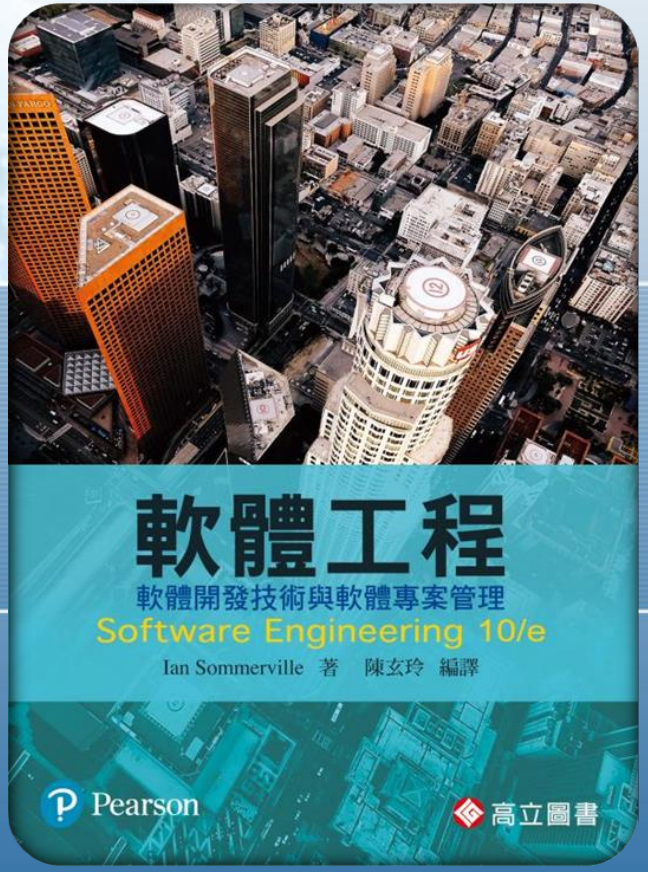
Chapter 9

軟體演進



本章內容

- 已經上線運作的軟體系統，在它的整個生命過程如果要保持有用處就得改變。
- 企業會改變，而使用者期望也會改變，對現有的系統就會產生新需求。
- 軟體的某些部分可能會因為幾個原因而必須修改，例如修正在運作過程中發現的錯誤。因此軟體系統在它的整個生命期間，從初始部署到最後退役都會一直保持適應和演進。
- 系統是重要的企業資產，因此他們必須投資在系統的改變上，以維護這些資產的價值。
- 所以多數大公司花在維護現有系統的預算比花在開發新系統上更多。

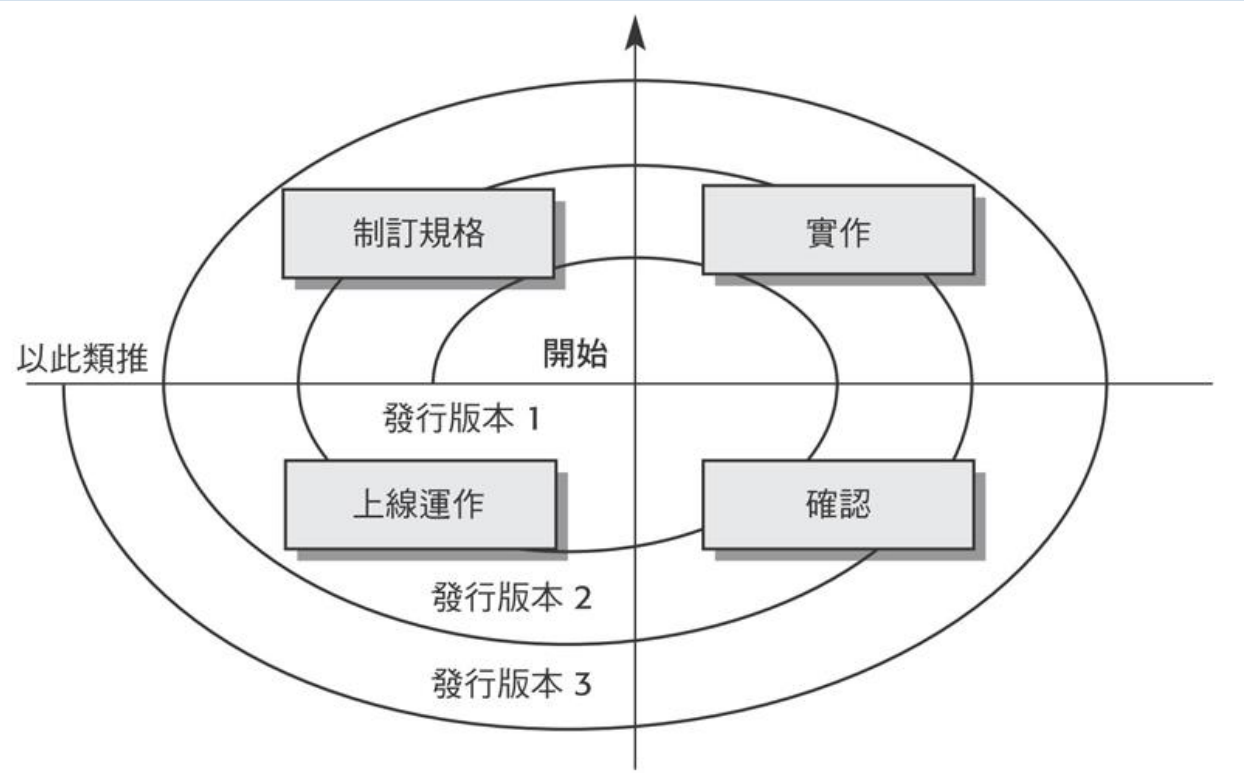


圖 9.1　軟體開發與演進的螺旋式模型

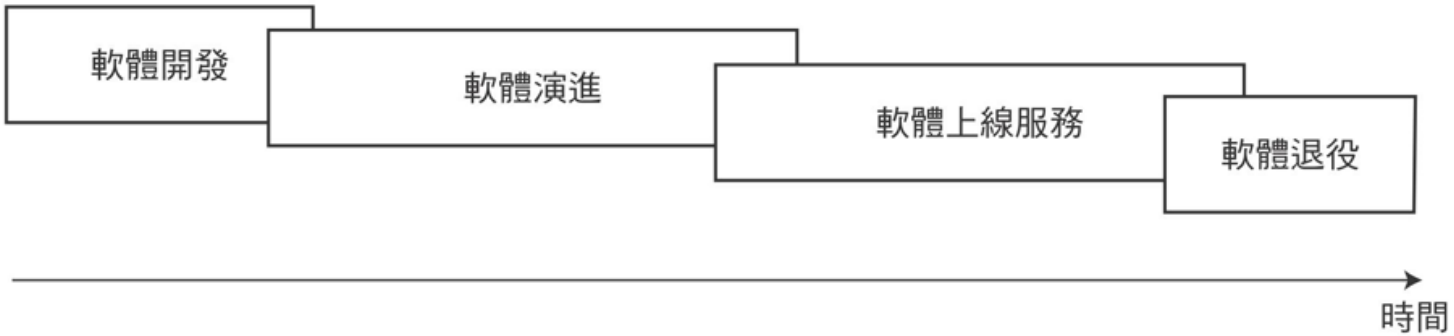


圖 9.2　軟體演進與上線服務

- Rajlich和Bennett (Rajlich and Bennett 2000) 曾提出商業系統的軟體演進生命週期的另一種觀點。這個模型是把演進與上線服務分開，演進 (evolution) 是指對軟體的架構和功能進行重大變更的階段，而上線服務 (servicing) 期間所進行的變更比較是小型而必要的變更。這些階段會彼此重疊，如圖9.2所示。
- 使用者如果遇到問題只能用其他迂迴方式來處理。到最後軟體會退役完全不再使用。

9.1 演進程序

- 軟體系統最適合的演進程序，會根據要維護的軟體類型、使用的開發程序，以及參與人員的技能不同而不同。
- 無論是正式或非正式的系統變更提案 (change proposal)，都是系統演進的驅動力。

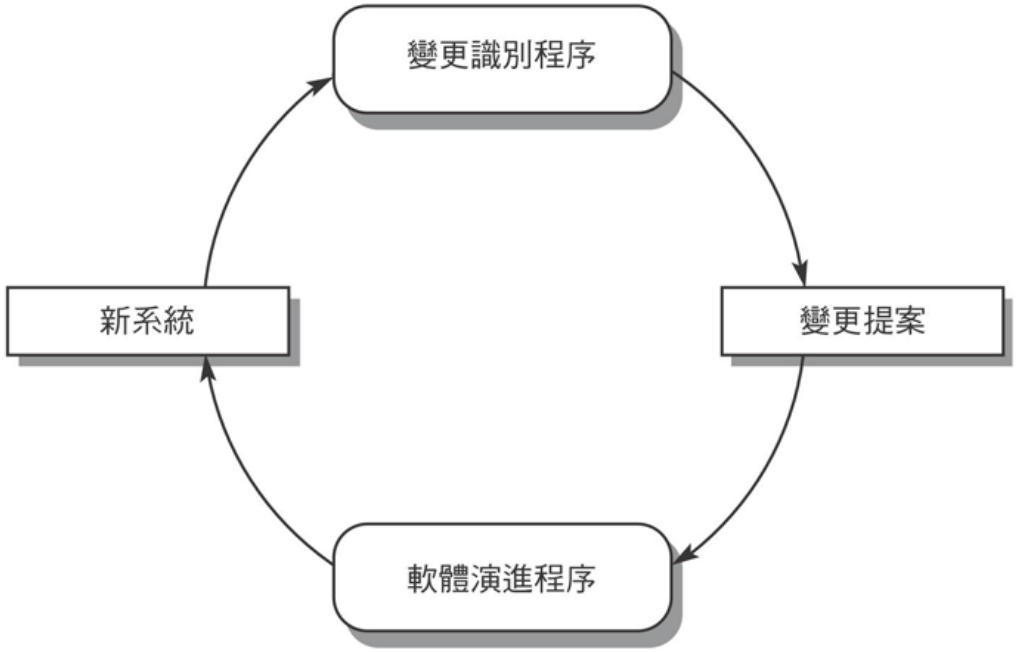


圖 9.3　變更識別與演進的程序

9.1 演進程序

- 每個變更提案在被正式接受前，需要先分析軟體會因此而必須修改哪些元件，以評估變更的成本和影響。

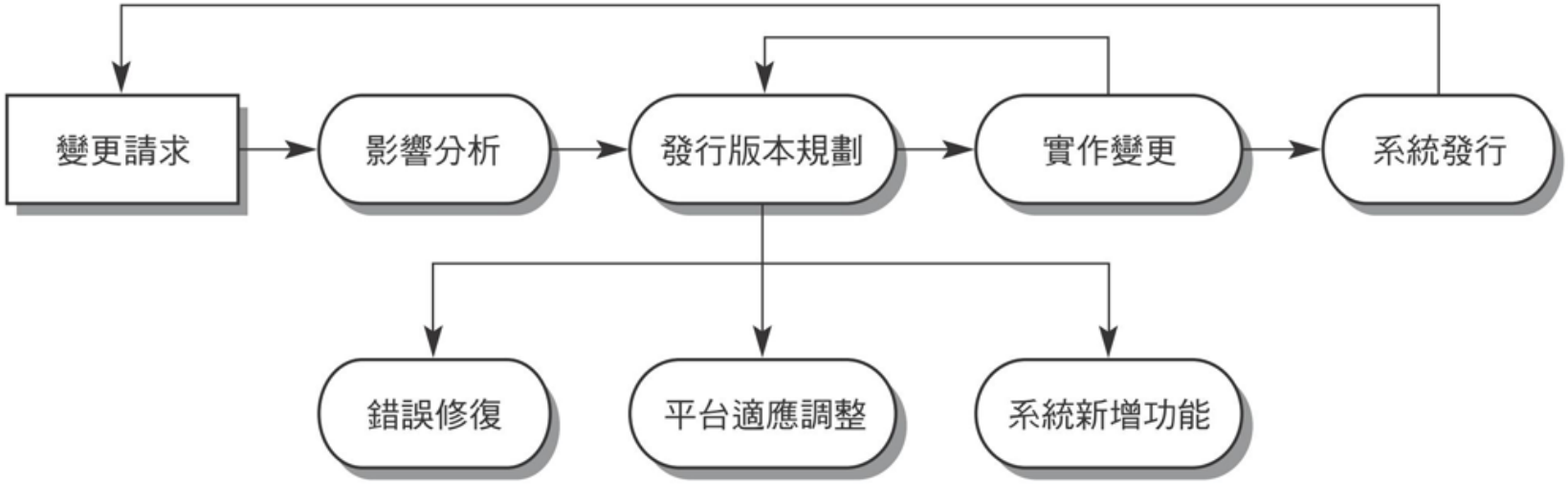


圖 9.4　軟體演進程序的一般模型

9.1 演進程序

- 假如是開發與演進整合在一起的情況，此時變更的實作過程只是開發程序的某一次反覆週期而已。對系統的修訂會經過設計、實作和測試。
- 如果有數個不同的團隊共同參與，那麼開發與演進之間有個重要的差別，就是在實作變更的第一個階段需要先瞭解程式。在這個階段新加入的開發人員必須瞭解程式的結構、它是如何達到功能的，還有該變更提案對程式可能的影響。

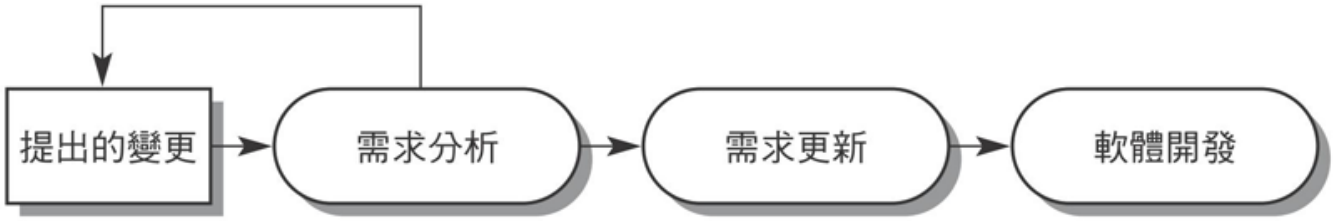


圖 9.5　變更實作

9.1 演進程序

- 會發生緊急變更有以下3個原因：
  1. 找到某個嚴重的系統錯誤，必須立即修復讓系統先能夠繼續運作。
  2. 假如變更對系統的作業環境有無法預期的影響，可能會中斷正常運作。
  3. 假如執行此系統的企業發生非預期的變化，例如新競爭者出現或新法規頒布。

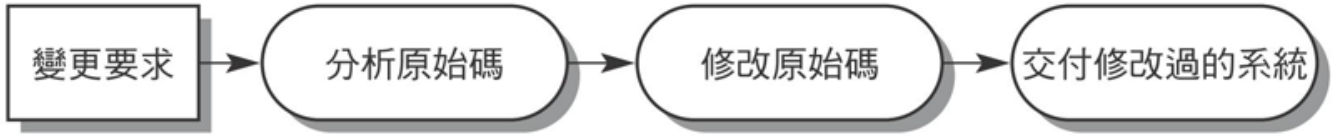


圖 9.6 緊急修復的程序

9.1 演進程序

- 第3章討論過的敏捷式方法和程序是應用在程式開發上，同樣的也可以應用在程式演進上。事實上，因為這些方法是根據增量式開發，因此使得從敏捷式開發到交付後演進的轉變過程應該是無縫接軌的。

9.1 演進程序

- 在把東西從開發團隊移交給負責系統演進的另一團隊時，可能會發生以下兩個問題情況：
  1. 假如之前開發團隊是使用敏捷式方法，但演進團隊寧可使用計畫式 (plan-based) 方法，結果會發生演進團隊可能預期會有詳盡的說明文件來支援演進工作。
  2. 假如開發時使用的是計畫式，但演進團隊想使用敏捷式方法，這時演進團隊可能得從零開始開發自動化的測試案例。還有因為程式碼可能還沒有像在敏捷式開發中一樣重構和簡化過，所以在使用敏捷式開發程序前，需要進行某種程式再造工程 (reengineering) 來改進程式碼。

9.1 演進程序

- 像是測試驅動式開發和自動化迴歸測試 (regression testing) 這些敏捷式技術，在進行系統變更時就很有用。系統變更可表達成使用者故事，而客戶參與可協助安排這些變更在已運作系統上的優先順序。

9.2 舊系統

- 舊系統 (legacy system) 指的是那些依賴過時的程式語言和技術 (開發新系統已不再使用) 的老系統。
- 舊系統的軟體可能需要依賴原來的老舊硬體 (如大型主機)，而且可能還搭配相關的舊程序和流程。
- 舊系統不只是軟體系統，它們其實是涵蓋硬體、軟體、程式庫，還有其他的支援軟體和企業程序，範圍更為廣闊的社會化技術 (sociotechnical) 系統。

9.2 舊系統

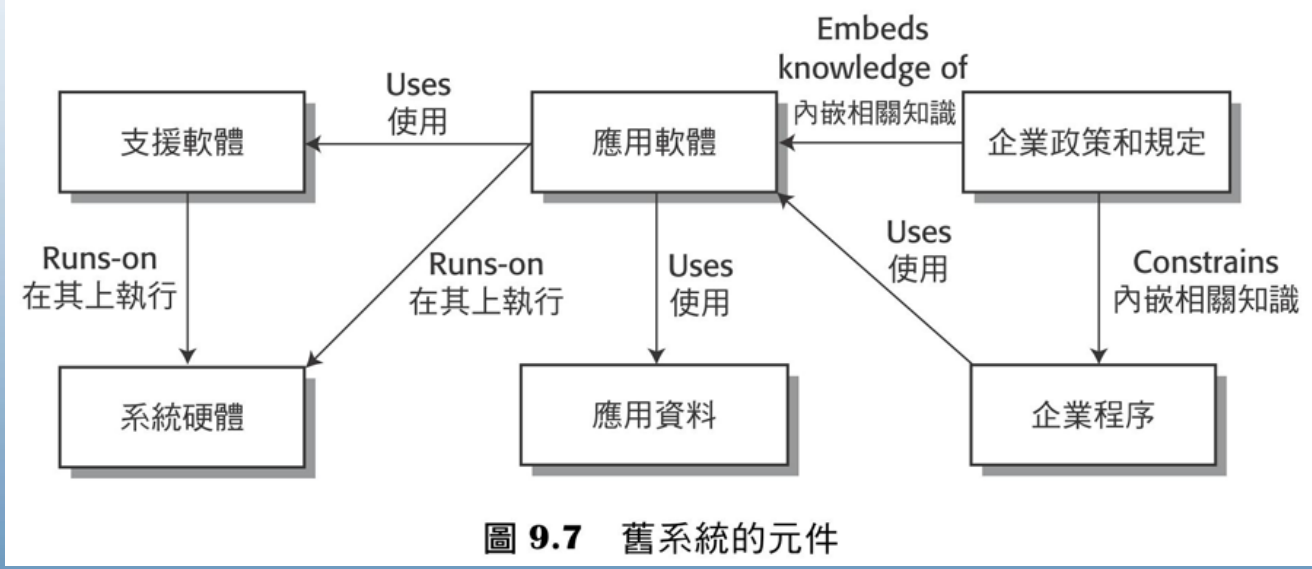


圖 9.7　舊系統的元件

1. **系統硬體 (system hardware)**：舊系統的硬體可能在市場上已經不存在。
2. **支援軟體 (support software)**：舊系統可能需要依賴一些支援軟體。同樣的，它們也可能過時，所以原來的廠商不再提供支援。

9.2 舊系統

3. **應用軟體 (application software)**：為企業提供服務的應用系統通常是由多個應用程式所組成。
4. **應用資料 (application data)**：這是經過應用系統處理過的資料。這些資料可能不一致、可能重複存在多個檔案中，而且也可能散布在多個不同的資料庫。
5. **企業程序 (business processes)**：企業使用這些程序是為了達到某些目標。企業程序如果是根據舊系統設計的，就會受限於它所能提供的功能。
6. **企業政策和規定 (business policies and rules)**：這些是定義企業應該實施和遵循的規範內容。在這些政策和規定中可能內嵌關於舊系統應用程式的使用。

9.2 舊系統

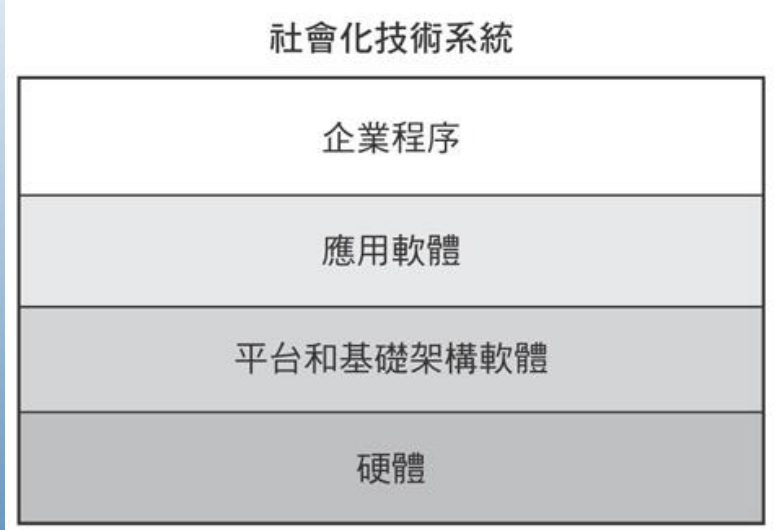


圖 9.8　舊系統的各個層級

- 那為什麼企業不乾脆直接把這些系統換成更現代的新系統呢？最直接的答案就是這樣做太貴而且風險太高。

9.2 舊系統

- 下面列出一些原因：
  1. 舊系統幾乎不會保存有完整的規格。
  2. 企業程序與舊系統的運作方式一定是緊密糾結的。
  3. 重要的企業規則可能早就被內嵌在軟體中，而且在其他地方都沒有紀錄。
  4. 開發新軟體本來風險就很高，所以伴隨著新系統可能會有非預期的問題。

9.2 舊系統

- 維持使用舊系統雖然能避開替換的風險，但隨著系統越來越老，要修改它的成本也無可避免的會越來越貴。
  1. 軟體內部的程式風格和用法習慣很不一致。
  2. 系統有部分甚至全部是用過時的程式語言所實作的，可能不容易找到懂這些語言的人員。
  3. 系統說明文件經常有不足和過時的狀況。
  4. 多年來的維護經常會讓系統結構劣化，使得它越來越難理解。
  5. 系統可能曾經針對空間利用或執行速度做過最佳化(optimization)，所以它在老舊硬體上反而執行效能很好。但如此一來會讓軟體很難懂。
  6. 可能有資料重複、過時、不正確和不完整等問題。

9.2.1 舊系統的管理

- 這包含對舊系統做實際評估，再決定哪種策略最適合這個舊系統的演進。這有4種策略：

  1. 淘汰系統
  2. 保留系統不變而且維持正常維護
  3. 再造系統提升其易維護性
  4. 用新系統替換整個或部分系統

9.2.1 舊系統的管理

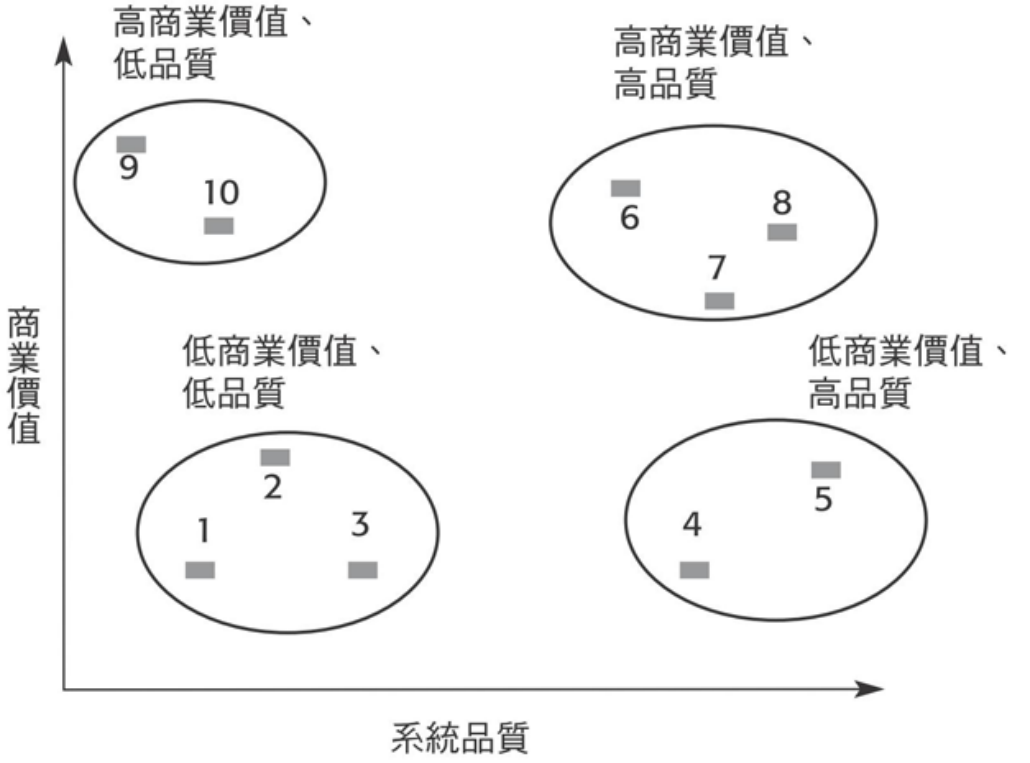


圖 9.9　舊系統的評估範例

9.2.1 舊系統的管理

- 如圖9.9。從這圖可看出系統依據評估結果分成4類：
  1. **低品質、低商業價值**：因此這些系統應該選擇淘汰。
  2. **低品質、高商業價值**：仍有重要貢獻。不過，低品質也意味著維護成本高，應該要進行再造以提高品質；如果有適合的現成系統，則應當用它來取代。
  3. **高品質、低商業價值**：所以如果硬體仍可使用，軟體變更成本也不高的話，就選擇保留系統。但如果軟體需要昂貴的變更成本，那麼就乾脆淘汰。
  4. **高品質、高商業價值**：必須維持運作，維持正常的系統維護。

9.2.1 舊系統的管理

- 要評估系統的商業價值，必須找出系統的利害關係人(stakeholder)，例如系統的終端使用者和經理人，並向他們詢問一連串的問題。其中必須討論的基本主題有4個：

1. **系統的用途**：假如系統只是少數人偶爾使用，那麼商業價值就很低。
2. **所支援的企業程序**：新系統在導入企業時，通常會同時導入搭配的程序。但隨著大環境改變，原來的企業程序可能因過時而不再使用。那麼這個系統的商業價值就很低，因為想用它就被迫要用低效率的舊程序。

9.2.1 舊系統的管理

3. **系統的可信賴度**：假如系統無法令人信賴，問題可能會直接影響顧客，那麼這個系統的商業價值就很低。
4. **系統輸出**：假如企業很依賴這個系統的輸出，那麼此系統就具有高商業價值。

9.2.1 舊系統的管理

因素	問題
供應商的穩定度	供應商是否還存在？供應商的財務狀況是否穩定，是否可以繼續經營？如果供應商退出業界，系統是否可以由其他廠商來維護？
故障率	硬體是否故障率高？支援的軟體若當掉，是否會強迫系統重新開機？
年齡	硬體和軟體使用多久了？硬體和支援軟體愈舊，它們的支援就愈過時。它們可能仍然可以正確的運作，但是也許有更顯著的經濟與商業上的好處，促使換成較現代的系統
執行效能	系統的執行效能是否適當？執行效能的問題是否嚴重影響系統使用者？
支援需求	軟硬體需要哪些當地廠商支援？如果支援成本過高，也許就該考慮替換系統
維護成本	硬體維護和軟體授權的成本為何？舊硬體的維護成本可能會比新硬體高。軟體每年的授權成本可能也很高
交互運作性 (Interoperability)	系統與其他系統的介面有無問題？例如像編譯器是否可以在最新版的作業系統上使用？

圖 9.10 環境評估時使用的因素

9.2.1 舊系統的管理

因素	問題
易理解性	目前系統的原始程式碼有多難理解？它所使用的控制結構有多複雜？使用的變數名稱是否有意義且能夠反映它們的功能？
說明文件	系統有哪些文件可用？這些文件是否完整、一致而且最新？
資料	系統是否有宣告明確的資料模型？資料在不同檔案間複製的程度如何？系統使用的資料是否最新而且一致？
執行效能	應用程式的執行效能是否適當？執行效能的問題是否嚴重影響系統使用者？
程式語言	程式語言是否有最新的編譯器可以用來開發系統？該程式語言是否仍會用來開發新系統？
組態管理	系統的所有部分的所有版本，是否都由組態管理系統來管理？目前系統中使用的元件是否有詳細且明確的版本說明？
測試資料	系統是否有測試資料？當新的功能加入系統時，是否有執行迴歸測試 (regression test) 的紀錄？
個人技能	團隊中是否有具備維護此應用程式所需技能的人？是否有曾經使用過這個系統的人？

圖 9.11　評估應用程式時使用的因素

9.2.1 舊系統的管理

- 你可能會收集一些量化的資料來協助你判斷系統的品質，例如：
  1. **系統變更請求的個數**：這個數字累積愈多，表示系統品質愈低。
  2. **使用者介面的個數**：介面數量愈多，它們之間就愈容易不一致和重複。
  3. **系統使用的資料量**：資料量(檔案數量、資料庫大小等)愈大，資料不一致和錯誤發生機率就愈高。清理舊資料是非常昂貴而費時的過程。

9.3 軟體維護

- 軟體維護 (software maintenance) 是指系統交付後變更系統的一般程序。
- 這裡所說的變更可能只是簡單的修正程式碼錯誤，或是範圍擴大到修正規格錯誤，甚至是增加新需求。
- 軟體維護的類型分成3種：

1. 修正軟體錯誤和弱點

2. 讓軟體適應新平台和環境

3. 為增加新功能和支援新需求而新增系統功能

9.3 軟體維護

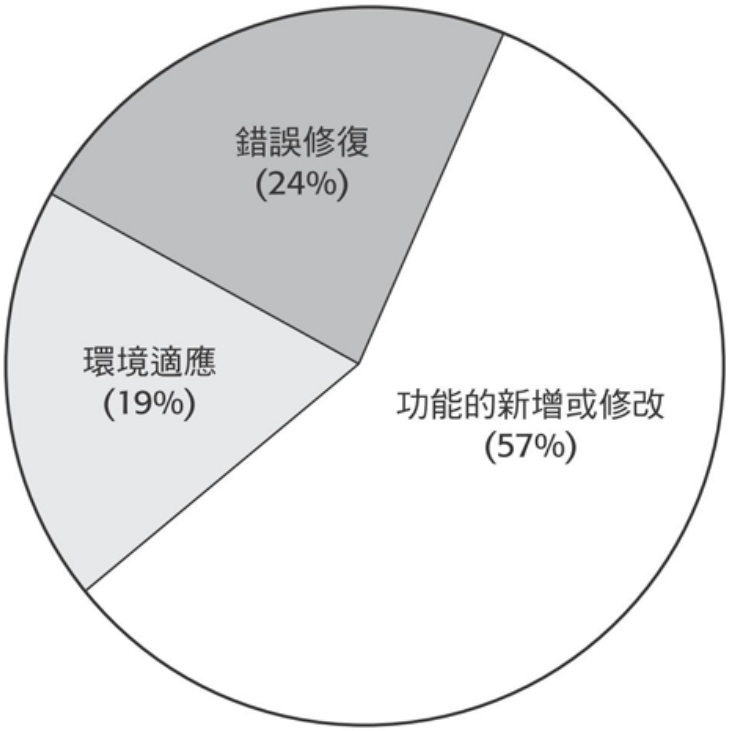


圖 9.12　維護工作量的分配

9.3 軟體維護

- 經驗顯示，系統在維護期間要增加新功能，其成本會比在初始開發時實作同樣的功能其成本會高很多。原因如下：
  1. **新團隊必須花時間瞭解要維護的程式**
  2. **把開發與維護分開，會讓開發團隊沒動力撰寫好維護的軟體**
  3. **程式維護工作較不熱門：**負責維護的人員可能在這些語言方面比較沒有經驗，因此必須先學習這種程式語言才能維護系統。
  4. **程式隨著年齡增長，結構會變差而且更難修改**

9.3.1 預測維護工作量

- 藉由預測變更，可估算出某段期間內的系統維護總成本是多少，而設定該軟體的維護預算。

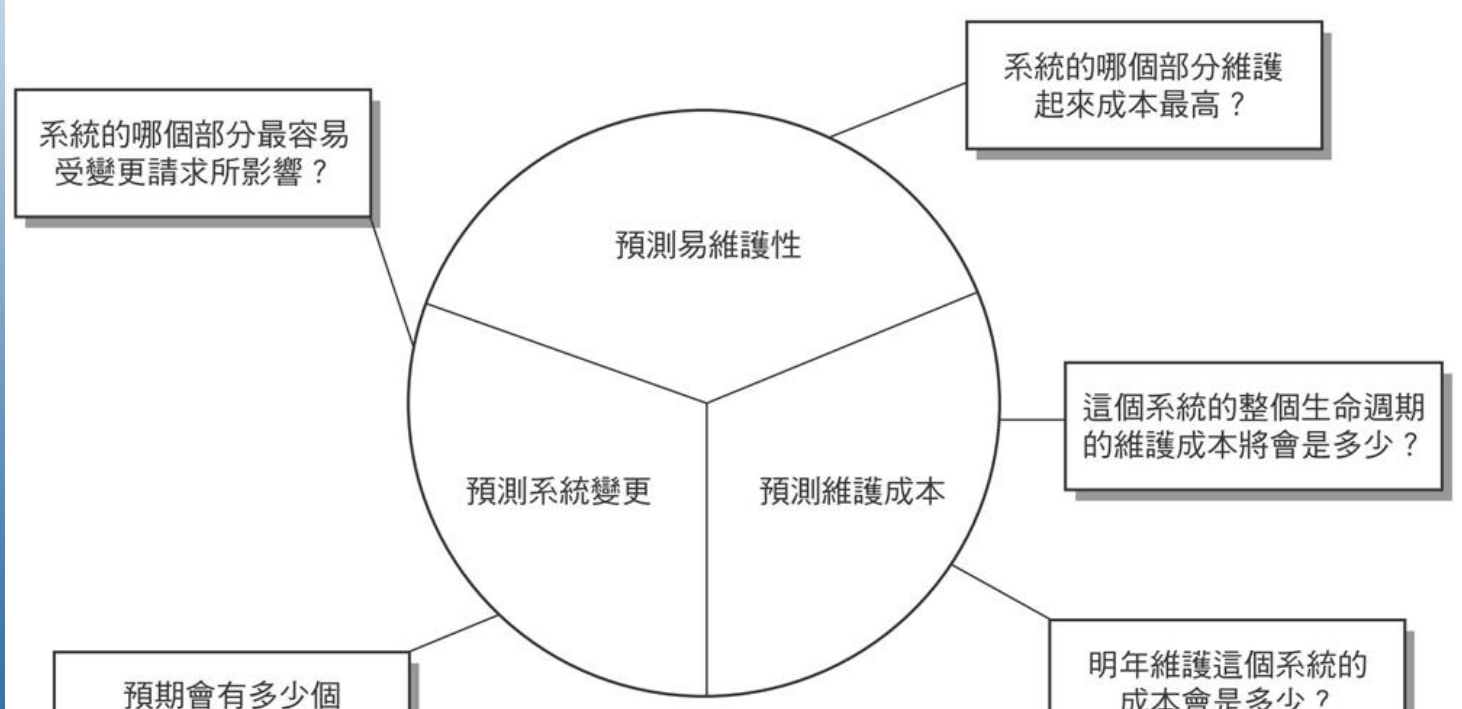


圖 9.13　預測維護工作量

9.3.1 預測維護工作量

- 要預測系統變更的個數，需要瞭解系統與其外在環境之間的關係。環境的改變無法避免的也會造成系統變更。在評估系統與環境之間的關係時，應該評估：

  1. 系統介面的個數和複雜度
  2. 本質易變的系統需求個數
  3. 使用系統的企業程序

9.3.1 預測維護工作量

- 研究結果顯示，系統或元件愈複雜，維護的成本就愈高。
- 因此如果要降低維護成本，應該嘗試將複雜的系統元件替換成較簡單的元件。
- 下列是幾個可用來評估易維護性的程序度量值 (process metrics)：

1. 修正型維護的請求數量

2. 影響分析所得的平均時間

3. 實作變更需求所花的平均時間

4. 重大變更需求的數量

9.3.2 系統再造工程

- 再造工程 (reengineering) 可能包括重新製作系統的說明文件、重構系統的架構、將系統轉換成較現代的程式語言，以及修改和更新系統資料的結構和數值。
- 兩個主要優點：
  1. **降低風險**：重新開發企業關鍵軟體的風險很高，制訂的系統規格中可能有錯誤，也許會有系統開發的問題。
  2. **降低成本**：再造工程的成本明顯比開發新軟體少很多。

9.3.2 系統再造工程

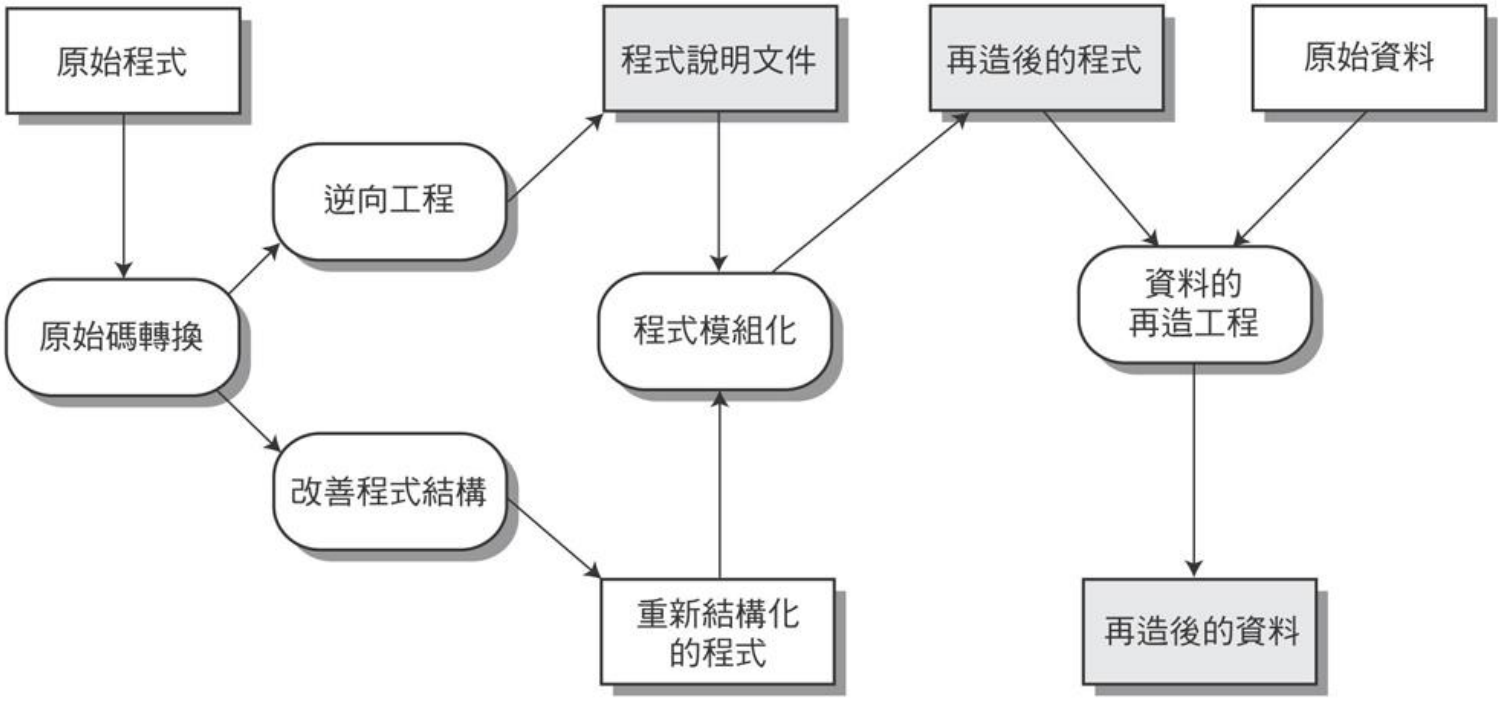


圖 9.14　再造工程的程序

9.3.2 系統再造工程

- 再造工程的程序包含下列幾項活動：
  1. **轉換原始程式碼**：轉換成相同語言的新版或另一種不同的語言。
  2. **逆向工程**
  3. **改善程式結構**：分析並修改程式的控制結構，使其易讀而且易懂。
  4. **程式模組化**：將程式相關的部分集合在一起，並適當的移除重複部分。
  5. **資料再造**：把目前的資料庫轉換成新的結構。通常此時也應該一併清理資料，工作包括尋找和訂正錯誤、移除重複紀錄等。

9.3.2 系統再造工程

- 再造工程的成本必須視再造的程度而定。圖9.15為再造工程的各種可行方式，成本由左到右遞增。

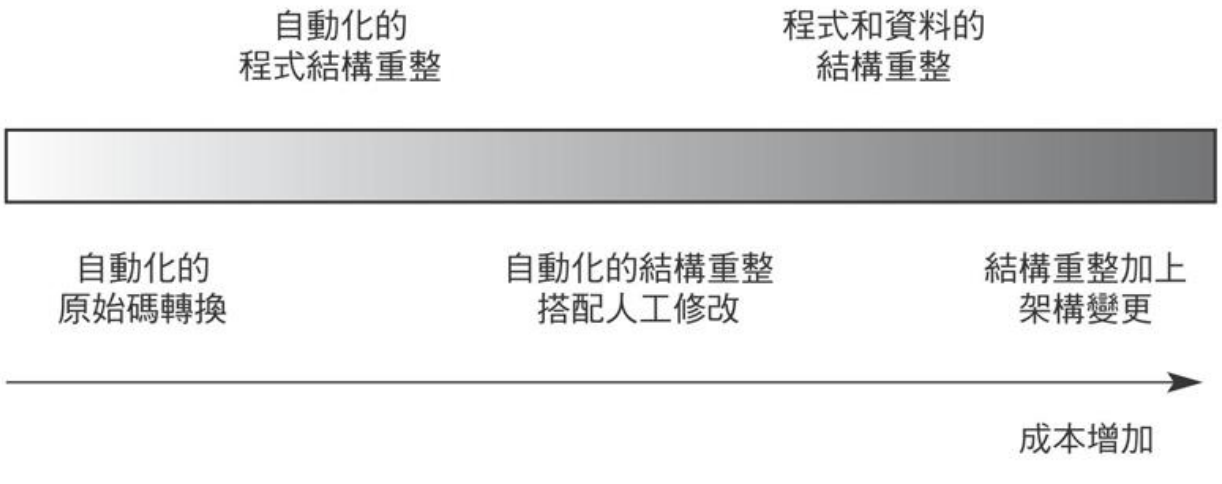


圖 9.15　再造工程的方式

9.3.3 重構

- 重構 (refactoring) 是一個改進程式以減緩程式因為變更而導致品質下降的程序；也就是修改程式來改進它的結構、降低複雜度或變得更容易理解。
- 在重構某個程式時，應該不要加上任何新增功能，而是把注意力集中在程式改進上。因此你可以把重構程序視為是一種「預防性的維護動作」(preventative maintenance)，目的是減少將來變更可能發生的問題。

9.3.3 重構

- 再造工程和重構程序兩者都是想要讓軟體更容易理解和變更，但其實它們是不同的。
- 再造工程發生在系統已經維護一陣子，而且維護成本一直增加的時候。我們可使用自動化工具對舊系統進行處理和再造工程，以建立出更容易維護的新系統。
- 重構程序則是一個在開發和演進程序的整個過程中持續不斷的改進程序。它的目的是避免結構以及程式碼的劣化(degradation)，而導致系統維護的成本和困難度增加。

9.3.3 重構

- Fowler等人提出一些值得改善的程式碼情況。
  1. **程式碼重複 (duplicate code)**：同一個程式中可能有相似的幾段程式碼出現在不同地方，可以將它們整合成一個方法或函式，需要時再呼叫即可。
  2. **方法太長 (long methods)**：假如某個方法太長，應該重新設計成幾個較短的方法。
  3. **switch或case敘述 (switch (case) statements)**：switch敘述是依據某個值的類別來選擇執行路線，因此經常包含重複的部分；switch敘述也可能散佈在程式各處。在物件導向語言經常會選擇使用多型 (polymorphism) 來達到同樣的效果。

9.3.3 重構

4. **資料聚集成塊 (data clumping)**：當相同的資料項目群組(類別中的欄位、方法中的參數)重複出現在同一程式的數個地方，就發生所謂的data clumping現象。這經常可用一個封裝 (encapsulate) 所有這些資料的物件來取代。
5. **推測一般化而預留 (speculative generality)**：有時開發人員會在程式中預留將來一般化要用到的程式碼，這部分通常可直接移除。