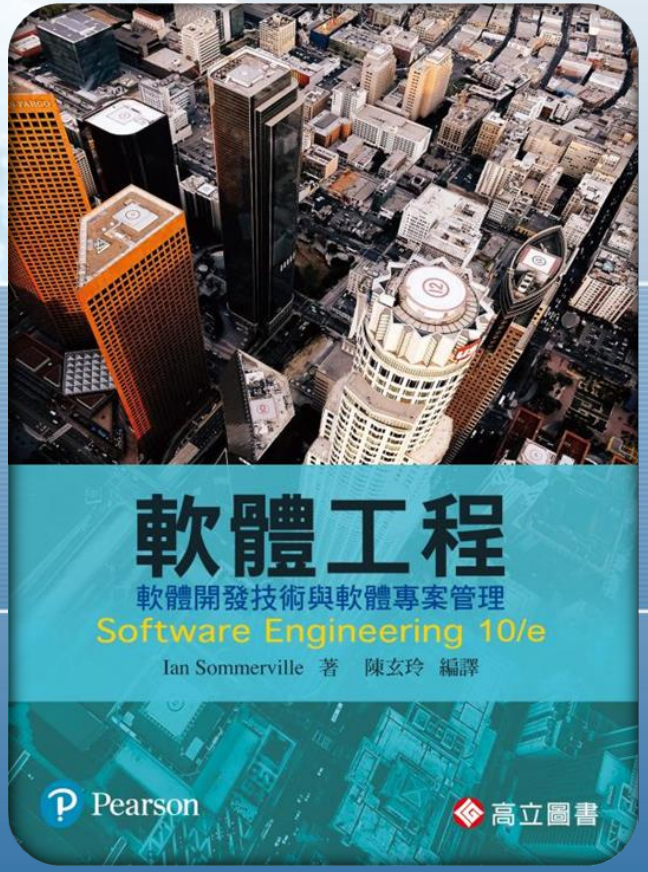
Chapter 10

可信賴的系統



本章內容

- 由於軟體系統對於政府、企業和個人都非常重要，因此這些系統一定要值得信賴。在需要時軟體一定要可以使用，而且要正確運作，不能出現意外的副作用，例如洩漏重要資訊等。
- 可信賴度它涵蓋可用性、可靠性、安全性和保全性這些系統特性。

- 有幾個原因：
  1. 系統故障會影響很多人。
  2. 不可靠、不安全或保全性不佳的系統，使用者經常會拒絕使用。
  3. 系統故障的成本可能非常龐大。
  4. 不可信賴的系統可能會導致失去資訊。

- 因此如果要設計出可信賴的系統，就必須考慮下列情況：
  1. **硬體故障**：可能是因為硬體元件設計不良、製造錯誤、潮濕或高溫等環境因素，或是元件壽命已到。
  2. **軟體故障**：可能是規格、設計或實作上的錯誤所造成。
  3. **操作失誤**：可能是因為系統的人員操作不當所造成。

10.1 可信賴度的性質

- 電腦系統的可信賴度 (dependability) 是系統的一項性質，反映出電腦值得信任的程度。
- 可信賴度有5個主要面向：

1. 可用性 (availability)

2. 可靠性 (reliability)

3. 安全性 (safety)

4. 保全性 (security)

5. 韌性 (resilience)

10.1 可信賴度的性質

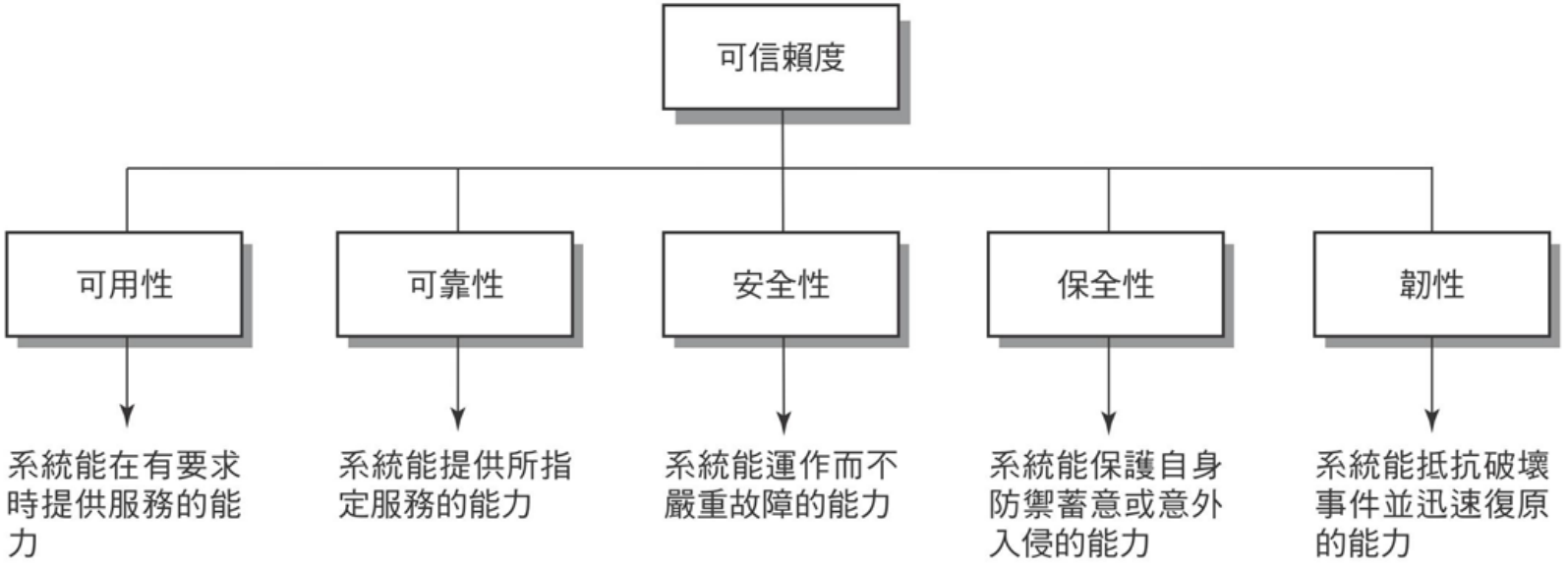


圖 10.1　可信賴度的主要性質

10.1 可信賴度的性質

- 還有其他一些系統性質也會影響系統的可信賴度：
  1. **易修復性 (repairability)**
  2. **易維護性 (maintainability)**
  3. **容錯能力 (error tolerance)**
- 例如系統運作要夠安全，先決條件必須系統是可用而且可靠的；假如系統變得不可靠，有可能是系統被入侵而資料損毀所導致。像對系統進行阻絕服務攻擊 (denial of service attack) 的意圖就是減損系統的可用性。如果系統被病毒感染，使用者對系統的可靠性或安全性就會沒信心，因為病毒可能會改變系統的行為。
- 因此要開發可信賴的軟體，必須確保：

10.1 可信賴度的性質

1. 在軟體規格制訂和開發期間，要避免引進附帶的錯誤到系統中。
2. 所設計的驗證和確認程序能有效找出會影響系統可信賴度的殘留錯誤。
3. 所設計的系統要能容錯 (fault tolerant)，以便在出事時還能持續工作。
4. 要設計能防禦外界攻擊的保護機制，這些攻擊可能會降低系統可用性或保全性。
5. 要根據被部署系統的作業環境正確的設定它與支援軟體。
6. 有內建能辨認外部網路攻擊並抵禦這些攻擊的系統功能。
7. 所設計的系統要能在故障或網路攻擊發生時儘快回復，而且不失去重要資料。

10.1 可信賴度的性質

- 要建構可信賴的系統成本很高。要提高系統的可信賴度，就需要額外的設計、實作與確認成本，像安全關鍵系統的驗證和確認成本就會特別高。確認程序除了確認系統的確符合需求外，可能還必須對外界監管機關證明系統是安全的。
- 圖10.2說明提高可信賴度與成本之間的關係。如果軟體原本就不是很可以信賴，那麼使用較好的軟體工程技術，就能以較低的成本得到很大的進步。但是假如系統原本就使用良好的實務技術，要提升可信賴度所花的成本會高得多，而且相較起來得到的好處也不多。

10.1 可信賴度的性質

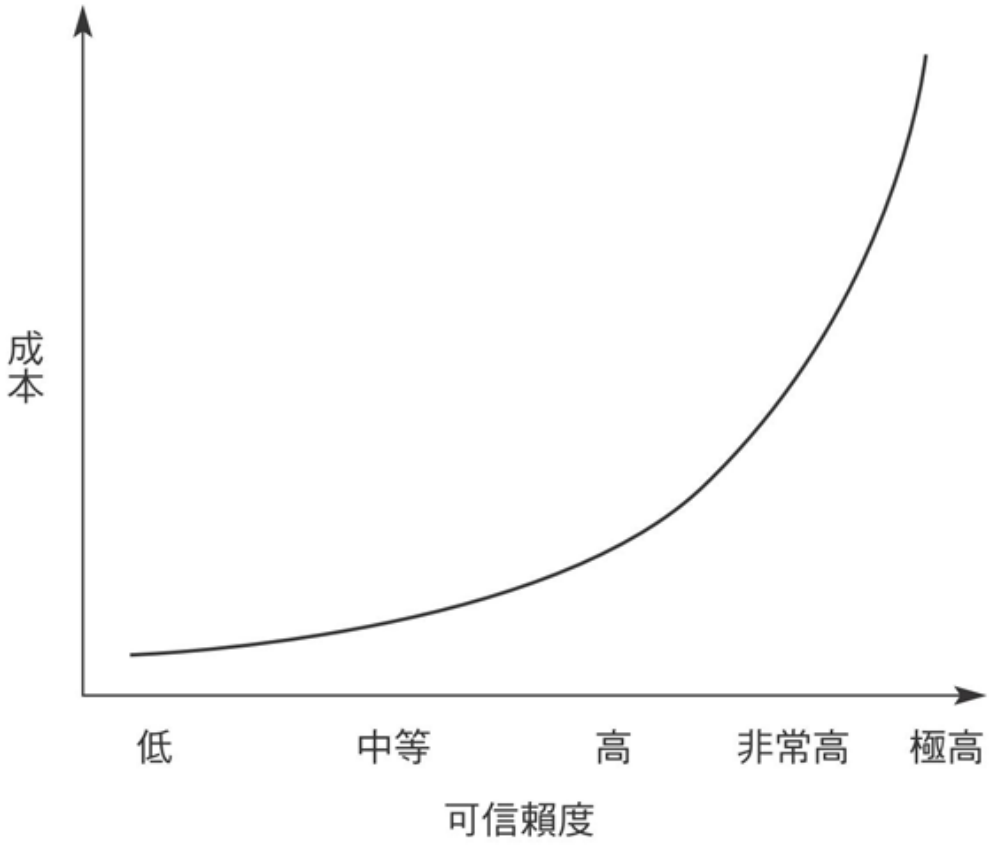


圖 10.2　成本與可信賴度曲線

10.2 社會化技術系統

- 軟體系統不是孤立的系統，而是某個包括人類、社會和組織目的的廣大系統中的一部分。
- 因此軟體工程也不是一個孤立的活動，而是更廣泛的系統工程 (第19章) 本身的一部分。
- 舉例而言，荒野測候系統的軟體負責控制氣象站裡的儀器設備。它會與其他軟體系統溝通，而且也是更廣大的本國和國際天氣預報系統的一部分。除了軟硬體之外，這些系統還包括如何預報氣象的程序，以及負責操作系統和分析輸出結果的人員。此外這系統還包括使用它來提供天氣預報給個人、政府和企業的機構。

10.2 社會化技術系統

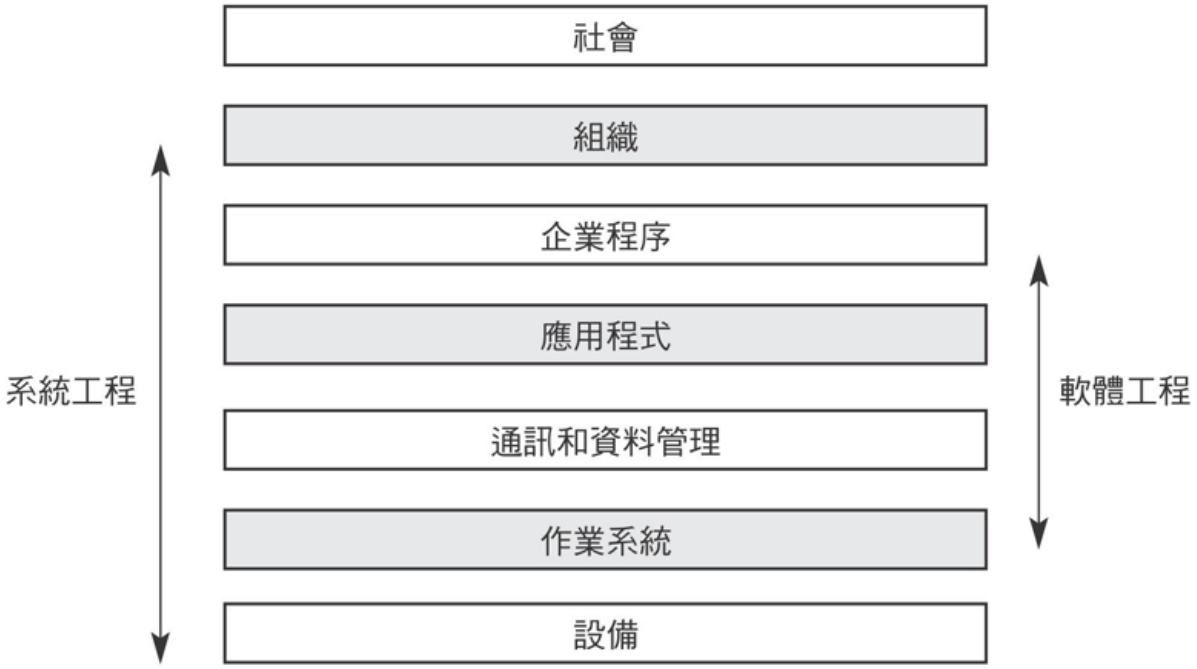


圖 10.3　社會化技術系統的堆疊

10.2 社會化技術系統

- 如圖10.3。這些層級便構成社會化技術系統的堆疊 (stack)：
  1. **設備層 (equipment layer)**：這一層是由硬體裝置所組成，其中有些可能是電腦。
  2. **作業系統層 (operating system layer)**：這一層與硬體溝通，並提供一組常用功能給系統中較高的軟體層級使用。
  3. **通訊和資料管理層 (communications and data management layer)**：並提供與其他更廣泛功能 (如存取遠端系統、存取系統資料庫等) 溝通的介面。
  4. **應用程式層 (application layer)**：這一層提供所需的應用程式特定功能。

10.2 社會化技術系統

5. **企業程序層 (business process layer)**：本層包含使用此軟體系統的企業程序。
6. **組織層 (organizational layer)**：這一層包含更高階的策略程序，還有使用系統時應該遵循的企業規則、政策和規範。
7. **社會層 (social layer)**：這一層定義監管這類系統運作的法律和社會規範。

10.2 社會化技術系統

- 本來原則上是希望大部分的溝通是在相鄰的層級之間，讓下層對上層隱藏自己的細節。但實際上並非永遠如此，在層級之間可能會發生非預期的溝通，結果導致系統整體的問題。
- 比如說，管理個人資訊存取的法律有了修改，這是來自社會層。而這導致產生新的組織程序和企業程序的變更。
- 在思考軟體保全性和可信賴度時，必須考慮系統整體，而不能只考慮到軟體。

10.2 社會化技術系統

- 儘可能確保軟體故障不會造成整體的系統故障，以確保下面兩點：
  1. 讓軟體故障儘量侷限在自己這個層級，而不要嚴重影響其他層級的運作。
  2. 要瞭解系統堆疊中其他哪些層級的缺失和故障可能會對軟體有影響。

10.2.1 監管和遵守

- 現在世界各地經濟方面的典型組織都差不多，就是由私人擁有的公司提供商品銷售並從中獲利。
- 為了確保民眾的安全，大多數政府會制訂標準要求私人公司遵守，證明他們的產品是合法安全的。
- 政府在各個領域都會制訂整套安全標準的規定和法規，也設置監管機構負責監督該領域的公司所提供的產品有符合規定。
- 開發安全關鍵系統的公司為了取得證明書，必須設計全面而大量的安全案例（第12章討論），證明該系統的確有遵守法規。
- 這些案例必須要能說服監管單位，該系統的確能安全運作。要開發像這樣的安全案例非常花錢。
- 而社會化技術系統的監管法規和遵守方式，必須適用在整個系統而不只是系統的軟體元件。

10.3 重複性與多樣性

- 重複性 (redundancy) 的意思是在系統中準備多餘的能力，萬一系統故障時就派上用場。
- 多樣性 (diversity) 是為系統準備不同類型的重複備用元件，降低它們同時故障的機率。
- 多樣性與重複性同樣也可用來設計可信賴的軟體開發程序。在可信賴的程序中的活動 (如軟體確認) 不會只靠單一程序或技術。
- 這些確認技術是互相搭配的。

10.3 重複性與多樣性

- 運用軟體重複性和多樣性可能會引進錯誤到軟體中。多樣性和重複性會讓系統變得更複雜，而且通常也會比較難瞭解。
- 複雜性越高代表程式設計人員就更容易犯錯，而且負責檢查系統的人也更難找到這些錯誤。
- 結果這導致有些人認為軟體最好要避免有重複性和多樣性。他們的觀點是設計軟體的方式越簡單越好，並且要採用非常嚴格的軟體驗證和確認程序。

10.3 重複性與多樣性

- 這兩種方式在商用的安全關鍵軟體系統都有人在使用。
- 例如空中巴士 (Airbus) A340的飛行控制硬體和軟體兩者都是多樣化且重複的。
- 而波音 (Boeing) 777上的飛行控制軟體則是在重複的硬體上執行，但每台電腦執行的是相同的軟體，該軟體已經被以極為嚴格的方式確認過。

10.4 可信賴的程序

- 可信賴的軟體程序 (dependable software process) 是指設計來生產出可信賴軟體的軟體程序。

程序特性	說明
可稽核的	程序應該要能讓參與者以外的人員所瞭解，由他們來檢查是否有遵循程序標準，並針對程序改進提出建議
多樣化	程序應該包含重複的和多樣化的驗證與確認活動
可產生說明文件	程序應該有事先定義的程序模型，它會定義程序中的活動，以及在這些活動進行期間會產生的說明文件
堅固的	程序應該能夠從個別的程序活動故障中回復
標準化	一組完整詳盡的軟體開發標準，定義要如何生產出軟體，以及應該要提供哪些說明文件

圖 10.4　可信賴程序的特性

10.4 可信賴的程序

- 系統開發人員通常會向主管機關提出他們所採用的程序模型，以及證明的確有遵循程序進行的證據。
- 主管機關也會審查是否所有參與者都有一致遵循這個程序。此程序必須是定義明確且可重複執行的：

1. 定義明確 (explicitly defined) 的程序是指具有既定的程序模型，可用來引領軟體生產過程。過程中一定要收集資料，證明程序模型中所有必要的步驟都有遵行。
2. 可重複執行 (repeatable) 的程序則是指程序不會因不同人而有不同的詮釋和判斷。無論參與程序的人為何，這些程序都應該能夠重複完成。

10.4 可信賴的程序

- 可信賴的程序中可能包含的活動如下：
  1. **需求審查 (requirements review)**：儘可能檢查需求是否完整而且一致。
  2. **需求管理 (requirements management)**：確保需求的變更是有管控的，同時被這個變更所影響的所有開發人員都瞭解此需求變更的影響。
  3. **正規化規格 (formal specification)**：建立軟體的數學模型並加以分析（關於正規化規格的優點在第10.5節有探討）。
  4. **系統塑模 (system modeling)**：將軟體設計明確的記錄成一組圖形模型，並將需求與這些模型之間的連結明確記錄在文件中。

10.4 可信賴的程序

5. **設計與程式碼檢查 (design and program inspection)**：由不同人員檢查系統是否有不一樣的描述。
6. **靜態分析 (static analysis)**：以自動化方式檢查程式的原始碼。
7. **測試規劃與管理 (test planning and management)**：測試程序本身也應該小心的管理，以確保這些測試有涵蓋系統需求，而且在測試程序中有被正確的運用。

10.4 可信賴的程序

- 事前的需求分析對於發現需求之間可能的衝突也很重要，這類衝突可能會影響系統的安全性和保全性。
- 這些做法與敏捷式開發的一般原則(共同發展出需求和文件極少化)會相衝突。
- 假設某程序內含如反覆式開發、測試優先的開發，以及使用者參與開發團隊等技術，也可以定義成一種敏捷式程序。
- 只要團隊有確實遵循該程序而且記錄下他們的行動，敏捷式技術也可以適用。

10.5 正規化方法和可信賴度

- 正規化方法是一種以數學為主的方法，在進行軟體開發時，先定義出軟體的正規化模型 (formal model)，接著再以正規方式分析這個模型，搜尋出錯誤和不一致的地方。
- 在電腦科學發展的早期階段，當時曾提出將數學的正規化方法應用在軟體開發上。它的觀念是正規化規格與程式兩者是可以獨立分別開發的。然後接著製作出數學證明來表達程式與它的規格是一致的。
- 程式證明現在都由自動化的大型定理證明 (theorem proving) 軟體來處理，所以較大的系統也能進行證明了。

10.5 正規化方法和可信賴度

- 精煉式 (refinement-based) 開發，這就不需要另外進行證明動作。做法是將系統的正規化規格，經過一連串正確性保留 (correctness-preserving) 的轉換步驟而產生出軟體。因為這些是可信賴的 (trusted) 轉換動作，所以產生出來的程式一定會與它的正規化規格一致。

10.5 正規化方法和可信賴度

1. **規格和設計方面的錯誤和遺漏：**在製作和分析軟體的正規化模型的過程中，可能會發現潛藏在軟體需求中的錯誤和遺漏。如果模型能從原始碼被自動化或系統化的產生出來，使用模型檢查來分析也許就會發現不希望發生的狀態，例如在並行 (concurrent) 系統中發現死結 (deadlock)。
2. **規格與程式之間不一致：**如果是採用精煉方法，就能避免發生因為開發人員犯錯而讓軟體與規格不一致的情況。程式證明步驟會發現程式與其規格不一致的地方。

10.5 正規化方法和可信賴度

- 優點如下：
  1. 在詳細開發正規化規格時，人員會逐漸深入和詳細瞭解系統規格。
  2. 因為規格是以具有正規化定義語意的語言表達的，因此可用自動化方式進行分析，找出不完整和不一致的地方。
  3. 假如使用如B方法這類的方法，就可以把正規化規格經由一連串的轉換步驟而轉換為程式，產生出來的結果程式因此保證符合它的規格。
  4. 可降低程式測試成本，因為程式已經與它的規格驗證過。

10.5 正規化方法和可信賴度

- 業界一直不願意採用正規化方法，原因有幾個：
  1. 問題負責人和領域專家無法瞭解正規化規格，所以不能檢查規格是否精確表達他們的需求。
  2. 建立正規化規格的成本很容易量化，但是使用它能節省多少成本卻很難估算，因此經理人不願意冒險採用。
  3. 大部分軟體工程師沒有接受過正規化規格語言的訓練，因此不願在開發程序中使用它。
  4. 目前的正規化方法很難拓展到非常大型的系統上使用。
  5. 正規化方法的工具支援很有限。它的市場規模太小，商業經營的公司不願投入。
  6. 正規化規格與主張少量逐步開發程式的敏捷式開發方法不相容。