兩者都提供給用戶——這正是WIMP介面所達到的目標。每個頻繁使用的功能表動詞（命令）都有一個快捷鍵，因此可以作爲組合通過左手一次性地輸入。例如，在Mac機器上，命令鍵（⇧[jypan2]）正好在Z和X鍵的下方，因此使用最頻繁的操作被編碼成⇧z、⇧x、⇧c、⇧v、⇧s。

從新手向熟練用戶的逐漸過渡。雙重指定命令動詞的系統不但滿足了新手快速學習的需要，而且它在不同的使用模式之間提供了平滑的過渡。被稱爲快捷鍵的字元編碼，顯示在功能表上的動詞旁邊，因此拿不准的用戶可以啓動下拉功能表，檢查對應的快捷鍵，而不是直接在菜單上選取。每個新手從他最常使用的命令中學習快捷鍵。由於⇧z可以撤銷任何單一操作，所以他可以嘗試任何感到不確定的快捷鍵。另外，他可以檢查功能表，以確定命令是否有效。新手會大量地使用功能表，而熟練用戶幾乎不使用，中間用戶僅偶爾需要訪問功能表，因爲每個人都瞭解組成自己大多數操作的少數快捷鍵。我們大多數的開發設計人員對這樣的介面非常熟悉，對其優雅而強大的功能感到非常欣慰。

強制體系結構的實施，作爲設備的直接整合。Mac介面在另一個方面很值得注意。沒有任何強迫，它的設計人員在所有的應用程式中使用標準介面，包括了大量的第三方程式。從而，用戶在介面上獲得的概念一致性不僅僅局限在機器所配備的軟體，而且遍及所有的應用程式。

Mac設計人員把介面固化到唯讀記憶體中，從而開發者使用這些介面比開發自己的特殊介面更容易和快速。這些獲取一致性的措施得到了足夠廣泛的應用，以致于可以形成實際的標準。Apple的管理投入和大量說服工作協助了這些措施。產品雜誌中很多獨立評論家，認識到了跨應用概念完整性的巨大價值，通過批評不遵從產品的反面例子，對上述方法進行了補充。

這是第6章中所推薦技術的一個非常傑出的例子，該技術通過鼓勵其他人直接將某人的代碼合併到自己的產品中來獲得一致性，而不是試圖根據某人的技術說明開發自己的軟體。

WIMP的命運：過時被淘汰。儘管WIMP有很多優點，我仍期望WIMP介面在下一代技術中成爲歷史。如同我們支配我們機器一樣，指標選取仍將是表達名詞的方式，語音則無疑成爲表達動詞的方法。Mac上的Voice Navigator和PC上的Dragon已經提供了這種能力。

## 沒有構建捨棄原型——瀑布模型是錯誤的！

一幅讓人無法忘懷的圖畫，倒塌的塔科馬大橋，開啓了第11章。文中強烈地建議："爲捨棄而計畫。無論如何，你一定要這樣做。"現在我覺得這是錯誤的，並不是因爲它太過極端，而是因爲它太過簡單。

在《未雨綢繆》一章體現的概念中，最大的錯誤是它隱含地假設了使用傳統的順序或者瀑布開發模型。該模型源自于類似甘特圖佈局的階段化流程，常常繪製成圖19.1的形狀。Winton Royce在1970年的一篇經典論文中，改進了順序模型，他提出：

存在一些從一個階段到前一個階段的回饋

將回饋限定在直接相鄰的先前階段，從而容納它引起的成本增加和進度延遲

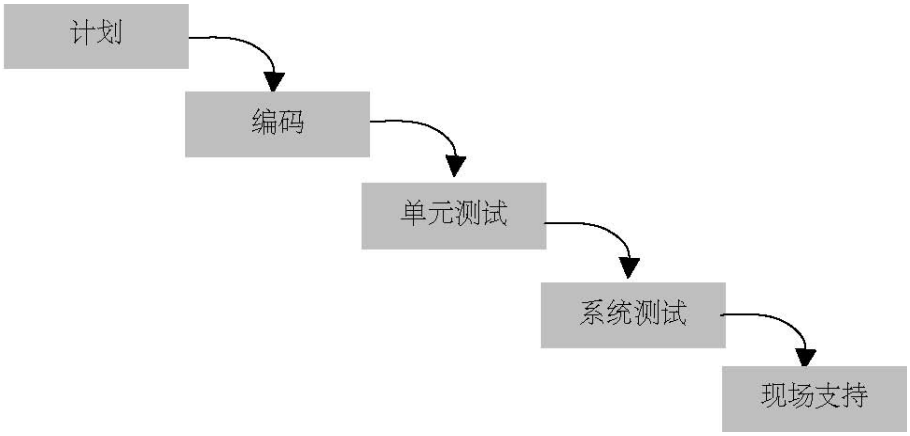



圖19.1：軟體發展的瀑布模型

他給開發者提出的建議　構建兩次　比《人月神話》的好[8]。受到瀑布模型不良影響並不只是第11章，而是從第2章的進度計畫規則開始，貫穿了整本書。第2章中的經驗法則分配了1/3

的時間用於計畫，1/6用於編碼，1/4用於單元測試以及1/4用於系統測試。

瀑布模型的基本謬誤是它假設項目只經歷一次過程，而且體系結構出色並易於使用，設計是合理可靠的，隨著測試的進行，編碼實現是可以修改和調整的。換句話說，瀑布模型假設所有錯誤發生在編碼實現階段，因此它們的修復可以很順暢地穿插在單元和系統測試中。

實際上，《未雨綢繆》並沒有迎面痛擊這個錯誤。它不是對錯誤的診斷，而是補救措施。現在，我建議應該一塊塊地丟棄和重新設計系統，而不是一次性地完成替換。就目前的情況而論，這沒有問題，但它並沒有觸及問題的根本。瀑布模型把系統測試，以及潛在地把用戶測試放在構件過程的末尾。因此，只有在投入了全部開發投資之後，才能發現無法接受的性能問題、笨拙功能以及察覺用戶的錯誤或不當企圖。不錯，Alpha測試對規格說明的詳細檢查是爲了儘早地發現這些缺陷，但是對於實際參與的用戶卻沒有對應的措施。

瀑布模型的第二個謬誤是它假設整個系統一次性地被構建，在所有的設計、大部分編碼、部分單元測試完成之後，才爲閉環的系統測試合併各個部分。

瀑布模型，這個大多數人在1975年考慮的軟體專案開發方法，不幸地被奉爲軍用標準DOD-STD-2167，作爲所有國防部軍用軟體的規範。所以，在大多數有見地的從業者認識到瀑布模型的不完備並放棄之後，它仍然得以倖存。幸運的是，DoD也已經慢慢察覺到這一點[9]。

計畫

編碼

單元測試

系統測試

現場支持

必須存在逆向移動。就像本章開始圖片中精力充沛的大馬哈魚一樣，在開發過程“下游”的經驗和想法必須躍行而上，有時會超過一個階段，來影響“上游”的活動。

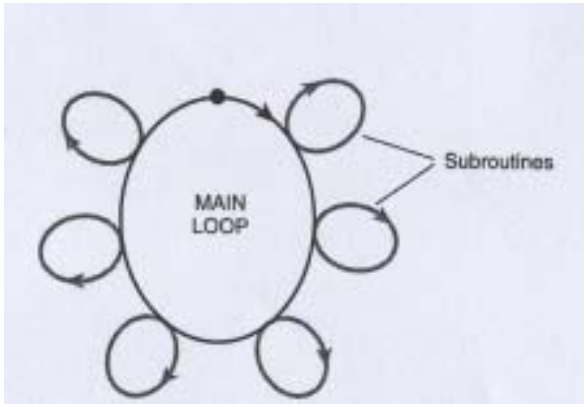
例如，設計實現會發覺有些體系結構的功能定義會削弱性能，從而體系結構必須重新調整。編碼實現會發現一些功能會使空間劇增，超過要求，因此必須更改體系結構定義和設計實現。

所以，在把任何東西實現成代碼之前，可能要往復迭代兩個或更多的體系結構－設計－實現迴圈。

## 增量開發模型更佳——漸進地精化

構建閉環的框架系統

從事即時系統開發的Harlan Mills，早期曾提倡我們首先應該構建即時系統的基本輪詢回路，爲每個功能都提供子函數調用（占位元符），但僅僅是空的子函數（圖19.2）。對它進行編譯、測試，使它可以不斷運行。它不直接完成任何事情，但至少是正常運行的[10]。




注：

MAIN LOOP－主迴圈

Subroutines－子函數

圖19.2

接著，我們添加（可能是基本的）輸入模組和輸出模組。瞧，一個可運行的系統出現了，儘管只是一個框架。然後，一個功能接一個功能，我們逐漸開發和增加相應模組。在每個階段，我們都擁有一個可運行的系統。如果我們非常勤勉，那麼每個階段都會有一個經過調試和測試的系統。（隨著系統的增長，使用所有先前的測試用例對每個新模組進行的回歸測試也採用這種方式進行。）

在每個功能基本可以運行之後，我們一個接一個地精化或者重寫每個模組——增量地開發（growing）整個系統。不過，我們有時的確需要修改原有的驅動回路，或者甚至是回路的模組介面。

因爲我們在所有時刻都擁有一個可運行的系統，所以

我們可以很早開始用戶測試，以及

我們可以採用按預算開發的策略，來徹底保證不會出現進度或者預算的超支（以允許的功能犧牲作爲代價）。

我曾在北卡羅來納大學教授軟體工程實驗課程22年，有時與David Parnas一起。在這門課程中，通常4名學生的團隊會在一個學期內開發某個真正的即時軟體應用系統。大約是一半的時候，我轉而教授增量開發的課程。我常常會被螢幕上第一幅圖案、第一個可運行的系統對團隊士氣產生的鼓舞效果而感到震驚。

Parnas產品族

在這整個20年的時間裏，David Parnas曾是軟體工程思潮的帶頭人。每個人對他的資訊隱藏概念都很熟悉，但對他另一個非常重要的概念　　將軟體作爲一系列相關的產品族來設計[11]相對瞭解較少。Parnas力勸設計人員對產品的後期擴展和後續版本進行預測，定義它們在功能或者平臺上的差異，從而搭建一棵相關產品的家族樹（圖19.3）。

設計類似一棵樹的技巧是將那些變化可能性較小的設計決策放置在樹的根部。

這樣的設計使得模組的重用最大化。更重要的是，可以延伸相同的策略，使它不但可以包括發佈產品，而且還包括以增量開發策略創建的後續中間版本。這樣，產品可以通過它的中間階段，以最低限度的回溯代價增長。

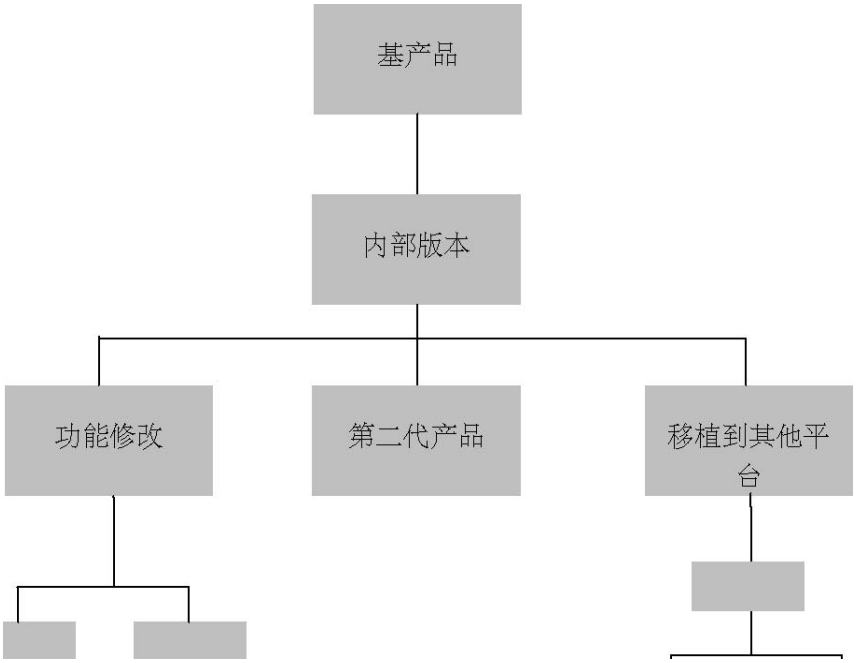



圖19.3

Microsoft的“每晚重建”方法

Jams McCarthy向我描述了他的隊伍和微軟其他團隊所使用的產品開發流程，這實際上是一種邏輯上的增量式開發。他說，

在我們第一次產品發佈之後，我們會繼續發佈後續版本，向已有的可運行系統添加更多的功能。爲什麼最初的構建過程要不一樣呢？因此，從我們第一個里程碑開始[第一次發佈有三個里程碑]，我們每晚重建開發中的系統[以及運行測試用

例]。該構建週期成了項目的“心跳”。每天，一個和多個程式師-測試員隊伍提交若干具有新功能的模組。在每次重建之後，我們會獲得一個可運行的系統。如果重建失敗，我們將停下整個過程，直到找到問題所在並進行修復。在任何時間，團隊中的每個人都瞭解專案的狀態。

這是非常困難的。你必須投入大量的資源，而且它是一個規範化、可跟蹤、開誠佈公的流程。它向團隊提供了自身的可信度，而可信度決定了你的士氣和情緒狀態。

其他組織的軟體發展人員對這個過程感到驚訝，甚至震驚。其中一個人說：“我們可以實現每週一次的重建，但是如果每晚一次的話，我想不大可能，工作量太大了。”這可能是對的。例如，Be11北方研究所就是每週重建1千2百萬行的系統。

功能修改

移植到其他平臺

第二代產品

內部版本

基產品

增量式開發和快速原型

增量開發過程能使真正的用戶較早地參與測試，那麼它與快速原型之間的區別是什麼呢？我認爲它們既是互相關聯，又是相互獨立的。各自可以不依賴對方而存在。

Hare1將原型精彩地定義成：

僅僅反映了概念模型準備過程中所做的設計決策[的一個程式版本]，它並未反映受實現考慮所驅使的設計決策[12]。

構建一個完全不屬於發佈產品的原型是完全可能的。例如，可以開發一個介面

原型，但是並不包含任何的實際功能，而僅僅是一個看上去履行了各個步驟的有限狀態機。甚至可以通過類比系統回應的嚮導技術來原型化和測試介面。這種原型化對獲取早期的用戶回饋非常有用，但是它和產品發佈前的測試區別很大。

類似的，實現人員可能會著手開發產品的某一塊，並完整地實現該部分的有限功能集合，從而可以儘早發現性能上的潛在問題。那麼，“從第一個里程碑開始構建”的微軟流程和快速原型之間的差別是什麼？功能。第一個里程碑產品可能不包含足夠的功能使任何人對它產生興趣，而可發佈產品和定義中的一樣，在完整性上——配備了一系列實用的功能集，在品質上——它可以健壯地運行。

## 關於資訊隱藏，**Parnas**是正確的，我是錯誤的

在第7章中，關於每個團隊成員應該在多大程度上被允許和鼓勵相互瞭解設計和代碼的問題，我對比了兩種方法。在作業系統OS/360專案中，我們決定所有的程式師應該瞭解所有的材料——每個項目成員都擁有一份大約10,000頁的專案工作手冊拷貝。Harlan Mills頗有說服力地指出“編程是個開放性的公共過程”。把所有工作都暴露在每個人的凝視之下，能夠幫助品質控制，這既源于其他人優秀工作的壓力，也由於同伴能直接發現缺陷和bug。

這個觀點和David Parnas的觀點形成了鮮明的對比。David Parnas認爲代碼模組應該採用定義良好的介面來封裝，這些模組的內部結構應該是程式師的私有財產，外部是不可見的。編程人員被遮罩而不是暴露在他人模組內部結構面前。這種情況下，工作效率最高[13]。

我在第7章中並不認同Parnas的概念是“災難的處方”。但是，Parnas是正確的，我是錯誤的。現在，我確信資訊隱藏——現在常常內建於面向物件的編程中——是唯一提高軟體設計水準的途徑。

實際上，任何技術的使用都可能演變成災難。Mill的技術是通過瞭解介面另一側的情況，使編程人員能理解他們所工作介面的詳細語義。這些介面的隱藏會導致系統的bug。Parnas的技術在面對變更時是很健壯的，更加適合爲變更設計的理念。

第16章指出了下列情況：

> 過去在軟體生產率上取得的進展大多數來自消除非內在的困難，如笨拙的編程語言、漫長的批次處理周轉時間等。
>
> 像這些比較容易解決的困難已經不多了。
>
> 徹底的進展將來自對根本困難的處理　　打造和組裝複雜概念性結構要素。

最明顯的實現這些的方法是，認爲程式由比獨立的高階語言語句、函數、模組或類等更大的概念結構要素組成。如果能對設計和開發進行限制，我們僅僅需要從已建成的集合中參數化

這些結構要素，並把它們組裝在一起，那麼我們就能大幅度提高概念的級別，消除很多無謂的工作和大量語句級別的錯誤可能性。

Parnas的模組資訊隱藏定義是研究專案中的第一步，它是面向物件編程的鼻祖。Parnas把模組定義成擁有自身資料模型和自身操作集的軟體實體。它的資料僅僅能通過它自己的操作來訪問。第二步是若干思想家的貢獻：把Parnas模組提升到抽象資料類型，從中可以派生出很多物件。抽象資料類型提供了一種思考和指明模組介面的統一方式，以及容易保證實施的類型規範化訪問方法。

第三步，面向物件編程引入了一個強有力的概念　　繼承，即類（資料）默認獲得類繼承層次中祖先的屬性[14]。我們希望從面向物件編程中得到的最大收穫實際上來自第一步，模組隱藏，以及預先建成的、爲了重用而設計和測試的模組或者類庫。很多人忽視了這樣一個事實，即上述模組不僅僅是程式，某種意義上是我們在第1章中曾討論過的編程產品。許多人希望大規模重用，但不付出構建產品級品質（通用、健壯、經過測試和文檔化的）模組所需要的初始代價　　這種期望是徒勞的。面向物件編程和重用在第16和17章中有所討論。

## 人月到底有多少神話色彩？**Boehm**的模型和資料

很多年來，人們對軟體生產率和影響它的因素進行了大量的量化研究，特別是在專案人員配備和進度之間的平衡方面。

最充分的一項研究是Barry Boehm對63個專案的調查，其中大多數是航空專案和25個TRW公司的專案。他的《軟體工程經濟學》（Software Engineering Economics）不但包括了很多結果，而且還有一系列逐步推廣的成本模型。儘管一般商業軟體的成本模型和根據政府標準開發的航空軟體成本模型中的係數肯定不同，不過他的模型使用了大量的資料來支撐。我想從現在起，這本書將作爲一代經典。

他的結果充分地吻合了《人月神話》的結論，即人力（人）和時間（月）之間的平衡遠不是線性關係，使用人月作爲生產率的衡量標準實際是一個神話。特別的，他發現：[15]

第一次發佈的成本最優進度時間，$T = 2.5(MM)^{1/3}$。即，月單位的最優時間是估計工作量（人月）的立方根，估計工作量則由規模估計和模型中的其他因數導出。最優人員配備曲線是由推導得出的。

當計畫進度比最優進度長時，成本曲線會緩慢攀升。時間越充裕，花的時間也越長。

當計畫進度比最優進度短時，成本曲線急劇升高。

無論安排多少人手，幾乎沒有任何項目能夠在少於3/4的最優時間內獲得成功！當高級經理向專案經理要求不可能的進度擔保時，這段結論可以充分地作爲專案經理的理論依據。

Brooks準則有多準確？曾有很多細緻的研究來評估Brooks法則的正確性，簡言之，向進度落後的軟體專案中添加人手只會使進度更加落後。最棒的研究發表在Abdel-Hamid和Madnick在1991年出版的一本頗有價值的書《軟體專案動力學：一條完整的路》[16]（Software Project Dynamics：An Integrated Approach）中。書中提出了專案動態特性的量化模型。關於Brooks準則的章節提供了更詳細的分析，指出了在各種假設下的情況，即何時添加多少人員將會產生什麼樣的結果。爲了進行研究，作者擴展了他們自己一個中型規模專案的模型，假設新成員有學習曲線和需要額外的溝通和培訓工作。他們得出結論　向進度落後的專案中添加人手總會增加項目的成本，但並不一定會使項目更加落後。　特別的，由於新成員總會立刻帶來需要數周來彌補的負面效應，所以在項目早期添加額外的人力比在後期加入更加安全一些。

Stutzke爲了進行相似的研究，開發了一個更簡單的模型，得出了類似的結果[17]。他對引入新成員進行了詳細的過程和成本分析，其中包括把他們的指導人員調離原有的項目任務。他在一個真正的專案上測試了他的模型，在專案中期的一些偏移之後，他成功地添加了一倍人手，並且保證了原先的進度。相對于增加更多程式師，他還試驗了的其他方法，特別是加班工作。在他的很多條實踐建議中，最有價值的部分是如何添加新成員，進行培訓，用工具來支持等等。特別值得注意的是，他建議開發專案後期增加的開發人員，必須作爲團隊成員，願意在過程中努力投入和工作，而不是企圖改變或者改進過程本身！

Stutzke認爲更大型的專案中，增加的溝通負擔是次要作用，沒有對它建模。至於Abdel-Hamid和Madnick是否或者如何考慮這個問題，則不是很清楚。上面提到的兩個模型都沒有考慮開發人員必須重新安排的事實，而在實際情況中，我發現這常常是一個非常重要的步驟。

這些細緻的研究使　異常簡化　的Brooks準則更加實用。作爲平衡，我還是堅持這個簡單的陳述，作爲真理的最佳近似，以及一項經驗法則　　警告經理們避免對進度落後的專案採取的盲目、本能的修補措施。

## 人就是一切（或者說，幾乎是一切）

很多讀者發現很有趣的是，《人月神話》的大部分文章在講述軟體工程管理方面的事情，較少涉及到技術問題。這種傾向部分因爲我在IBM 360作業系統（現在是MVS/370）項目中角色的性質。更基本的是，這來自一個信念，即對於專案的成功而言，專案人員的素質、人員的組織管理是比使用的工具或採用的技術方法更重要的因素。

隨後的研究支持了上述觀點。Boehm的COCOMO模型發現團隊品質目前是項目成功最大的決定因素，實際上是下一個次重要因素的4倍。現在，軟體工程的大多數學術研究集中在工具上。我很欣賞和期盼強大的工具，同樣我也非常鼓勵對軟體管理動態特徵——對人的關注、激勵、培養——的持續研究。

人件。近年來，軟體工程領域的一個重大貢獻是DeMarco和Lister在1987年出版的資料，《人件：高生產率的項目和團隊》（Peopleware：Productive Projects and Teams）。它所表達的觀點是“我們行業的主要問題實質上更側重於社會學（sociological）而不是科學技術（technological）。”它充滿了很多精華，如“管理人員的職責不是要人們去工作，而是是創造

工作的可能。”它涉及了如空間、佈置、團隊的餐飲等主題。DeMarco和Lister從他們的Coding War Games專案中提供的資料，顯示了相同組織中開發人員的表現之間，和工作空間和生產率以及缺陷水準之間令人吃驚的關聯。

頂尖人員的空間更加安靜、更加私人、保護得更好以免受打斷，還有很多□□這對你真的很要緊嗎□□是否安靜、空間和免受打攪能夠幫助你的人員更好地完成工作，或者[換個角度]能幫助你吸引和留住更好的人員？

我衷心地向我的讀者推薦這本書。

項目轉移。DeMarco和Lister對團隊融合給予了相當大的關注，團隊融合是一個無形的，但是非常關鍵的特性。很多地點分散的公司，項目從一個實驗室轉移到另一個。從中，我認爲團隊融合正是管理上被忽視的因素。

我的觀察和經驗大約局限在六、七個專案轉移中，其中沒有一個是成功的。任務可以成功地轉移，但是對於項目的轉移，即使擁有良好的文檔、先進的設計以及保留部分原有人員，新隊伍實際上依然是重新開始。我認爲正是由於破壞了原有團隊的整體性，導致了產品雛形的夭折，專案重新開始。

## 放棄權力的力量

如果人們認同我在文中多處提到的觀點——創造力來自於個人，而不是組織架構或者開發過程，那麼專案管理面對的中心問題是如何設計架構和流程，來提高而不是壓制主動性和創造力。幸運的是，這個問題並不是軟體組織所特有，一些傑出的思想家正努力地致力於這項工作。E.F.Schumacher在他的經典《小就是美：人們關心的經濟學》（Small is Beautiful：Economics as if People Mattered）中，提出了最大化員工創造力和工作樂趣的理論。他的第一個原理是引自Pope Pius XI教皇通諭Quadragesimo Anno中的“附屬職能行使原理”：

向大型組織指派小型或者附屬機構能夠完成的職責是不公平的，同時也是正常次序的不幸和對它的干擾。對於每項社會活動，就其本質而言，應該配備對社會個體成員的幫助，而不是去破壞和吸收它們□□那些當權者應該確信遵守“附屬職能行使”原理，能在各種各樣的組織中維持更加完美的次序，越強和越有效的社會權威將會是國家更加融洽和繁榮的條件[19]。

Schumacher繼續解釋到：

附屬職能行使原理告訴我們——如果較低級別組織的自由和責任得以保留，中心權威實際上是得到了加強；其結果是，從整體而言，組織機構實際上將“更加融洽和繁榮”。

如何才能獲得上述的架構？□□大型組織機構由很多准自治單元構成，我們稱之爲准公司。它們中的每一個都擁有大量的自由，來爲創造性和企業家職能提供最大的可能機會□□。每個准公司同時具備盈虧帳目和資產負債表[20]。

軟體工程中最激動人心的進展是將上述組織理念付諸實踐的早期階段。首先，微型電腦革

命創造了新型的軟體工業，出現了成百上千的新興公司。所有這些小規模的公司熱情、自由和富有創造性。隨著很多小型公司被大公司收購，這個產業正在發生著變化，而那些大公司是否理解和保留小規模的創造性尚待分曉。

更不尋常的是，一些大型公司的高層管理已經開始著手將一些權力下放到軟體專案團隊，使它們在結構和責任上接近於Schumacher的准公司。其運作的結果是令人欣喜和吃驚的。

微軟的Jim AcCarthy向我描述了他在解放團隊上的經驗：

每個隊伍（30至40人）擁有自己的任務、進度，甚至如何定義、構建、發佈的過程。團隊由4或5個專家組成，包括開發、測試和書寫文檔等。由團隊而不是老闆對爭論進行仲裁。我無法形容授權和由團隊自行負責專案的成功與否的重要性。

Earl Wheeler，IBM軟體業務的退休主管，告訴我他著手下放IBM部門長期集權管理權力的經驗：

[近年來]關鍵的措施是將權力向下委派。這就像是魔術！改進的品質、提高的生產率、高漲的士氣。我們的小型團隊，沒有中心控制。團隊是流程的所有者，並且必須擁有一個流程。他們有不同的流程。他們是進度計畫的所有者，因此感受到市場的壓力。這種壓力導致他們使用和利用自己的工具。

和團隊成員個人的談話，顯示了他們對被委派的權力和自由的贊同，同時也反映出真正的下放顯得多少有些保守。不過，授權是朝著正確方向邁出的一大步，它產生了像Pius XI所預言的好處：通過權力委派，中心的權威實際上是得到了加強；從整體而言，組織機構實際上更加融洽和繁榮。

## 最令人驚訝的新事物是什麼？數百萬的電腦

每位元我曾交談過的電腦帶頭人都承認，對微型電腦革命和它引發的塑膠薄膜包裝軟體產業感到驚訝。毫無疑問，這是繼《人月神話》後二十年中最重要的改變。它對軟體工程意味著很多。

微型電腦革命改變了每個人使用電腦的方式。Schumacher在20年前，陳述了面對的挑戰：

我們真正想從科學家和技術專家那裏得到什麼？我會回答：我們需要這樣的方法和設備：

價格足夠低廉，使幾乎所有人都能夠使用

適合小型的應用，並且

滿足人們對創造的渴望[21]。

這些正是微型電腦革命帶給電腦產業和它的用戶（現在已覆蓋到普通公眾）的傑出特性。一般人現在不但可以買得起自己的電腦，而且還可以負擔20年前只有國王的薪水才能買得起的軟體。Schumacher的目標值得仔細思考，每個目標達到的程度值得品評，尤其是最後一個。在一個一個的領域中，普通人同專家一樣可以應用新的自我表達方法。

其他領域中進步的部分原因和軟體創造相近　　消除了次要的困難。例如，文書處理方式曾經是很僵化的，合併更改內容需要重新打字，成本和時間都比較高昂。一份300頁的手稿，常常每3到6個月就需要重新輸入一遍，這中間，人們往往還不斷地產生新文稿。另外，邏輯流程和語句韻律的修訂很難進行。而現在，文書處理已經非常方便和流暢了[22]。

電腦同樣給其他一些領域帶來了相似的處理能力，繪畫、制訂計畫、機械製圖、音樂創作、攝影、攝像、幻燈、多媒體甚至是試算表等。在這些領域中，手工操作都需要重新拷貝大量的未改變的部分，以便在上下文中區別修改情況。現在我們能享受這樣的好處，即立刻對結果進行修訂和評估，無須失去思維的連貫性，就象分時帶給軟體發展的好處一樣。

同樣，新的、靈活的輔助工具增進了創造力。以寫作爲例，我們現在擁有拼寫檢查、語法檢查、風格顧問、目錄生成系統以及對最終排版預覽的能力。

最重要的是，當一件創造性工作剛剛成形時，工作介質的靈活性使得對多種徹底不同的可選方案的探索變得容易。這實際上是一個量變引起質變的例子，即時間變化引起工作方式上的巨大變化。

繪圖工具使建築設計人員爲每小時的創造性投資展現了更多的選擇。電腦與合成器的互聯，加上自動生成或者演奏樂譜的軟體，使得人們更容易捕獲創作的靈感。數位式相機，和Adobe Photoshop一起，使原先在暗室中需要數日的工作在幾分鐘內就可以完成。試算表可以對大量 what if 的各種情況進行實驗、比較。

最後，個人電腦的普遍存在導致了全新創造性活動介質的出現。Vannevar Bush在1945年提出的超文本，僅能在電腦上實現[23]。多媒體表現形式和體驗更是如此　　在電腦和大量價格低廉的軟體出現以前，實現起來有太多的困難。至於現在並不便宜或普遍的虛擬環境系統，將成爲另一個創造性活動的媒介。

微型電腦革命改變了每個人開發軟體的方式。70年代的軟體過程本身被微處理器革命和它所帶來的科學技術進步所改變。很多軟體發展過程的次要困難被消除。快速的個人電腦現在是軟體發展者的常規工具，從而周轉時間的概念幾乎成爲了歷史。如今的個人電腦不僅僅比1960年的超級電腦要快，而且它比1985年的Unix工作站還要快。所有這些意味著即使在最差的電腦上，編譯也是快速的，而且大記憶體消除了基於磁片鏈結所需要的等待時間。另外，符號表和目標代碼可以在記憶體中保存，從而高級別的調試無需重新編譯。

在過去的20年裏，我們幾乎全部採用了分時作爲構建軟體的方法學。在1975年，分時才剛剛作爲最常用的技術替換了批次處理計算。網路使軟體構建人員不僅可以訪問共用檔，還可以訪問強大的編譯、鏈結和測試引擎。今天，個人工作站提供了計算引擎，網路主要提供了對檔的共用訪問，這些檔作爲團隊開發的工作產品。客戶－伺服器系統則使測試用例檢入、開發和

應用的共用訪問更加簡單。

同樣，用戶介面也取得了類似的進步。和一般的文本一樣，WIMP介面對程式文本提供了更加方便迅捷的編輯方式。24行、72列的螢幕已經被整頁甚至是雙頁的螢幕所取代，因此程式師可以看到所作更改的更多上下文。

## 全新的軟體產業　　塑膠薄膜包裝的成品軟體

在傳統軟體產業的旁邊，爆發了另一個全新的產業。產品以成千上萬，甚至是數百萬的規模銷售。整套內容豐富的套裝軟體可以以少於開發成本的價格獲得。這兩個產業在很多方面都不同，它們共同存在著。

傳統軟體產業。在1975年，軟體產業擁有若干可識別的、但多少有些差異的組成部分，如今他們依然存在：

電腦提供商：提供作業系統、編譯器和一些實用程式

應用程式用戶：如公共事業、銀行、保險、政府機構等，他們爲自己使用的軟體發展應用套裝程式。

定制程式開發者：爲用戶承包開發私用套裝軟體，需求、標準和行銷步驟都是與眾不同的。

商業包開發者：那個時候是爲專業市場開發大型應用，如統計分析和CAD系統等。

Tom DeMarco注意到了傳統軟體產業的分裂，特別是應用程式用戶。

我沒有料到的是：整個行業被分解成各個特殊的領域。你完成某事的方式更像是專業領域的職責，而不僅僅是通用系統分析方法、通用語言、通用測試技術的使用。Ada是最後一個通用語言，並且它已經慢慢變成了一門專業語言。

在日常的商業應用領域中，第4代語言作出了巨大的貢獻。Boehm說，　大多數成功的第4代語言是以選項和參數方式系統化某個應用領域的結果。　這些第4代語言最普遍的情況是帶有查詢語言、資料庫－通訊套裝軟體的應用生成程式。

作業系統世界已經統一了。在1975年，存在著很多作業系統：每個硬體提供商每條產品線最少有一種作業系統，很多提供商甚至有兩個。如今是多麼的不同啊！開放式系統是基本原則。目前，人們主要在5大作業系統環境上行銷自己的應用套裝程式（按照時間順序）：

IBM MVS和VM環境

DEC VMS環境

Unix環境，某個版本

IBM PC環境，DOS、OS-2或者Windows

Apple Macintosh環境

塑膠薄膜包裝的成品軟體產業。對於這個產業的開發者，面對的是與傳統產業完全不同的經濟學：軟體成本是開發成本與數量的比值，包裝和市場成本非常高。在傳統內部的應用開發產業，進度和功能細節是可以協商的，開發成本則可能不行；而在競爭激烈的開發市場面前，進度和功能支配了開發成本。

正如人們所預期的，完全不同的經濟學引發了非常不同的編程文化。傳統產業傾向於被大型公司以已指定的管理風格和企業文化所支配。另一方面，始於數百家創業公司的成品軟體產業，行事自由，更加關注結果，而不是流程。在這種趨勢下，那些天才的個人程式師更容易獲得認可，這隱含了　卓越的設計來自於傑出的設計人員　的觀點。創業文化能夠對那些傑出人員，根據他們的貢獻進行獎勵。而在傳統軟體產業中，公司的社會化因素和薪資管理計畫總會使上述做法難以實施。因此，很多新一代的明星人物被吸引到薄膜包裝的軟體產業，這一點並不奇怪。

## 買來開發　　使用塑膠包裝的成品套裝軟體作爲構件

徹底提高軟體健壯性和生產率的唯一途徑，是提升一個級別，使用模組或者物件組合來進行程式的開發。一個特別有希望的趨勢是使用大眾市場的套裝軟體作爲平臺，在上面開發更豐富和更專業化的產品。如使用塑膠包裝的資料庫和通訊套裝軟體來開發貨運跟蹤系統，或者學生的資訊系統等。而電腦雜誌上的徵文欄目提供了許許多多的Hypercard Stack、Excel範本、Minicard的特殊Pascal函數以及AutoCad的AutoLisp函數。

元編程。Hypercard Stack、Excel範本、Minicard函數的開發有時被稱爲元編程（metaprograming），爲部分套裝軟體用戶進行功能定制的過程。元編程並不是新概念，僅僅是重新被提出和重新命名。在60年代早期，很多電腦提供商和資訊管理系統（MIS）廠商都擁有小型專家小組，他們使用組合語言的巨集來裝備應用編程語言。Eastman Kodak的MIS開發車間使用一種用IBM 7080巨集組譯定義的自有應用語言。類似的，IBM的OS/360 佇列遠端通訊訪問方法中（Queued Telecommunications Access Method），在遇到機器級別指令之前，人們可以讀到若干頁組合語言的通訊程式。現在元編程人員提供要素的規模是巨集的若干倍。這種二級市場的開發是非常鼓舞人心的——當我們在期待C++類開發的高效市場時，可重用元程式的市場正在悄無聲息地崛起。

它處理的確實是根本問題。因爲包開發現象並沒有影響到一般的MIS編程人員，所以對於軟體工程領域並不是很明顯。不過，它將快速地發展，因爲它針對的正是概念結構要素打造的根本問題。成品套裝軟體提供了大型的功能模組和精心定制的介面，它內部的概念結構根本無

需再設計。功能強大的軟體產品，如Excel或者4th Dimension實際上是大型的模組，而且它們作爲廣爲人知、文檔化、測試過的模組，可以用來搭建用戶化系統。下一級應用程式的開發者可以獲得豐富的功能、更短的開發時間、經過測試的元件、良好的文檔和徹底降低的成本。

當然，存在的困難是成品軟體是作爲獨立實體來設計，元程式師無法改變它的功能和介面。另外，更嚴肅地說，對於成品軟體的開發者而言，把產品變成更大型系統中的模組似乎沒有什麼吸引力。我認爲這種感覺是錯誤的，在爲元程式師開發提供套裝軟體方面，有一個未開拓的市場。

那麼需要什麼呢？我們可以識別出四個層次的軟體用成品戶：

直接使用用戶。他們以簡便直接的方式來操作，對設計者提供的功能和介面感到滿意。

元程式師。在單個應用程式的基礎上，使用已提供的介面來開發範本或者函數，主要爲最終用戶節省工作量。

外部功能作者，向應用程式中添加自行編制的功能。這些功能本質上是新應用語言原語，調用通用語言編寫的獨立模組。這往往需要命令中斷、回調或者重載函數技術，向原介面添加新功能。

元程式師，使用一個和多個特殊的應用程式，作爲更大型系統的構件。他們是需求並沒有得到滿足的用戶群。同時，這也是能在構建新應用程式方面獲得較大收穫的用法。

對於成品軟體，最後一種類型的用戶還需要額外的文檔化介面，即元編程介面（metaprogramming interface，MPI）。這在很多方面提出了要求。首先，元程式需要在整個軟體集的控制之下，而每個軟體通常假設是受自己的控制。軟體集必須控制用戶介面，而應用程式一般認爲這是自己的職責。軟體整體必須能夠調用任何應用程式的功能，就好像是用戶使用命令行傳遞參數那樣。它還應該像螢幕一樣接受應用程式的輸出，只不過螢幕是顯示一系列字串，而它需要將輸出解析成適當資料類型的邏輯單元實體。某些應用程式，如FoxPro，提供了一些接收命令的後門介面（wormhole），不過它返回的資訊是不夠充分和未被解析的。這些介面是對通用解決方案需要的一個特殊補充。

擁有能控制應用程式集合之間交互的腳本語言是非常強有力的。Unix首先使用管道和標準的ASCII字串格式提供了這種功能。今天，AppleScript是一個非常優秀的例子。

## 軟體工程的狀態和未來

我曾問過北卡羅來納州大學化學系的系主任Jim Ferrell關於化學工程的歷史以及和化學的區別的問題，於是他作了一個1小時的出色即興演說，從很多產品（從鋼鐵到麵包，到香水）的不同生產過程開始。他講述了Arthur D. Little博士如何在1918年在麻省理工學院建立了第一個化學工程系，來發現、發展和講授所有過程的共有技術基礎。首先是經驗法則，接著是經驗圖表，後來是設計特殊零件的公式，再後來是單個導管中熱傳導、品質轉移和動量轉移的模型。

如同Ferrell故事所展現的，在幾乎50年後，我仍被化學工程和軟體工程之間的很多相似之處所震動。Parans對我寫的關於軟體工程（software engineering）的文章提出了批評。他對比了電氣工程和軟體領域，覺得把我們所做的稱爲“工程”十分冒昧。他可能是正確的，這個領域可能永遠不會發展成像電氣工程那樣的工程化領域，擁有精確的數學基礎。畢竟，軟體工程就像化學工程一樣，與如何擴展到工業級別處理過程的非線性問題有關。而且，和工業工程類似，它總是被人類行爲的複雜性所困擾。

不過，化學工程的發展過程讓我覺得“27歲的”軟體工程並不是沒有希望的，而僅僅是不夠成熟的，就好像1945年的化學工程。畢竟，在二次世界大戰之後，化學工程師才真正提出閉環互聯的連續流系統。

今天，軟體工程的一些特殊問題正如第1章中所提出的：

如何把一系列程式設計和構建成系統

如何把程式或者系統設計成健壯的、經過測試的、文檔化的、可支援的產品

如何維持對大量的複雜性的控制

軟體工程的焦油坑在將來很長一段時間內會繼續地使人們舉步維艱，無法自拔。軟體系統可能是人類創造中最錯綜複雜的事物，只能期待人們在力所能及的或者剛剛超越力所能及的範圍內進行探索和嘗試。這個複雜的行業需要：進行持續的發展；學習使用更大的要素來開發；新工具的最佳使用；經論證的管理方法的最佳應用；良好判斷的自由發揮；以及能夠使我們認識到自己不足和容易犯錯的　　上帝所賜予的謙卑。

# 結束語：令人嚮往、激動人心和充滿樂趣的五十年（*Epilogue Fifty Years of Wonder, Excitement, and Joy*）

我依然記得那種嚮往和開心的感覺－當我在1944年8月7日讀到哈佛大學Mark I型電腦研製成功的報導時－那時候我才13歲。Mark I是電子機械學上的奇跡，哈佛大學的Aiken是它的構架設計師，而IBM的工程師Clair Lake，Benjamin Durfee和Francis Hamilton是它的實施設計師。同樣令人嚮往的是讀到Vannevar Bush 1945年4月發表在亞特蘭大月刊上的論文“That We May Think”（我們的期望？）的時候，在這篇論文中，他建議將大量的知識組織成超文本的網路方式，並爲用戶提供機器從已有的鏈結以及指明其他的相關鏈結。

我對電腦的熱情在1952年進一步高漲，因爲得到了IBM在紐約恩迪科特的一份暑期工，正是那次，我有了在IBM 604上編程的實際經驗，也瞭解了如何編制IBM 701（它的第一個存儲程式

電腦）程式的正式指令；從哈佛大學Aiken和Iverson名下畢業終於讓我的職業夢想變成了現實，並且，就這樣沉迷了一輩子。感謝上帝，讓我成爲了爲數不多的那些開開心心做著自己喜歡的工作的人之一。

我實在無法想像還有哪種生活會比熱愛電腦更加激動人心，自從從真空管發展到積體電路以來，電腦技術已經飛速發展。我用來工作的第一台電腦，是從哈佛剛剛出爐的IBM7030 Stretch超級電腦，Stretch在1961到1964年間都是世界上運算速度最快的電腦，一共賣出了9台。而我現在用的電腦，Macintosh Powerbook，不但快，還有大容量記憶體和大容量硬碟，而且便宜了1000倍（如果按定值美元來算，便宜了5000倍）。我們依次看到了電腦革命，電子電腦革命，小型電腦革命，微型電腦革命，這些技術上的革命每一次都帶來了電腦數量上的劇增。

在電腦技術進步的同時，電腦相關學科知識也在飛速發展。當我在五十年代剛從學校畢業的時候，我能看完當時所有的期刊和會議報告，掌握所有的潮流動向。而我現在只能對層出不窮的學科分支遺憾地說"再見"，對我所關注的東西也越來越難以全部掌握。興趣太多，令人興奮的學習、研究和思考的機會也太多——多麼不可思議的矛盾啊！這個神奇的時代遠遠沒有結束，它依然在飛速發展。更多的樂趣，盡在將來。

# 注解和參考文獻（*Notes and References*）

## 第**1**章

1. Ershov認爲編程是一種樂趣和苦惱共存的活動。A.P. Ershov, "Aesthetics and the human factor in programming," CACM, 15,7(July,1972), pp. 501-505.

## 第**2**章

1. Bell電話實驗室的V.A. Vyssotsky估計一個大專案必須維持每年30%的人員投入. 這導致了巨大的壓力, 甚至是限制了在第7章中, 所討論的根本、非正式結構和溝通的演化.

麻省理工學院的F.J. Corbató指出, 一個長期的專案必須預見到每年有20%的人員更替, 這必須進行技術上進行培訓以及集成到原有的結構.

2. International Computers Limited的C. Portman提出: "當所有的一切看上去可以工作, 已經被集成時, 你至少還有4個多月的工作需要完成." 在Wolverton, R. W., "The cost of developing lage-scale software," *IEEE Trans. on Computers*, C-23, 6(June, 1974) pp.615-636提出了若干其他的進度劃分.

3. 圖2.5至2. 8出自Jerry Ogdin, 他引用了這章的早期版本, 必須改進相應的描述. Ogdin, J. L., "The Mongolian hordes versus superprogrammer," *Infosystems* (Dec., 1972), pp.20-23。

## 第3章

1. Sackman, H., W. J. Erikson, and E. E. Grant, “Exploratory experimental studies comparing online and offline programming performance,” CAM, 11, 1(Jan., 1968), pp. 3-11.
2. Mills, H., “Chief programmer teams, principles, and procedures,” IBM Federal Systems Division Report FSC 715108, Gaitherburg, Md., 1971
3. Baker F. T., “Chief programmer team management of production programming,” IBM Sys. J. 11, 1 (1972).

## 第4章

1. Eschapasse, M., Reims Cathedral, Caisse Nationale des Monuments Historiques, Paris, 1967.
2. Brooks, F. P., “Architectural philosophy,” in W. Buchholz(ed.), Planning A Computer System. New York: McGraw-Hill, 1962.
3. Blaauw, G.. A., “Hardware requirements for the fourth generation,” in F. Gruenberger (ed.), Fourth Generation Computers. Englewood Cliffs, N. J.: Prentice-Hall, 1970.
4. Brooks, F. P., and K. E. Iverson, Automatic Data Processing, System/360 Edition. New York: Wiley, 1969, Chapter 5.
5. Glegg G. L., The Design of Design. Cambridge: Cambridge Univ. Press, 1969, 提出“乍一看, 用任何規則或者原理來約束創造性思維的想法是一種阻礙, 而不是幫助, 但實際情況中完全不是這樣. 規範的思維實際上是促進而不是阻礙了靈感的產生.”
6. Conway, R. W., “The PL/C Compiler,” Proceedings of a Conf. on Definition and Implementation of Universal Programming Languages. Stuttgard, 1970.
7. 關於編程技術必要性的討論, 參見C. H. Reynolds, “What's wrong with computer programming management?” in G. F. Weinwurm (ed.). *On the Management of Computer Programming.* Philadelphia: Auerbach, 197.1 pp. 35-42.

## 第5章

1. Strachey C., “Review of Planning a Computer System”, Comp. J., 5, 2 (July, 1962), pp. 152-153.
2. 這僅僅適用於控制程式. OS/360專案中的一些編譯器開發團隊正構建他們的第三個或第四個系統, 他們卓越的產品展示了這一點.
3. Shell, D. L., “The Share 709 systems: a cooperative effort”; Greenwald, I. D., and M. Kane, “The Share 709 system: programming and modification”; *Boehm E. M., and T. B. Steel, Jr.* “The Share 709 system: machine implementation of symbolic programming”; all in JACM, 6, 2(April, 1959), pp. 123-140.

## 第6章

1. Neustadt R. E., *Presidential Power.* New York: Wiley, 1960, Chapter 2.
2. Backus J. W., “The syntax and semantics of the proposed international algebraic language.” *Proc. Intl. Conf. Inf. Proc. UNESCO,* Paris, 1959, published by R. Oldenbourg, Munich, and Butterworth, London. Besides this, a whole collection of papers on the subject is contained in T. B. Steel, Jr. (ed.). *Formal Language Description Languages for Computer Programming.* Amsterdam: North Holland, 1966.
3. Lucas, P., K. Walk, “On the formal description of PL/I” *Annual Review in Automatic*

*Programming Language*. New York: Wiley, 1962. Chapter 2, p. 2.
4. Iverson K. E. *A Programming Language.* New York: Wiley, 1962. Chapter 2.
5. Falkoff A. D., K. E. Iverson, E. H. Sussenguth, "A formal description of System/360," *IBM Systems Journal.* 3, 3,(1964), pp. 198-261.
6. Bell C. G., A. Newell, *Computer Structures.* New York: McGraw-Hill, 1970, pp. 120- 136, 517-541.
7. Bell, C. G.., private communication.

# 第**7**章

1. Parnas D. L., "Information distribution aspects of design methodology," Carnegie-Mellon Univ., Dept. of Computer Science Technical Report, February, 1971.
2. Copyright 1939, 1940 Street & Smith Publications, Copyright 1950, 1967 by Robert A. Heinlein. Published by arrangement with Spectrum Literary Agency.

# 第**8**章

1. Sackman ,H., W. J. Erikson, and E. E. Grant, "Exploratory experimentation studies comparing online and offline programming performance," CACM, 11, 1( Jan. 1968), 11, pp. 3-11.
2. Nanus, B., and L. Farr, "Some cost contributors to large-scale programs," AFIPS Proc. SJCC, 25(Spring, 1964), pp. 239-248.
3. Weinwurm, G. F., "Research in the management of computer programming," Report SP-2059, System Development Corp. Santa Monica, 1965.
4. Morin, L. H., "Estimation of resources for computer programming projects," M. S. thesis. Univ. Of North Carolina, Chapel Hill, 1974.
5. Portman, C., private communication.
6. 一份未發表的E. F. Bardain1964研究指出程式師實際的生產時間占27%. (爲D. B. Mayer and A. W. Stalnaker所引用, "Selection and evaluation of computer personnel, " *Proc. 23d ACM Conf.,* 1968, p. 661.)
7. Aron, J. , Private communication.
8. 材料在小組會議中給出, 沒有包括於the AFIPS Proceedings.
9. Wolverton, R. W. "The cost of developing large-scale software," *IEEE Trans. On Computers*. C-23, 6, (June,1974), pp. 615-636. 這篇重要新近發表的文章包含的資料核實了生產率方面的結論, 同時還有許多所討論問題的資料.
10. Corbató, F. J. "Sensitive issues in the design of multi-use systems," 在好萊塢EDP技術中心1968年的公開演講.
11. W. M. Taliaffero同時指出了在Fortran和Cobol編譯器方面的生產率爲2400語句/年. 參見" Modularity. The key to system growth potential," *Software*, 1, 3. (July, 1971), pp. 245-257.
12. E. A. Nelson's System Development Corp. Report TM-3225, *Management Handbook for Estimation of Computer Programming Costs,* 儘管標注有較大的背離, 仍然顯示了高階語言帶來了1至3倍生產率的提高(pp. 66-67).

# 第**9**章

1. Brooks F. P., and K. E .Iverson, Automatic Data Processing, System/360 Edition. New York: Wiley, 1969. Chapter 6.
2. Knuth, D. E., The *Art of Computer Programming.* Vols. 1 - 3. Reading, Mass.:

Addison-Wesley, 1968. ff.

# 第**10**章

1. Conway, M. E., “How do committees invent? ” *Datamation*. 14,4(April. 1968 ), pp. 28-31.

# 第**11**章

1．在Oglethorpe大學1932年5月22號的演講.
2. 描述了Multics在兩個成功系統上所獲得經驗的書籍是F. J. Corbató, J. H. Saltzer, and C. T. Clingen, “Multics-the first seven years,” AFIPS Proc SJCC. 40(1972), pp. 571-583.
3. Cosgrove, J., “Needed: a new planning framework,” *Datamation,* 17, 23(Dec.1971 ), pp. 37-39.
4. 設計變更的問題是很複雜的, 這裏我過於簡化了. 參見J. H. Saltzer “Evolutionary design of complex systems,” in D. Eckman (ed.), *Systems: Research and Design.* New York: Wiley, 1961. 當所有的事被提出和完成, 我依然提倡構建一個被拋棄的實驗性系統.
5. Campbell, E., “Report to the AEC Computer Information Meeting,” December, 1970. 該現象同時有J. L. Ordin 在“Designing reliable software,” *Datamation*. 18, 7(. July. 1972), pp. 71-78中討論. 至於曲線是否會再次下降, 我的具有豐富檢驗的朋友們各執己見.
6. Lehman, M., and L. Belady, “Programming systems dynamics,” given at the ACM SIGOPS Third Symposium on Operating Systems Principles ,October, 1971.
7. Lewis, C. S., *Mere Christianity.* New York: Macmillan, 1960, p. 54.

# 第**12**章

1. 參見J. W. Pomeroy, “A guide to programming tools and techniques,” IBM *Sys. J.*, 11,3(1972), pp. 234-254.
2. Landy B., R. M. Needham, “ Software engineering techniques used in the development of the Cambridge Multiple-Access System” *Software*, 1,2 (April, 1971), pp. 167-173.
3. *Corbato F. J.* , “PL/I as a tool for system programming” *Datamation*, 15, 5(May, 1969), pp. 68-76.
4. Hopkins, M., “Problems of PL/I for system programming” IBM Research Report RC 3489. Yorktown Heights, N. Y., August 5, 1971.
5. Corbato F. J., J. H. Saltzer, and C. T. Clingen, “MULTICS - the first seven years”, *AFIPS Proc SJCC*, 40(1972) pp. 571-582. “出於達到最優性能的原因, 僅有半打使用*PL/L*編程的領域重新用彙編進行了改寫. 許多最初使用機器語言編寫的程式都用*PL/L*重新編寫,譯提高它們的可維護性.”
6. 引用Corbato論文中的參考資料3: *"PL/I is here now and the alternatives are still untested"*. 同時,書寫良好的提出反面意見的文章, 參見Henricksen J. O. and R. E. Merwin, “Programming language efficiency in real-time software systems”, *AFIPS Proc SJCC.* 40(1972). pp. 155-161.
7. 並不是所有人都同意. 在一次私下的交流中, Harlan Mills說: “我的經驗開始告訴我, 在產品開發中, 將秘書安排到終端面前. 其思想是使編程成爲在眾多團隊成員監督下, 更加大眾化的實踐, 而不是一項專有的技術..”
8. Yarr J., “Programming Experience for the Number 1 Electronic Switching System,” paper given at the 1969 SJCC.

# 第**13**章


1. Vyssotsky V. A., 在Chapel Hill, N. C1972年舉辦的電腦程式測試方法討論會"Common sense in designing testable software". Vyssitsky的大多數演講收錄在Hetzel, W. C. (ed.), *Program Test Methods.* Englewood Cliffs, N. J.: Prentice-Hall, 1972. pp. 41-47.
2. Wirth, N., "Program development by stepwise refinement," CACM 14, 4(April, 1971) pp. 221-227. 參見Mills, H., "Top-down programming in large systems," in R. Rustin (ed.). *Debugging Techniques in Large Systems. Englewood Cliffs,* N. J.: Prentice-Hall, 1971, pp. 41-55; and Baker F. T., "System quality through structured programming," *AFIPS Proc FJCC.* 41-I(1972), pp. 339-343.
3. Dahl O. J., E. W. Dijkstra, and C. A. R. Hoare, *Structured programming.* London and New York: Academic Press, 1972. 該專欄包括了最完整的討論處理. 參見Dijkstra的書信"GOTO statement considered harmful," CACM., 11,3(March, 1968), pp. 147-148.
4. Bohm C.,and A. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," CACM., 9, 5(May, 1966), pp. 366-371.
5. Codd E. F., E. S. Lowry, E. McDonough, and C. A. Scalzi, "Multiprogramming STRETCH: Feasibility considerations," CACM., 2, 11(Nov., 1959), pp. 13-17.
6. Strachey, C., "Time sharing in large fast computers," *Proc. Int. Conf. on Info. Processing.* UNESCO (June, 1959), pp. 336-341. 參見Codd在p.341上的評論, 他彙報了類似於Strachey論文中所建議工作的進展.
7. Corbato F. J., M. Merwin-Daggett, and R. C. Daley "An experimental time-sharing system," *AFIPS Proc SJCC.*, 2, (1962), pp. 335-344. 重印於S. Rosen, *Programming Systems and Languages.* New York: McGraw-Hill, 1967, pp. 683- 698.
8. Gold, M. M., "A methodology for evaluating time-shared computer system usage," Ph. D. dissertation. Carngie-Mellon University, 1967, p. 100.
9. Gruenberger, F., "Program testing and validating," Datamation., 14,7 (July, 1968), pp. 39-47.
10. *Ralston, A., Introduction to Programming and Computer Science.* New York: McGraw-Hill, 1971. pp. 237-244.
11. Brooks F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition.* New York: Wiley, 1969, pp. 296-299.
12. 一種良好的規格說明開發和系統構建及測試處理方法由F. M. Trapnell提出, "A systematic approach to the development of system programs," *AFIPS Proc SJCC*, 34, (1969), pp. 41-48.
13. 即時系統需要環境模擬器. 例子參見M. G. Ginzberg, "Notes on testing real-time system programs," *IBM Sys. J.*, 4, 1(1965), pp. 58-72.
14. Lehman, M., and L. Belady, "Programming systems dynamics," 提出於ACM SIGOPS Third Symposium on Operating Systems Priciples, October, 1971.


# 第**14**章


1. See C. H. Reynolds, "What's wrong with computer programming management?" in G. F. Weinwurm (ed.), *On the Management of Computer Programming.* Philadelphia: Auerbach, 1971, pp. 35-42.
2. King, W. R., and T. A.Wilson, "Subjective time estimates in critical path planning-a preliminary analysis," Mgt. Sci., 13, 5(Jan., 1967), pp. 307-320, and sequel, W.R. King, D. M. Witterrongel, K. D. Hezel, "On the analysis of critical path time estimating behavior," Mgt. Sci., 14,1(Sept., 1967), pp. 79-84.
3. 更詳細的討論, 參見Brooks, F. P., and K. E. Iverson, *Automatic Data Processing, System/360 Edition.* New York: Wiley, 1969. P. 428-230.
4. Private communication.

# 第**15**章

1. Goldsteine H. H., and J. von. Neumann, 在爲U.S. Army Ordinance Department, 1947; 所提交的報告中“Planning and coding problems for en electronic computing instrument,” Part II, Vol. 1.; 並在 J. von. Neumann, “Collected Works,”中重新發表 A. H. Taub (ed.). Vol. v., New York: Macmillan. P. 80-151.
2. Private communication, 1957. 該觀點在Iverson, K. E., “The use of APL in Teaching,” Yorktown, N.Y.: IBM Corp., 1969中提出.
3. PL/I的另外一個例子由B. Walter and M. Bohl, 在“From better to best - tips for good programming,” Software *Age,* 3, 11(Nov., 1969), pp. 46-50中提出. 相同的技術可以使用在Algol中, 甚至還有一個Fortran格式的程式”STYLE”來達到上述效果. 參見D. D. McCracken, and G. M. Weinberg, “How to write a readable FORTRAN program,” *Datamation*, 18, 10(Oct., 1972), pp. 73-77.

# 第**16**章

1. 提名爲"No Silver Bullet"的論文源自於Information Processing 1986, 由H. –J. Kugler (1986) 所編輯的the Proceedings of the IFIP Tenth World Computing Conference, pp. 1069-76. 在IFIP 和Elsevier Science B. V., Amsterdam, The Netherlands的獲准後重印.
2. Parnas, D. L., “Designing software for ease of extension and contraction,” *IEEE Trans on SE,* **5**, 2 (March, 1979), pp. 128-138.
3. Booch, G., “Object-oriented design,” Software Engineering with Ada. Menlo Park, Calif.: Benjamin/Cummings, 1983.
4. Mostow, J., ed., Special Issue on Artificial Intelligence and Software Engineering, IEEE Trans. on SE, 11, 11 (Nov., 1985).
5. Parnas, D. L., “Software aspects of strategic defense systems,” Communications of the ACM, 28, 2 (Dec., 1985), pp. 1326-1335. Also in *American Scientist*, 73,5 (Sept.-Oct., 1985), pp. 432-440.
6. Balze , R., “A 15-year perspective on automatic programming,” 在Mostow, 引文中.
7. Mostow, 引文.
8. Parnas, 1985, 引文.
9. Raeder, G., “A survey of current graphical programming techniques,” in R. B. Grafton and T. Ichikawa, eds., Special Issue on Visual Programming, *Computer*, 18, 8 (Aug., 1985), pp. 11 -25.
10. 該題目在本書的第15章有所討論.
11. Mills, H., “Top-down programming in large systems,” *Debugging Techniques in Large Systems,* R. Rustin, ed., Englewood Cliffs, N. J.: Prentice-Hall, 1971.
12. Boehm, B. W., “A spiral model of software development and enhancement,” *Computer*, 20, 5 (May, 1985), pp. 43-57.

# 第**17**章

未被引用的材料源自於私下交流.

1. Brooks, F. P., “No silver bullet - essence and accidents of software engineering,” in *Information* Processing *86*, H. J. Kugler ed., Amsterdam: Elsevier Science, (North Holland), 1986, pp. 1069-1076.

2. Brooks, F. P., “No silver bullet - essence and accidents of software engineering,” *Computer*, 20, 4 (Apr., 1987), pp. 10-19.
3. 許多信件和一些回復, 出現在the July, 1987 issue of *Computer*.

非常高興地看到《沒有銀彈》沒有接受任何大獎, Bruce M. Skwiersky’s的評論作爲 Computer Reviews 在1988年選出的最佳評論. E. A. Weiss, “Editorial,” Computer Reviews (June, 1988), pp. 283-284, 均宣佈了上述評論的獲獎情況和重新提出了Skwiersky的觀點. 該評論有一個重大的錯誤: “sixfold”應該爲“$10^6$”.
4. “根據經院哲學中亞里斯多德提出, 次要(accident)是不屬於事物必要或者根本的屬性, 而是作爲其他原因引起的後果. *Webster's New International Dictionary of the English Language,* 2d ed., Springfield, Mass.: G. C. Merriam, 1960.
5. Sayers, D. L., *The Mind of the Maker.* New York: Harcourt, Brace, 194.
6. Glass, R. L., and S. A. Conger, “Research software talks: Intellectual or clerical?” *Information or Management*, 23, 4 (1992). 作者提出關於軟體需求的度量結果是80%的智力和20%的書記工作. Fjelstadt and Hamlen, 1979, 對應用軟體維護得到了相同的結果. 對於完整的任務而言, 據我所知還沒有類似的測量.
7. Herzberg, F., B. Mausner, and B. B. Sayderman. *The Motivation to Work,* 2nd ed. London: Wiley, 1959.
8. Cox, B. J., “There is a silver bullet,” *Byte* (Oct., 1990), pp. 209-218.
9. Harel, D., “Biting the silver bullet: Toward a brighter future for system development,” Computer (Jan., 1992), pp. 8-20.
10. Parnas, D. L., “Software aspects of strategic defense systems,” *Communication of the ACM,* 28, 12 (Dec., 1985), pp. 1326-1335.
11. Turski, W. M., “And no philosophers' stone, either,” in Information Processing 86, H. J. Kugler ed., Amsterdam: Elsevier Science, North Holland, 1986, pp. 1077-1080.
12. Glass, R. L., and S. A. Conger, “Research software tasks: Intellectual or clerical?” *Information and Management*, 23, 4 (1992), pp. 183-192.
13. *Review of Electronic Digital Computers, Proceedings of a Joint AIEEIRE Computer Conference* (Philadelphia, Dec. 10-12, 1951). New York: American Institute of Electrical Engineers. pp. 13-20.
14. *Ibid.,* pp. 36, 68, 71, 97.
15. *Proceedings of the Eastern Joint Computer Conference,* (Washington, Dec. 8-10, 1953). New York: Institute of Electrical Engineers. pp. 45-47.
16. *Proceedings of the 1955 Western Joint Computer Conference,* (Los Angeles, March 1 -3, 1955). New York: Institute of Electrical Engineers.
17. Everett, R. R., C. A. Zraket, and H. D. Bennington, “SAGE - a data processing system for air defense,” *Proceedings of the Eastern Joint Computer Conference* (Washington, Dec. 11-13, 1957). New York: Institute of Electrical Engineers.
18. *Harel D., Lachover H., Haamad A., Pnueli A., Politi M., Sherman R., Shtul-Traurig A.* “Statemate: *A* working environment for the development of complex reactive systems,” *IEEE Trans. on SE,* 16, 4 (1990), pp. 403-444.
19. Jones, C., *Assessment and Control of Software Risks.* Engltwood Cliffs, N. J.: Prentice-Hall, 1994. p. 619.
20. Coqui, H., “Corporate survival: The software dimension,” *Focus '89*, Cannes, 1989.
21. Coggins, J. M., “Designing C++ libraries,” *C++ Journal.* 1, 1 (June, 1990), pp. 25-32.
22. 時態是將來時, 我所瞭解到的是, 沒有類似關於第15次應用的報告.
23. Jones, 引文, p. 604.
24. Huang, Weigiao, “Industrializing software production,” *Proceedings ACM 1988 Computer Science Conference.* 1988. Atlanta. 我覺得在類似的安排中, 缺乏個人工作機會的增長.
25. 關於重用的整個IEEE Software1994年9月期刊.
26. Jones, 引文, p. 323.

27. Jones, 引文, p. 329.
28. Yourdon, E., *Decline and Fall of the American Programmer*. Englewood Cliffs, N. J.: Yourdon Press, 1992. p. 22.
29. Glass, R. L., "Glass" (專欄), *System Development.* (Jan., 1988), pp. 4-5.

# 第**18**章

1. Boehm, B. W., *Software Engineering Economics.* Englewood Cliffs, N. J.: Prentice-Hall, 1981. p. 81-84.
2. McCarthy, J., "21 Rules for Delivering Great Software on Time," Software World USA Conference, Washington (Sept. 1994).

# 第**19**章

未被引用的材料源自於私下交流.

1. 關於這個痛苦的話題, 參見Niklaus Wirth "A plea for lean software," *Computer*, 28, 2 (Feb., 1995), pp. 64-68.
2. Coleman, D., "Word 6.0 packs in features; update slowed by baggage," *MacWeek*, 8, 38 (Sept. 26, 1994) , p. 1.
3. 在發佈安裝之後, 一些機器語言和編程語言命令的概率資料被發表. 例子可參見J. Hennessy and D. Patterson, *Computer Architecture*. 儘管這些概率資料從不會精確匹配, 但對構建後續的產品非常有用. 據我所知, 在產品設計之前沒有任何書面的概率估計, 事先估計和實際情況的比較就更少. Ken Brooks建議即使只有少數人會作出答復, 現在Internet上的公告牌爲提供成本更低廉的方法, 從新產品的預期用戶獲取資料.
4. Conklin, J., and M. Begeman, "gIBIS: A hypertext Tool for Exploratory Policy Descussion," *ACM Transactions on Office Information Systems,* Oct. 1988. p. 303-331.
5. Englebart, D., and W. English, "A research center for augmenting human intellect," *AFIPS Conference Proceedings, Fall Joint Computer Conference.* San Francisco (Dec. 9-11, 1968). p. 395-410.
6. Apple Computer, Inc., *Macintosh Human Interface Guidelines*, Reading, Mass.: Addison-Wesley, 1992.
7. Apple Desk Top Bus在電氣上可以控制兩個滑鼠, 但作業系統並未提供類似功能.
8. Royce, W. W., 1970. "Managing the development of large software system, s: Concepts and techniques," *Proceedings, WESCON* (Aug., 1970). 在*ICSE 9 Proceedings*上重新發表. Royce和其他人均認爲軟體過程從始至終不修訂前期文檔是不可能的; 模型是作爲理想情況和概念提出的. D. L. Parnas, and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, SE-12, (Feb., 1986), p. 251-257.
9. DOD-STD-2167重新制訂的工作產生了DOD-STD-2167A (1988), 它允許但並爲制訂新的模型如螺旋模型等. Boehm報告指出: 不幸的是, 2167A所參考的軍標MILSPECS和說明性的例子依然是面向瀑布模型的, 因此依然繼續使用瀑布模型. Larry Druffel和George Heilmeyer所領導的國防科學委員會(Defence Science Board Task Force), 在他們1994年的報告"Report of the DSB task force on acquiring defense software commercially"中曾提倡大規模的使用更現代的模型.
10. Mills, H., "Top-down programming in large systems," in *Debugging Techniques in Large Systems,* R.Rustin ed., Englewood Cliffs, N. J.: Prentice-Hall, 1971.
11. Parnas, D. L., "On the design and development of program families," *IEEE Trans. on Software Engineering*, SE-2, 1 (March, 1976), p. 1-9; Parnas, D. L., "Designing software for ease

of extension and construction," *IEEE Trans. on Software Engineering,* SE-5, 2 (March, 1979), p. 128-138.
12. D. Harel, "Biting the silver bullet," *Computer*, (Jan., 1992), p. 8-20.
13. 信心隱藏方面的開創性文章是: Parnas, D. L., "Information distribution aspects of design methodology," *Carnegie-Mellon Univ., Dept. Of Computer Science Technical Report.* (Feb., 1971); Parnas D. L., "A technique for software module specification with examples," *Comm. ACM*, 5, 5 (May, 1972), p. 330-336; Parnas, D. L. (1972). "On the criteria to be used in decomprosing systems into modules," *Comm. ACM,* 5, 12 (Dec., 1972), p. 1053-1058.
14. 物件的思想首先由Hoare and Dijkstra提出, 但是第一個和最有影響力的案例是Dahl and Nygaard發明的Simula-67語言.
15. Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, N. J.: Prentice-Hall, 1981. pp. 83-94; 470-472.
16. Abdel-Hamid, T., and S. Madnick, Software Project Dynamics: An Integrated Approach. Ch. 19, "Model enhancement and Brooks's law." Englewood Cliffs, N. J.: Prentice-Hall, 1991.
17. Stutzke, R. D., "A mathematical expression of Brooks's Law," In *Ninth International Forum on COCOMO and Cost Modeling*. Los Angeles, 1994.
18. DeMarco, T., and T. Lister, *Peopleware: Productive Projects and Teams.* New York: Dorset House, 1987.
19. Pius XI, Encyclical *Quadragesimo Anno*, [Ihm, Claudia Carlen. (ed.). *The Papal Encyclicals 1903-1939*. Raleigh, N. C.: McGrath. P. 428.]
20. Schumacher, E. F., *Small Is Beautiful: Economics as if People Mattered*. Perennian Library Edition. New York: Harper and Row, 1973. P. 244.
21. Schumacher, 引文, p. 34.
22. 一則發人深省的海報聲稱: "言論自由屬於擁有它們的人."
23. Bush, V., "That we may think," *Atlantic Monthly*, 176, 1 (Apr., 1945), p. 101-108.
24. Unix的發明人Ken Thompson of Bell Labs很早就認識到大螢幕對編程的重要性. 他在他原始的Tektronix電子顯像管上發明了在兩列中顯示120行代碼的方法. 他在整個高速顯像管和小型視窗的時代中堅持使用該終端.

| 英文 | 中文 |
|---|---|
| accounting | 管理 |
| Ada | Ada語言 |
| administrator | 管理員 |
| Adobe Photoshop | N/A |
| advancement, dual ladder of | 兩條職位晉升線 |
| advisor, testing | 測試顧問系統 |
| Aiken, H. H. | N/A |
| aiplane-seat metaphor | "飛機坐艙座椅"比喻 |
| Algol | Algol語言 |
| algorithm | 演算法 |
| allocation, dynamic memory | 動態記憶體分配 |
| alpha test | alpha 測試 |

| | |
|---|---|
| alpha version | alpha版本 |
| Alto personal workstation | Alto個人工作站 |
| ANSI | 美國國家標準化組織 |
| APL | APL語言 |
| Apple Computer, inc. | 美國Apple電腦公司 |
| Apple Desk Top Bus | Apple桌面匯流排 |
| Apple Lisa | Apple Lisa型電腦 |
| Apple Macintosh | Apple Macintosh型電腦 |
| AppleScript | AppleScript語言 |
| architect | 體系結構師 |
| architecture | 體系結構 |
| archive, chronological | 根據時間順序歸檔 |
| aristocracy | 貴族專政 |
| Aristotle | 亞里斯多德 |
| Aron, J. | N/A |
| ARPA network | ARPA網路 |
| artifical intelligence | 人工智慧 |
| assembler | 彙編 |
| authority | 權威 |
| AutoCad | AutoCad軟體 |
| AutoLisp | AutoLisp語言 |
| automatic programming | 自動編程 |
| Bach, J.S. | N/A |
| Backus, J. W. | N/A |
| Backus-Naur Form | 巴科斯範式 |
| Backer, F. T. | N/A |
| Balzer, R. | N/A |
| Bardain, E. F. | N/A |
| barrier, sociological | 社會性障礙 |
| Begeman, M. | N/A |
| Belady, L. | N/A |
| Bell Northern Research | Bell北方研究所 |
| Bell Telephone Laboratories | Bell電話實驗室 |
| Bell, C. G. | N/A |
| Bengough, W. | N/A |
| Bennington, H. D. | N/A |
| beta version | beta版本 |
| Bible | 聖經 |
| Bierly, R. | N/A |

| | |
|---|---|
| Blaauw, G. A. | N/A |
| Bloch, E. | N/A |
| Blum, B. | N/A |
| Boehm, B. W. | N/A |
| Boehm, E. M. | N/A |
| Boes, H. | N/A |
| Bohl, M. | N/A |
| Bohm, C. | N/A |
| Booch, G. | N/A |
| Boudot-Lamotte, E. | N/A |
| brass bullet | 銅質子彈 |
| breakthrough | 突破 |
| 英文 英文 英文 工作手冊 | workbook |

| | |
|---|---|
| Abdel-Hamid, T. 工作站 | workstation |

abstract data type 萬維網

accident 後門介面

Breughel, P. , the Elder

Brooks's Law
Brooks, F. P. Jr.
Brooks, K. P.
Brooks, N. G.
Buchanan, B.
Buchholz, W.
budget
  access
  size
bug
  documented
Build-every-night approach


World-Wide Web

wormhole

N/A

Brooks法則
N/A
N/A
N/A
N/A
N/A
預算
  訪問
  規模
N/A
  文檔化
"每晚重建"方法

| | |
|---|---|
| build, incremental system | 增量式開發系統 |
| build-to-budget strategy | 按預算開發的策略 |
| build-up, manpower | 內建，人力 |
| building a program | 構建程式 |
| bullet, brass silver | 銅質子彈 銀彈 |
| Burke, E. | N/A |
| Burke, A. W. | N/A |
| Bush, V. | N/A |
| Butler, S. | N/A |
| buy versus build | 購買和自行開發 |
| C++ | C++語言 |
| Cambridge Multiple-Access System | 劍橋多重訪問系統 |
| Cambridge University | 劍橋大學 |
| Campbell, E. | N/A |
| Canova, A. | N/A |
| Capp, A. | N/A |
| Carnegie-Mellon University | 卡內基－梅隆大學 |
| CASE statement | CASE語句 |
| Case, R. P. | N/A |
| Cashman, T. J. | N/A |
| cathedral | 大教堂 |
| change summany | 變更小結 |
| change control of design organization | 變更 變更控制 設計 組織機構 |
| changeability | 可變性 |
| channel | 通道 |
| chemical engineering | 化學工業 |
| chief programmer | 首席程式師 |
| ClarisWorks | N/A |
| class | 類 |
| Clements, P. C. | N/A |
| clerk, program | 程式職員 |
| client-server system | 客戶機－伺服器系統 |
| Clingen, C. T. | N/A |
| COBOL | COBOL語言 |
| Codd, E. F. | N/A |

| Coding War Games | Coding War Gamesx項目 |
| --- | --- |
| coding | 編碼 |
| Coggins, J. M. | N/A |
| Coleman, D. | N/A |
| command key | 命令鍵 |
| command | 命令 |
| comment | 評論 |
| committee | 委員會 |
| communication | 交流、溝通 |
| compatibility | 相容性 |
| compile-time operation | 編譯操作 |
| compiler | 編譯器 |
| complexity | 複雜度 |
| arbitrary | 任意的、隨意的 |
| conceptual | 概念複雜度 |
| component debugging | 單元測試 |
| component | 構件、組件 |
| dummy | 僞構（組）件 |
| comprehensibility | 理解程度 |
| computer facility | 電腦設施 |
| conceptual construct | 概念性結構要素 |
| conceptual integrity | 概念完整性 |
| conceptual structure | 概念結構 |
| conference | 大會 |
| conformity | 一致性 |
| Conger, S. A. | N/A |
| Conklin, J. | N/A |
| control program | 控制程式 |
| convergence of debugging | 調試的收斂性 |
| Conway, M. E. | N/A |
| Conway, R. W. | N/A |
| Cooley, J. W. | N/A |
| copilot | 副手 |
| Coqui, H. | N/A |
| Corbato, F. J. | N/A |
| Cornell University | 康奈爾大學 |
| Cosgrove, J. | N/A |
| cost | 成本 |
| cost, development | 開發成本 |
| front-loaded | 先行投入 |
| courge, managerial | 管理勇氣 |

court, for design disputes 仲裁設計分歧的會議

Cox, B. J. N/A
Crabbe, G. N/A
creation, component stages 構件階段的創造

creative joy 創造的樂趣
creative style 創造性
creative work 創造性工作
creativity 創造力
critical-path schedule 關鍵路徑進度
Crockwell, D. N/A
Crowley, W. R. N/A
cursor 游標
customizability 客戶化
customization 定制

d'Orbais, J. N/A
Dahl, O. J. N/A
Daley, R. C. N/A
data base 資料基礎
data service 資料服務
database 資料庫
datatype, abstract 抽象資料類型
date, estimated 估計日期
  scheduled 計畫日期
debugging aid 調試輔助程式
debugging, component, 構件單元測試
  high-level language, 高階語言
  interactive, 互動式
  on-machine, 本機調試
  sequential nature of, 次序特性
  system 系統集成調試

DEC PDP-8 DEC PDP-8型電腦

DEC VMS operating system DEC VMS作業系統

DECLARE DECLARE語句
Defense Science Board Task Force on Military Software 國防科學委員會軍事軟體工作組

| | |
|---|---|
| Defense Science Board | 國防科學委員會 |
| DeMarco, T. | N/A |
| democracy | 民主政治 |
| Department of Defense | 國防部 |
| dependability of debugging vehicle | 可靠的調試平臺 |
| description; *See* specification | 描述，參見規格說明 |
| design change | 設計變更 |
| design-for-change | 爲變更設計 |
| designer, great | 卓越的設計人員 |
| desktop metaphor | 桌面的類比 |
| development, incremental | 增量式開發 |
| diagram | 圖 |
| difference in judgement | 觀點差異 |
| Digitek Corporation | Digitek公司 |
| Dijkstra, E. W. | N/A |
| director, technical, role of | 技術主管的角色 |
| discipline | 學科、領域、規範 |
| Disk Operation System, IBM 1410-7010 | IBM 1410-7010磁片作業系統 |
| display terminal | 顯示終端 |
| division of labor | 人力劃分 |
| DO...WHILE | DO□□WHILE語句 |
| document | 文檔 |
| documentation system | 文檔系統 |
| documentation | 文檔 |
| DOD-STD-2167 | 軍標DOD-STD-2167 |
| DOD-STD-2167A | 軍標DOD-STD-2167A |
| Dragon voice recognition system | Dragon語音識別系統 |
| Druffel, L. | N/A |
| dual ladder of advancement | 兩條職位晉升線 |
| dummy component | 僞構件 |
| dump, memory | 記憶體傾印 |
| Durfee, B. | N/A |

| | |
|---|---|
| ease of use | 易用性 |
| Eastman Kodak Company | Eastman Kodak公司 |
| Eckman, D. | N/A |
| editor, job description for text | 編輯，職責描述 文字 |
| Einstein, A. | N/A |
| electronic mainl | 電子郵件 |
| electronic notebook | 電子手冊 |
| Electronic Switching System | 電子交換系統 |
| encapsulation | 封裝 |
| Engelbart, D. C. | N/A |
| English, W. | N/A |
| entropy | 熵 |
| environment | 環境 |
| Erikson, W. J. | N/A |
| Ershov, A. P. | N/A |
| Eschapasse, M. | N/A |
| essence | 根本（困難） |
| estimating | 估計 |
| Evans, B. O. | N/A |
| Everett, R. R. | N/A |
| Excel | Excel軟體 |
| expert system | 專家系統 |
| extension | 擴展 |
| Fagg, P. | N/A |
| Falkoff, A. D. | N/A |
| family, software product | 軟體產品族 |
| Farr, L. | N/A |
| Fast Fourier Transform | 快速傅立葉變換 |
| featuritis | 盲目的功能 |
| Ferrell, J. | N/A |
| file, dummy miniature | 僞文件 縮影 |
| filters | 篩檢程式 |
| Fjelstadt | N/A |
| floorspace | 空間 |
| flow arrow | 箭頭 |
| flow chart | 資料流程圖 |
| forecast | 預測 |
| formal definition | 形式化定義 |

| | |
|---|---|
| formal progression of release | 正式發佈的進展 |
| formality, of written proposals | 書面建議的正式性 |
| Fortran | Fortran語言 |
| Fortran, H. | N/A |
| FoxPro database | FoxPro資料庫 |
| Franklin, B. (Poor Richard) | N/A |
| Franklin, J. W. | N/A |
| frequency data | 頻率資料 |
| frequency guessing | 頻率猜測 |
| fusion | 融合 |
| Galloping Gertie, Tacoma Narrows Bridge | 塔科馬大橋 |
| Gantt Chart | 甘特圖 |
| General Electric Company | 通用電氣公司 |
| generator | 發生器 |
| gIBIS | gIBIS系統 |
| Ginzberg, M. G. | N/A |
| Glass, R. L. | N/A |
| Glegg, G. L. | N/A |
| Global Positioning System | 全球定位系統 |
| GO TO | GO TO語句 |
| God | 上帝 |
| Godel | N/A |
| Goethe, J. W. won | N/A |
| Gold, M. M. | N/A |
| Goldstine, H. H. | N/A |
| Gordon, P. | N/A |
| GOTO | GOTO語句 |
| Grafton, R. B. | N/A |
| Grant, E. E. | N/A |
| graph | 圖 |
| structure | 結構圖 |
| graphical programming | 圖形化編程 |
| great designer | 卓越的設計人員 |
| Green wald, I. D. | N/A |
| growing software | 培育軟體 |
| Gruemberger, F. | N/A |

| | |
|---|---|
| Hamilton, F. | N/A |
| Hamlen | N/A |
| hardware, computer | 電腦硬體 |
| Hardy, H. | N/A |
| Harel, D. L. | N/A |
| Harr, J. | N/A |
| Hayes-Roth, R. | N/A |
| Heilmeyer, G. | N/A |
| Heinlein, R. A. | N/A |
| Hennessy, J. | N/A |
| Henricksen, J. O. | N/A |
| Henry, P. | N/A |
| Herzberg, F. | N/A |
| Hetzel, W. C. | N/A |
| Hezel, K. D. | N/A |
| hierarchical structure | 層次化結構 |
| high-level language, *See* language, high-level | high-level language, 參見language, high-level |
| Hoare, C. A. R. | N/A |
| Homer | N/A |
| Hopkins, M. | N/A |
| Huang, W. | N/A |
| hustle | 進取 |
| Hypercard | Hypercard |
| hypertext | 超文本 |
| IBM 1401 | IBM 1401型電腦 |
| IBM 650 | IBM 650型電腦 |
| IBM 7030 Stretch computer | IBM 7030 Stretch型電腦 |
| IBM 704 | IBM 704型電腦 |
| IBM 709 | IBM 709型電腦 |
| IBM 7090 | IBM 7090型電腦 |
| IBM Corporation | IBM公司 |
| IBM Harvest computer | IBM Harvest型電腦 |
| IBM MVS/370 operating system | IBM MVS/370作業系統 |
| IBM Operating System/360, *See* Operating System/360 | IBM Operating System/360, 參見 Operating System/360 |
| IBM OS-2 operating system | IBM OS-2作業系統 |

| | |
|---|---|
| IBM PC computer | IBM PC電腦 |
| IBM SAGE ANFSQ/7 data processing system | IBM SAGE ANFSQ/7資料處理系統 |
| IBM System/360 Model 165 | IBM System/360 模型165 |
| IBM System/360 Model 30 | IBM System/360 模型30 |
| IBM System/360 Model 65 | IBM System/360 模型65 |
| IBM System/360 Model 75 | IBM System/360 模型75 |
| *IBM System/360 Principles of Operation* | *IBM System/360 Principles of Operation* |
| IBM VM/360 operating system | IBM VM/360作業系統 |
| IBSYS operating system for the 7090 | 7090的IBSYS作業系統 |
| Ichikawa, T. | N/A |
| icon | 圖示 |
| ideas, as stage of creation | 構思，作爲創造的階段 |
| IEEE *Computer* magazine | IEEE 電腦雜誌 |
| IF...THEN...ELSE | IF...THEN...ELSE語句 |
| Ihm, C. C. | N/A |
| implementation | 設計實現 |
| implementations, multiple | 多重實現 |
| implementer | 實現者 |
| incorporation, direct | 直接整合 |
| incremental development | 增量式開發 |
| incremental-build model | 增量開發模型 |
| indenting | 縮進 |
| information hiding | 信息隱藏 |
| information theory | 資訊理論 |
| inheritance | 繼承 |
| initialization | 初始化 |
| input range | 輸入範圍 |
| input-output format | 輸入－輸出格式 |

instrumentation 配備
integrity, conceptual 概念完整性
interaction, as part of creation 交互，作爲創造的一部分
  first of session 第一次會話

interactive debugging 互動式調試
interactive programming 互動式編程

interface 介面、介面
  metaprogramming 元編程
  module 模組
  WIMP WIMP

Interlisp Interlisp語言
International Computers Limited 國際電腦有限公司

Internet Internet
interpreter, for space-saving 解釋程式，用於節省空間

invisibility 不可見性
iteration 迭代
Iverson, K. E. N/A
Jacopini, A. N/A
Jobs, S. N/A
Jones, C. N/A
joys of the craft 行業的樂趣

Kane, M. N/A
keyboard 鍵盤
Keys, W. J. N/A
King, W. R. N/A
Knight, C. R. N/A
Knuth, D. E. N/A
Kugler, H. J. N/A

label 標籤
Lachover, H. N/A
Lake, C. N/A
Landy, D. E. N/A
language description, formal 形式化語言描述

language translator 語言翻譯程式

| | |
|---|---|
| language, fourth-generation | 第四代語言 |
| high-level | 高級 |
| machine | 機器 |
| programming | 編程 |
| scripting | 腳本 |
| late project | 進度落後的專案 |
| lawyer, language | 語言專家 |
| Lehman, M. | N/A |
| Lewis, C. S. | N/A |
| library | 庫 |
| class | 類庫 |
| macro | 宏庫 |
| program | 程式庫 |
| linkage editor | 鏈結編輯程式 |
| Lister, T. | N/A |
| Little, A. D. | N/A |
| Locken, O. S. | N/A |
| Lowry, E. S. | N/A |
| Lucas, P. | N/A |
| Lukasik, S. | N/A |
| Macintosh WIMP interface | Macintosh WIMP介面 |
| Madnick, S. | N/A |
| magic | 魔術 |
| maintenance | 維護 |
| man-month | 人月 |
| management information system (MIS) | 管理資訊系統 |
| manual | 手冊 |
| System/360 | 360系統手冊 |
| market, mass | 大眾市場 |
| matrix management | 矩陣管理 |
| matrix-type organization | 矩陣類型的組織結構 |
| Mausner, B. | N/A |
| Mayer, D. B. | N/A |
| McCarthy, J. | N/A |
| McCracken, D. D. | N/A |
| McDonough, E. | N/A |
| Mealy, G. | N/A |
| measurement | 測量、度量 |

| | |
|---|---|
| medium of creation, tractable | 容易駕馭的創造媒介 |
| meeting, problem action | 問題－行動會議 |
| status review | 狀態檢查會議 |
| memory use pattern | 記憶體使用方式 |
| mentor | 導師 |
| menu | 菜單 |
| Merwin, R. E. | N/A |
| Merwin-Dagget, M. | N/A |
| metaphor | 類比、象徵 |
| metaprogramming | 元編程 |
| microcomputer revolution | 微型電腦革命 |
| microfiche | 微縮膠皮 |
| Microsoft Corporation | 微軟公司 |
| Microsoft Windows | Microsoft Windows |
| Microsoft Word 6.0 | Microsoft Word 6.0 |
| Microsoft Works | Microsoft Works |
| milestone | 里程碑 |
| Mills, H. D. | N/A |
| MILSPEC documentation | MILSPEC軍用標準 |
| mini-decision | 細小（迷你）決定 |
| MiniCad design program | MiniCad設計軟體 |
| MIT | 麻省理工學院 |
| mnemonic name | 助記名 |
| model | 模型 |
| COCOMO | COCOMO |
| incremental-build | 增量開發 |
| spiral | 螺旋 |
| waterfall | 瀑布 |
| Modula | Modula |
| modularity | 模組化 |
| module | 模組 |
| modules, number of | 模組數量 |
| Mooers, C. N. | N/A |
| Moore, S. E. | N/A |
| Morin, L. H. | N/A |
| Mostow, J. | N/A |
| mouse | 滑鼠 |
| moving projects | 項目遷移 |

Mozart, W. A. N/A
MS-DOS MS-DOS
Multics Multics系統
multiple implementations 多重實現

MVS/370 MVS/370

Nammed, A. N/A
Nanus, B. N/A
Naur, P. N/A
Needham, R. M. N/A
Nelson, E. A. N/A
nesting, as documentation aid 嵌套，作爲文檔輔助

network nature of communication 交流的網路特性

Neustadt, R. E. N/A
Newell, A. N/A
Noah N/A
North Carolina State University 北卡羅來納大學

notebook, status system 狀態記錄系統
Nygaard N/A

object 對象
object-oriented design 面向物件設計
object-oriented programming 面向物件編程

objective 目標
cost and performance 成本和性能
space and time 空間和時間

obsolescence 陳舊過時
off-the-shelf package 現貨成品包
office space 辦公室空間
Ogdin, J. L. N/A
open system 開放系統
operating system 作業系統
Operating System/360 作業系統/360
optimism 樂觀主義
option 選項
Orbais, J. d’ N/A
order-of-magnitude improvement 數量級的提高

| | |
|---|---|
| organization chart | 組織結構圖 |
| organization | 組織結構 |
| *OS/360 Concepts and Facilities* | *OS/360*概念和設施 |
| OS/360 Queued Telecommunications Access Method | OS/360佇列遠端通訊訪問方法 |
| OS/360, *See* Operating System/360 | OS/360, 參見Operating System/360 |
| overlay | 覆蓋 |
| overview | 總覽 |
| Ovid | N/A |
| Padegs, A. | N/A |
| paperwork | 文書工作 |
| Parnas families | N/A |
| Parnas, D. L. | N/A |
| partitioning | 分區 |
| Pascal programming language | Pascal語言 |
| Pascal, B. | N/A |
| pass structure | 資料流程 |
| Patrick, R. L. | N/A |
| Patterson, D. | N/A |
| people | 人 |
| *Peopleware: Productive Projects and Teams* | 人件：高生產率的項目和團隊 |
| perfection, requirement for | 要求完美 |
| performance simulator | 性能仿真裝置 |
| performance | 性能 |
| PERT chart | PERT圖 |
| pessimism | 悲觀注意 |
| Peter and Apostle | N/A |
| philosopher's stone | 點金石 |
| Piestrasanta, A. M. | N/A |
| pilot plant | 試驗工廠 |
| pilot system | 試驗系統 |
| pipes | 管道 |
| Pisano, A. | N/A |
| Pius XI | N/A |
| PL/C language | PL/C語言 |
| PL/I | PL/I語言 |

planning 策劃、計畫
Plans and Controls organization 計畫和控制機構

playpen 開發庫
Pnueli, A. N/A
pointing 指針選取
policed system 具有錯誤控制的系統
Politi, M. N/A
Pomeroy, J. W. N/A
Poor Richard (Benjamin Franklin) N/A

Pope, Alexander N/A
Portman, C. N/A
power tools for the mind 創造性活動的強大工具

power, giving up 放棄權力
practice, good software engineering 軟體工程的良好實踐

price 價格
PROCEDURE PROCEDURE語句
procedure, catalogued 具有目錄的過程庫
producer, role of 製片人的角色
product test 產品測試
product, exciting 激動人心的產品
  programming system 編程系統產品
  programming 編程產品

productivity equation 生產率公式
productivity, programming 軟體發展的生產率

program clerk 程式職員
program library 程式庫
program maintenance 程式維護
program name 程式名稱
program products 編程產品
program structure graph 程式結構圖

program 程式
  auxiliary 輔助程式
  self-documenting 自文檔化程式
programmer retraining 程式師的再培訓

programming environment 編程環境

| | |
|---|---|
| programming language | 編程語言 |
| programming product | 編程產品 |
| programming system | 編程系統 |
| programming systems product | 編程系統產品 |
| programming system project | 編程系統專案 |
| programming, automatic | 編程自動化 |
| graphical | 圖形化 |
| visual | 視覺化 |
| progressive refinement | 漸進地精化 |
| Project Mercury Real-Time System | Mercury即時系統項目 |
| project workbook | 專案工作手冊 |
| promotion, in rank | 按級晉升， |
| prototyping, rapid | 快速原型 |
| Publilius | N/A |
| purple-wire technique | 紫色線束的手法 |
| purpose, of a program | 程式的目標 |
| of a variable | 變數的目的 |
| *Quadragesimo Anno, Encyclical* | 教皇通諭*Quadragesimo Anno* |
| quality | 品質 |
| quantization, of change | 變更量子化（階段化） |
| of demand for change | 變更要求的量子化（階段化） |
| Raeder, G. | N/A |
| raise in salary | 漲薪 |
| Ralston, A. | N/A |
| rapid prototyping | 快速原型化 |
| real-time system | 即時系統 |
| realism | 現實主義 |
| realization, step in creation | 物理實現，創造過程的一個步驟 |
| refinement, progressive | 漸進地精化 |
| requirements | 需求 |

| | |
|---|---|
| regenerative schedule disaster | 再現的進度災難 |
| Reims Cathedral | 蘭斯大教堂 |
| release, program | 程式發佈 |
| reliability | 可靠性 |
| remote job entry | 遠程任務項 |
| repartitioning | 重新分配 |
| representation, of information | 資訊的表現形式 |
| requirements refinement | 需求精化 |
| rescheduling | 重新計畫進度 |
| responsibility, versus authority | 責任和權威 |
| Restaurant Antoine | Antoine餐廳 |
| reusable component | 可重用的構件 |
| reuse | 重用 |
| Reynolds, C. H. | N/A |
| role conflict, reducing | 減少角色衝突 |
| ROM, read-only memory | ROM，唯讀記憶體 |
| Roosevelt, F. D. | N/A |
| Rosen, S. | N/A |
| Royce, W. W. | N/A |
| Rustin, R. | N/A |
| Ruth, G. H.(Babe) | N/A |
| Sackman, H. | N/A |
| Salieri, A. | N/A |
| Saltzer, J. H. | N/A |
| Sayderman, B. B. | N/A |
| Sayers, D. L. | N/A |
| scaffolding | 腳手架（測試平臺） |
| scaling up | 擴建、增大規模 |
| Scalzi, C. A. | N/A |
| schedule, *See* Late project cost-optimum | 進度，參見Late project 最優進度 |
| scheduler | 調度程式 |
| scheduling | 計畫安排、進度安排 |
| Schumacher, E. F. | N/A |
| screen | 螢幕 |
| second-system effect | 開發第二個系統的後果（畫蛇添足） |
| secretary | 秘書 |

| | |
|---|---|
| security | 安全性 |
| self-documenting program | 自文檔化程式 |
| Selin, I. | N/A |
| semantics | 語義 |
| Shakespeare, W. | N/A |
| Shannon, E. C. | N/A |
| Share 709 Operating System (SOS) | Share 709作業系統（SOS） |
| Share Operating System for IBM | IBM Share作業系統 |
| Shell, D. L. | N/A |
| Sherman, M. | N/A |
| Sherman, R. | N/A |
| short cuts | 快捷鍵 |
| shrink-wrapped software | 塑膠薄膜包裝的成品軟體 |
| Shtul-Trauring, A. | N/A |
| side effect | 副作用 |
| silver bullet | 銀彈 |
| simplicity | 簡潔 |
| Simula-67 | Simula-67語言 |
| simulator | 仿真裝置 |
| environment | 環境 |
| logic | 邏輯 |
| performance | 性能 |
| size, program | 程式規模 |
| Skwiersky, B. M. | N/A |
| slippage, schedule, *See* Late project | slippage, schedule, 參見Late project |
| Sloane, J. C. | N/A |
| *Small is Beautiful* | 小就是美 |
| Smalltalk | Smalltalk語言 |
| Smith, S. | N/A |
| snapshot | 快照 |
| Snyder, Van | N/A |
| sociological barrier | 社會障礙 |
| Sodahl, L. | N/A |
| *Software Engineering Economics* | 軟體工程經濟學 |
| Software Engineering Institute | 軟體工程研究所 |
| software industry | 軟體工業 |
| *Software Project Dynamics* | 軟體專案動力學 |

| | |
|---|---|
| Sophocles | N/A |
| space allocation | 空間分配 |
| space, memory | 記憶體空間 |
| office | 辦公室空間 |
| program, *See* Size, program | 程式空間，參見程式規模 |
| specialization of function | 限定職責範圍 |
| specification | 規格說明 |
| architectural | 體系結構 |
| functional | 功能 |
| interface | 介面 |
| internal | 內部 |
| performance | 性能 |
| testing the | 測試規格說明 |
| speed, program | 程式運行速度 |
| spiral, pricing-forecasting | 價格預測螺旋 |
| spreadsheet | 試算表 |
| staff group | 團隊 |
| staffing, project | 專案人員配備 |
| Stalnaker, A. W. | N/A |
| standard | 標準 |
| standard, de facto | 事實標準 |
| Stanford Research Institute | 斯坦福研究機構 |
| Stanton, N. | N/A |
| start-up firm | 新興公司 |
| Statemate design tool | 狀態機設計工具 |
| status control | 狀態控制 |
| status report | 狀態報告 |
| status review meeting | 狀態檢查會議 |
| status symbol | 狀態特徵 |
| Steel, T. B., Jr. | N/A |
| Strachey, C. | N/A |
| straightforwardness | 直白 |
| Strategic Defense Initiative | 防禦系統 |
| Stretch Operating System | Stretch作業系統 |
| stub | 占位符 |
| Stutzke, R. D. | N/A |
| subroutine | 副程式 |

Subsidiary Function, Principle of 附屬職能行使原理

superior-subordinate relationship 上下級的關係

support cost 支持成本
surgical team 外科手術隊伍
Sussenguth, E. H. N/A
Swift, J.. N/A
synchronism in file 文件同步
syntax 語法
  abstract 抽象
system build 系統構建
system debugging 系統集成調試
System Development Corporation System Development Corporation（公司）

system integration sublibrary 系統集成子庫

system test 系統測試
system, large 大型系統
  programming 編程
System/360 computer family System/360電腦家族

systems product, programming 系統編程產品

Tacoma Narrows Bridge 塔科馬大橋

Taliaffero, W. M. N/A
target machine 目的機器
Taub, A. H. N/A
Taylor, B. N/A
team, small, sharp 小型、精幹的隊伍
technical director, role of 技術主管的角色

technology, programming 編程技能

telephone log 電話日誌
test case 測試用例
test, component 構件單元測試
  system 系統集成測試
tester 測試人員
testing advisor 測試顧問系統
testing 測試
  regression 回歸測試
  specification 規格說明的測試

| | |
|---|---|
| TESTRAN debugging facility | TESTRAN調試軟體 |
| text-editing system | 文本編輯系統 |
| Thompson, K. | N/A |
| throw-away | 拋棄 |
| time, calendar | 日期時間 |
| machine | 機器時間 |
| Time-Sharing System, PDP-10 | PDP－10分時系統 |
| Time-Sharing System/360 | 分時System/360 |
| time-sharing | 分時 |
| tool | 工具 |
| power, for the mind | 創造性活動的強大工具 |
| toolsmith | 工具維護人員 |
| top-down design | 從上至下設計 |
| top-down programming | 從上至下編程 |
| Tower of Babel | 巴比倫塔 |
| TRAC language | TRAC語言 |
| tracing, program | 程式跟蹤 |
| trade-off, size-function | 規模－功能折衷 |
| size-speed | 規模－速度折衷 |
| training, time for | 培訓時間 |
| transient area | 臨時性空間 |
| Trapnell, F. M. | N/A |
| tree organization | 樹型組織機構 |
| Truman, H. S. | N/A |
| TRW, Inc. | TRW公司 |
| Tukey, J. W. | N/A |
| turnaround time | 周轉時間 |
| turnover, personnel | 演變 |
| Turski, W. M. | N/A |
| two-cursor problem | 雙游標問題 |
| two-handed operation | 雙手操作 |
| type, abstract data | 抽象資料類型 |
| type-checking | 類型檢查 |
| Univac computer | Univac電腦 |
| Unix workstation | Unix工作站 |
| Unix | Unix |

| | |
|---|---|
| University of North Carolina at Chapel Hill | 位於查布林希爾的北卡來羅來納大學 |
| user | 用戶 |
| novice | 新手 |
| power | 專業用戶 |
| USSR Academy of Sciences | 蘇聯科學院 |
| utility program | 實用程式 |
| Vanilla Framework design methodology | 香草框架設計方法學 |
| vehicle machine | 輔助機器 |
| verification | 驗證 |
| version | 版本 |
| alpha | alpha版本 |
| beta | beta版本 |
| Vessey | N/A |
| virtual environment | 虛擬環境 |
| virtual memory | 虛擬記憶體 |
| visual programming | 虛擬編程 |
| visual representation | 虛擬表達 |
| vocabularies, large | 大量的辭彙 |
| Voice Navigator voice recognition system | Voice Navigator語音識別系統 |
| von Neumann, J. | N/A |
| Vyssotsky, V. A. | N/A |
| Walk, K. | N/A |
| Walter, A. B. | N/A |
| Ward, F. | N/A |
| waterfall model | 瀑布模型 |
| Watson, T. J., Jr. | N/A |
| Watson, T. J., Sr. | N/A |
| Weinberg, G. M. | N/A |
| Weinwurm, G. F. | N/A |
| Weiss | N/A |
| *Wells Apocalypse，The* | 威爾斯啓示錄 |
| werewolf | 人狼 |
| Wheller, E. | N/A |
| William III of England, Prince of Orange | N/A |
| Wilson, T. A. | N/A |
| WIMP interface | WIMP介面 |
| window | 窗口 |

| | |
|---|---|
| windows NT operating system | windows NT作業系統 |
| Windows operating system | Windows作業系統 |
| Wirth, N. | N/A |
| Witterrongel, D. M. | N/A |
| Wizard-of-Oz, technique | 嚮導技術 |
| Wolverton, R. W. | N/A |
| Wright, W. V. | N/A |
| Xerox Palo Alto研究中心 | |
| Yourdon, E. | N/A |
| zoom | 推進 |
| Zraket, C. A. | N/A |

# 索引（*Index*）

中文 中文 中文
N/A
抽象資料類型
次要

封底文字

Frederick P. Brooks，Jr.榮獲了電腦領域最具聲望的圖靈獎（A.M.Turing Award）桂冠。美國電腦協會（ACM）稱讚他“對電腦體系結構、作業系統和軟體工程作出了里程碑式的貢獻。”（*landmark contributions to computer architecture, operating system, and software engineering.*）

內容簡介

軟體專案管理領域很少能有著作能像《人月神話》一樣具有影響力和暢銷不衰。Brooks爲任何人管理複雜專案提供了頗具洞察力的見解，既有很多發人深省的觀點，也有大量的軟體工程現實。這些論文出自Brooks的IBM System/360家族和OS/360專案管理經驗。在第一次出版20年後的今天，Brooks重新審視了他原先的觀點，增加了一些新的想法和建議。這既是爲了熟悉原有內容的人們，也爲了第一次閱讀它的讀者。

增加的章節包括（1）原著中的所提出觀點的一些精華，包括Brooks《人月神話》的核心論點：由於人力的劃分，大型專案遭遇的管理問題與小型專案的不同；因此，產品的概念完整性

很關鍵；取得這種統一性是很困難，但並不是不可能的。（2）一個時代以後，Brooks對這些觀點的看法。（3）重新收錄了1986年的經典文章《沒有銀彈》。（4）以及對1986年所下論斷——“在十年內不會出現銀彈。”——現在的認識。