功能正常，不代表結構正常：軟體表觀完好命題

v0.1 / 2026-08-01 / Neo.K

https://thisoneisneok.com/html/papers/ois-01.html

系列：《表觀完好系統：從軟體屎山、補償性完整到動態架構治理》

篇次：01 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

軟體工程長期存在一個容易被忽略的判斷錯位：當一個產品能正常啟動、主要功能可用、測試大致通過、使用者能持續操作，而且商業上甚至相當成功時，人們往往直覺地把這種「功能正常」延伸為「系統整體良好」。然而，功能層面的成功並不能充分推出架構、責任邊界、資料權限、演化能力與系統知識的完整。

本文提出「軟體表觀完好命題（Software Observable Integrity Proposition）」，將軟體的外部可觀察完整性與內部結構完整性明確區分。本文主張，一個系統完全可能同時具有很高的功能完整性與很低的結構完整性；軟體的可修補性、可包裝性、可重試性、可人工介入性與高度抽象性，使這種狀態不但可能，而且在長期演化系統中相當常見。

本文以 Big Ball of Mud、technical debt、software architecture erosion、Lehman 軟體演化研究與 Hyrum's Law 等既有軟體工程研究作為背景，並以 MSSP/FPL 從架構方法論走向可執行規格後暴露的實作問題作為一個研究動機。本文不主張「能運作的軟體都是壞的」，也不把所有結構問題統稱為技術債；相反地，本文建立多維完整性模型，區分功能完整性、結構完整性、演化完整性、知識完整性與外部可觀察完整性，為後續「補償性完好」「有效架構」「補償凝固」與「軟體結構動力學」提供理論地基。

關鍵詞：軟體架構、表觀完整性、結構完整性、技術債、架構侵蝕、Big Ball of Mud、MSSP、FPL、軟體演化

SECTION

1. 問題：一個能正常工作的系統，究竟證明了什麼？

對一般使用者而言，軟體系統的品質通常透過操作結果被理解。

應用程式能啟動、按鈕有反應、資料能保存、訂單能成立、付款能完成、網站不容易當機，這些都屬於真實而重要的品質訊號。工程團隊也會進一步檢查測試、效能、安全性與錯誤率。

問題在於：這些訊號主要回答的是：

> 系統目前是否能產生預期的可觀察結果？

它們不必然回答：

> 系統內部的責任、權限、依賴、資料所有權、失敗語義與演化邊界是否良好？

因此本文首先拒絕下列隱含推論：

> Functional Success
> ⇒
> Structural Correctness

更精確地說：

> I_o nRightarrow I_s

其中：

- I_o：Observable Integrity，外部可觀察完整性；
- I_s：Structural Integrity，內部結構完整性。

一個系統「看起來正常」，只能證明某些輸入經過現有路徑後，能在某些條件下產生可接受輸出。

它不能單獨證明：

- 模組責任清楚；
- 依賴方向合理；
- 資料所有權明確；
- 權限沒有穿透；
- 失敗處理完整；
- 不存在隱性契約；
- 新需求可以低成本加入；
- 團隊成員能理解真正的系統；
- 系統在下一次環境改變後仍然可維護；
- 當前成功不是依賴大量人工或技術補償。

這就是「表觀完好」問題的起點。

SECTION

2. 軟體為什麼特別容易形成表觀完好？

物理產品的某些結構缺陷會迅速轉化成明顯失敗。橋樑若失去關鍵承載結構，可能直接崩塌；機械零件尺寸錯誤，也常會以摩擦、破裂或無法裝配的形式顯現。

軟體不同。

軟體具有高度的邏輯可塑性。當原本模型不完整時，開發者往往可以再加入一個條件、一層轉換、一個重試、一個快取、一個例外表、一個手動維運流程，使原本不能成立的路徑重新得到可接受結果。

例如：

原始設計不足
→ 加入例外判斷
→ 出現新邊界案例
→ 再加入特殊路徑
→ 資料不一致
→ 增加 reconciliation
→ 外部服務不穩

→ 增加 retry / fallback
→ 自動流程仍有缺口
→ 加入人工處理

因此，軟體存在一個其他工程系統較不明顯的特性：

> 局部結構問題可以長期被其他軟體結構與人類操作遮蔽。

這使得「目前沒有失敗」和「內部沒有重大問題」之間產生巨大的認知距離。

SECTION

3. Big Ball of Mud：混亂不等於不能生存

Foote 與 Yoder 在 Big Ball of Mud 中描述了一類缺乏清楚高階架構、由權宜修改與反覆修補塑形的系統。他們真正重要的洞察不是單純嘲笑這類架構，而是指出：這種系統十分普遍，而且具有不可否認的實用性。

換句話說，軟體工程很早就遇到了以下現象：

> Architectural Disorder
> not⇒
> Immediate Operational Failure

甚至：

> Messiness
> ≠
> Non-viability

這是一個重要轉折。

如果混亂系統必然立刻失敗，那麼 Big Ball of Mud 不會成為長期存在的工程現象。真正值得研究的是：它為什麼能活？

答案之一，是軟體的局部修補能力。

另一個答案，是商業與工程決策的時間尺度不同。短期的交付、修復與市場需求可以有很高價值，而架構收益通常在較長時間後才顯現。因此，某些現在看來不漂亮的結構，在當時可能是合理甚至必要的妥協。

本文因此不把「表觀完好系統」等同於「低品質系統」。

我們關心的是：

> 外部成功與內部完整之間可能存在多大的差距，以及這個差距如何被維持。

SECTION

4. Technical Debt：現在可用，不代表未來便宜

SEI 對 technical debt 的經典工程化定義指出，一種設計或建造方式可以在短期帶來便利，但使未來完成相同工作需要付出更高成本。

這再次說明：

> Short-Term Effectiveness
> ≠
> Long-Term Structural Economy

一個選擇可以同時滿足：

> $U_{(now)} > 0$

以及：

> $C_{(future)} \uparrow$

其中 $U_{(now)}$ 表示當下效用，$C_{(future)}$ 表示未來修改、維護或演化成本。

但本文需要特別區分：

> 不是所有表觀完好系統的結構問題都叫 technical debt。

因為有些問題不是故意以未來成本換取現在速度，而是：

- 對問題域理解不足；
- 組織邊界改變；
- 外部 API 行為改變；

- 使用者形成隱性依賴；
- 原本合理的設計被新需求穿透；
- 人員更替造成知識遺失；
- 架構與實作逐步漂移；
- 系統規模跨過原本的設計尺度。

因此 technical debt 只是更大「結構負擔」集合中的一部分。

SECTION

5. Architecture Erosion：原本設計過，也仍然可能變差

這一點對本文尤其重要。

表觀完好命題並不只針對「一開始就沒有設計」的軟體。

Architecture erosion 的研究指出，軟體在演化過程中可能逐漸偏離原始架構；這種侵蝕不只表現為靜態結構違反，也會影響品質、維護與後續演化，而且成因同時包括技術因素與非技術因素。

因此即使存在：

> A□ = well-designed architecture

隨時間演化：

> A□ → A□ → A□ → ... → A□

仍然可能出現：

> D(A□, A_(intended)) uparrow

其中 D 表示實際架構與宣告／預期架構之間的偏離程度。

這與 MSSP 實作帶來的觀察高度相關。

MSSP 原本就不是完全無設計的架構。其早期版本已經明確區分：

- FMS：系統元資料與「系統是什麼、為何存在」；
- SMS：不可或缺的核心功能；
- TMS：可插拔的擴展功能；

  並進一步嘗試將架構圖、DSL、Linter、依賴驗證與專案生成結合。

  然而，當這些概念真正被放進可執行規格時，新的問題開始出現：

- SMS/TMS 可能不是永恆固定身份；
- 不同上下文會改變「核心」的判定；
- import dependency 不等於完整 dependency；
- 資料權限不能只由資料夾位置決定；
- TMS 可能需要自己的持久化狀態；
- 宣告架構與實際運行架構可能漂移；
- 一條在抽象模型上漂亮的硬規則，在真實工程裡可能需要例外與信心等級。

  這並不直接證明 MSSP 是錯的。

  相反地，它提供了一個重要研究案例：

> 即使一套架構方法從一開始就試圖顯式管理結構，當它被推進到更大型、更動態、更真實的工程環境時，仍會暴露新的結構不完備。

那麼對完全沒有明確架構治理的系統而言，隱藏問題很可能更多，而不是更少。

SECTION

6. Hyrum's Law：沒有寫進契約的行為，也可能變成契約

結構與表觀之間另一個重要裂縫來自 API 使用。

Hyrum's Law 的核心觀察是：當 API 使用者數量足夠大時，系統所有可觀察行為都可能被某個使用者依賴。

這表示：

> C_(formal)
> ⊆
> C_(effective)

其中：

- C_(formal)：正式宣告契約；
- C_(effective)：實際被外部世界依賴的有效契約。

  這種差異會產生非常有趣的情況：

  某個行為可能原本被開發者認為是：

  implementation detail
  但當外部系統長期依賴它後，它事實上變成：

  compatibility constraint
  因此，所謂「修正一個不漂亮的行為」甚至可能破壞真正的系統。

  這意味著軟體結構不能只由原始碼與正式規格理解。

  真正的系統還受到：

- 使用者習慣；
- 外部依賴；
- 歷史輸出；
- 時序；
- 效能特徵；
- 非正式操作流程；

  等因素約束。

  這將在後續「有效架構」與「補償凝固」中進一步展開。

SECTION

7. 軟體完整性不是一維標量

為避免「好軟體／壞軟體」這種過度簡化判斷，本文提出一個最小多維完整性模型。

令一個軟體系統在時間 t 的完整性狀態為：

> I(t)
> =
> [
> I_f(t),
> I_s(t),
> I_e(t),
> I□(t),
> I_o(t)
>]

其中：

7.1 功能完整性

> I_f
> =
> Functional Integrity

表示系統是否能完成承諾功能與關鍵使用流程。

7.2 結構完整性

> I_s
> =
> Structural Integrity

表示模組邊界、依賴、權限、資料所有權、狀態與失敗語義是否清楚並保持一致。

7.3 演化完整性

> I_e
> =
> Evolutionary Integrity

表示系統是否能在需求、規模、外部環境與技術棧改變時，以可控制成本進行修改。

7.4 知識完整性

> I□
> =
> Knowledge Integrity

表示系統真正需要的知識是否已被外部化、文件化、形式化或可被重新推導，而不是只存在少數工程師的記憶中。

7.5 表觀完整性

> I_o
> =
> Observable Integrity

表示對使用者、管理者或外部監測而言，系統看起來有多正常、穩定與完整。

因此，一個系統完全可能存在：

> $I_o \approx 1$

同時：

> $I_s \ll 1$

甚至：

$$I_e, I\boxtimes \ll I_o$$

這就是本文所稱的「表觀完好」。

SECTION

8. 表觀完好命題

本文提出以下命題。

SECTION

命題 1：軟體表觀完好命題

對一個軟體系統 S 而言，高可觀察完整性不足以推出高結構完整性：

$$I_o(S) nRightarrow I_s(S)$$

同樣地，高功能完整性也不足以推出高演化完整性與知識完整性：

$$I_f(S)
nRightarrow
I_e(S)$$

$$I_f(S)
nRightarrow
I\boxtimes(S)$$

直觀解釋

「今天能完成任務」描述的是現在某些路徑的結果。

「系統結構良好」描述的是更廣泛的內部關係與未來可變性。

兩者相關，但不是邏輯等價。

SECTION

9. 為什麼這不是一句普通的「能跑不代表好程式」？

這個命題如果只停在「能跑不代表程式碼漂亮」，價值有限。

本文真正想建立的是下一步：

> 如果 I_o 很高，而 I_s 不高，那麼兩者之間的差距究竟由什麼維持？

定義：

> G_I
> =
> I_o-I_s

其中 G_I 可暫稱為「表觀—結構差距」。

注意：這只是一個概念性表示，不主張目前已經存在可直接用單一數值精確測量兩種完整性的通用方法。

但它讓我們能提出下一個研究問題：

若：

> G_I ≫ 0

那麼系統必然存在某些額外機制，使內部結構不足仍能得到外部正常結果。

可能來源包括：

- retry；
- fallback；
- adapter；
- wrapper；
- migration；

- monitoring；
- feature flag；

  -人工修復；
- 維運 SOP；
- 未文件化知識；
- 相容性 shim；
- 資料 reconciliation；
- 客服處理；
- 外部平台能力；
- 開發者默契。

  下一篇將把這整組機制正式稱為：

SECTION

補償性完好（Compensated Integrity）

也就是：

> 一個系統的可觀察完整性，不一定完全來自其內部原生結構；它也可能部分來自技術、人類、操作與相容機制的共同補償。

SECTION

10. 軟體成功甚至可能加劇這個問題

Lehman 對軟體演化的長期研究提醒我們：與真實世界互動的 E-type 系統需要持續演化，而複雜度若不被主動控制，會隨演化增加。

因此一個成功產品可能經歷：

> Success
> →
> More Users
> →
> More Requirements
> →
> More Changes
> →
> More Historical Constraints

如果架構治理能力沒有同步提升：

> →
> Structural Burden

所以：

> Commercial Success
> nRightarrow
> Architectural Elegance

甚至在某些條件下：

> Commercial Success
> →
> Higher Evolutionary Pressure

這可以解釋為什麼某些非常成功、非常重要的軟體，內部結構反而比小型展示專案更不漂亮。

不是因為成功必然導致爛架構，而是因為成功系統承受的歷史、相容、組織與規模壓力更大。

SECTION

11. MSSP 對這個命題的特殊意義

MSSP 的價值恰好在於它不是完全沒有設計的架構。

它一開始就嘗試回答：

- 系統是什麼；
- 哪些是核心；
- 哪些是可插拔能力；
- 模組之間如何依賴；
- 架構能否被視覺化；
- 規格能否轉為機器可讀結構；
- 實作能否被自動驗證。

因此 MSSP 實作暴露出的問題，反而使原始命題變得更有力量：

> 如果一套明確考慮過結構、分層、可視化與依賴治理的方法，在進入真實實作後仍然需要持續修正，那麼「沒有暴露問題」不能被當成「沒有問題」的證據。

這也是為什麼未來 MSSP 不應只走向「更多硬性規則」。

它更可能需要：

> Static Classification
> →
> Dynamic Observation

以及：

> Rule Enforcement
> →
> Evidence-Assisted Judgment

但這屬於系列後半段的問題。

本文暫時只保留一個結論：

> 架構設計不是一次性保證，而是一個持續被現實反駁、校正與重新理解的過程。

SECTION

12. 邊界與限制

本文命題需要避免四種過度延伸。

SECTION

12.1 表觀完好不等於欺騙

一個系統外部功能正常但內部結構不佳，不代表開發者故意掩飾問題。

很多差距來自合理歷史演化。

SECTION

12.2 結構不完美不等於失敗

對短生命週期原型、小型腳本、探索性系統而言，過度架構反而可能提高成本。

Big Ball of Mud 相關討論早已指出，過早架構同樣可能成為限制。

SECTION

12.3 不能用單一指標宣告「架構健康」

本文的 I_f,I_s,I_e,I□,I_o 首先是理論分解，不是已驗證的標準化量表。

不同系統的測量方式必須依情境設計。

SECTION

12.4 不能把所有結構問題都叫 technical debt

有些問題是債，有些是侵蝕，有些是歷史殘留，有些是隱性契約，有些只是問題域本身的必要複雜度。

後續篇章會逐步拆開。

SECTION

13. 結論

本文從一個非常簡單的工程直覺開始：

> 程式能跑，不代表程式結構良好。

但將它推進一步後，可以得到更重要的命題：

> 【I_o nRightarrow I_s】

外部可觀察完整性不能充分推出內部結構完整性。

軟體的特殊性在於，它可以透過局部修補、抽象層、相容層、維運流程與人類介入，在相當長的時間內維持高功能完整性，即使其內部結構已經存在顯著問題。

Big Ball of Mud 告訴我們，混亂架構仍可能有效；technical debt 告訴我們，短期便利可以轉化為未來成本；architecture erosion 告訴我們，即使原本有設計，架構仍可能在演化中偏離；Hyrum's Law 告訴我們，沒有被正式承諾的行為也可能變成真實契約；Lehman 的軟體演化研究則提醒我們，長期存活本身意味著持續改變與複雜度治理。

因此，下一個真正的問題不再是：

> 「為什麼壞軟體還能跑？」

而是：

> 「當結構完整性不足時，是哪些額外機制在替系統承擔完整性？」

這將導向本系列第二篇：

〈補償性完好：為什麼不完整的軟體仍然可以正常運作〉。

SECTION

參考文獻

- Foote, B., & Yoder, J. (1997/1999). Big Ball of Mud. Pattern Languages of Program Design 4 / PLoP '97. https://www.laputan.org/mud/
- Software Engineering Institute, Carnegie Mellon University. (2016). Managing Technical Debt in Complex Software Systems. https://www.sei.cmu.edu/library/managing-technical-debt-in-complex-software-systems/
- Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2022). Understanding software architecture erosion: A systematic mapping study. Journal of Software: Evolution and Process, 34(3), e2423. https://doi.org/10.1002/smr.2423
- Lehman, M. M., & Ramil, J. F. (2003). Software evolution—Background, theory, practice. Information Processing Letters, 88(1–2), 33–44. https://doi.org/10.1016/S0020-0190(03)00382-X
- Hyrum Wright. Hyrum's Law. https://www.hyrumslaw.com/
- Harrand, N., Benelallam, A., Soto-Valero, C., Bettega, F., Barais, O., & Baudry, B. (2019). API Beauty is in the eye of the Clients: 2.2 Million Maven Dependencies reveal the Spectrum of Client-API Usages. arXiv:1908.09757.
- Neo K. (2025). 資料夾程式語言（Folder Programming Language）：從認知架構到可執行規範的範式革命. EveMissLab. （本系列後續將 FPL 重新定位為 Formal Project Language。）