屎山為什麼沒有死？Big Ball of Mud 作為生存架構

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

https://thisoneisneok.com/html/papers/ois-04.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：04 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

「屎山程式碼」通常被理解為一種應當被避免、重寫或重構的失敗狀態。然而，若這類系統真的只是純粹失敗，它們理應很快被淘汰；現實卻恰恰相反：大量缺乏清楚高階架構、依賴關係混亂、充滿特殊分支與歷史殘留的系統，仍然可以運行多年乃至數十年，持續支撐重要商業與組織活動。

Foote 與 Yoder 在 Big Ball of Mud 中早已注意到這個悖論。他們將 Big Ball of Mud 描述為由權宜而非整體設計主導的常見架構，並明確指出其長期普及不能只用「大家不在乎架構」來解釋；相反地，必須理解這種模式為何具有不可否認的有效性。其相關模式包括 Throwaway Code、Piecemeal Growth、Keep It Working、Shearing Layers、Sweeping It Under the Rug 與 Reconstruction。

本文在前述「表觀完好」「補償性完好」與「有效架構」基礎上，提出：Big Ball of Mud 應被理解為一類局部適應能力強、全局結構整合能力弱，但具有高度生存性的軟體架構狀態。它的生存能力主要來自局部可修補性、低局部變更門檻、歷史知識沉積、成熟介面與運行慣例、補償機制、組織熟悉度，以及「保持系統繼續運作」所帶來的選擇壓力。

本文區分「結構品質」與「生存適應度」，提出：

> $Q_s \neq F_v$

其中 Q_s 表示 Structural Quality，F_v 表示 Viability/Fitness。由此，一個系統可以具有較低的全局結構品質，同時具有相當高的現實生存適應度。本文進一步提出「局部適應—全局侵蝕張力」「生存性架構」「熟悉度資本」「重寫風險」與「可替換性悖論」等概念，並指出：對 Big Ball of Mud 的合理治理目標不是追求一次性的架構純化，而是降低不可見依賴、補償負載與演化風險，同時保留其已被現實反覆驗證的生存能力。

關鍵詞：Big Ball of Mud、屎山程式碼、軟體演化、legacy system、局部適應、架構侵蝕、熟悉度資本、補償性完好、MSSP、軟體生存性

SECTION

1. 如果它真的那麼爛，為什麼它還活著？

對「屎山」最直覺的工程態度通常是：

> 這東西寫得這麼亂，早晚會爆炸。

但一個令人不舒服的現實是：

有些系統已經被這樣形容十年、二十年，甚至更久，而它仍然：

- 每天處理交易；
- 持續增加新功能；
- 維持客戶；
- 通過稽核；
- 支撐企業核心流程；
- 每年照樣被維護與部署。

因此，如果：

> Bad Structure
> ⇒
> Immediate Death

那麼我們不應該看到如此多長壽的 legacy system 與 Big Ball of Mud。

現實更接近：

> Bad Structure
> not⇒
> Immediate Death

甚至：

> Messiness
> ≠
> Non-viability

這不是為屎山辯護。

它只是要求我們回答一個比「這程式很醜」更困難的問題：

> 它究竟靠什麼活下來？

SECTION

2. Big Ball of Mud 原本就不是純粹的罵人詞

Foote 與 Yoder 在 Big Ball of Mud 中描述的 Big Ball of Mud，是一種隨意、權宜、缺乏清楚整體結構的系統。

重要的不是這個名稱有多難聽，而是作者的研究問題。

他們指出，這種架構實際上極為普遍，其持久流行不能簡單理解成所有開發者都不在乎架構。

相反地，他們直接追問：

> 這種方式到底做對了什麼？

其完整模式群包括：

- Big Ball of Mud；

- Throwaway Code；
- Piecemeal Growth；
- Keep It Working；
- Shearing Layers；
- Sweeping It Under the Rug；
- Reconstruction。

這是一個非常重要的框架。

因為它沒有把 Big Ball of Mud 當成靜態終點，而是把它看成軟體在真實壓力下演化時可能反覆進入的狀態。

SECTION

3. 第一種生存力：Piecemeal Growth

大型系統很少按照一個完整藍圖一次建成。

更常見的是：

功能 A
→ 加需求 B
→ 修問題 C
→ 接第三方 D
→ 增加例外 E
→ 支援舊客戶 F
→ 再做功能 G

每一次改動都可能局部合理。

因此對第 i 次變更，可以有：

> $\Delta U_i > 0$

其中 ΔU_i 表示該變更帶來的局部效用。

但是長期累積：

> Σ ⊠ Δ U⊠ > 0

並不保證：

> Δ Q_s > 0

甚至可能：

> Δ Q_s < 0

亦即：

> 每一步都解決了當下問題，但整體結構反而越來越難理解。

這就是本文所稱：

SECTION

局部適應—全局侵蝕張力

SECTION

Local Adaptation–Global Erosion Tension

形式上：

> A_(local)uparrow
> ∧
> Q_(global)downarrow

兩者可以同時成立。

SECTION

4. Lehman：不改會死，改了又會變複雜

Lehman 的軟體演化研究提供了一個幾乎完美的背景。

對與真實世界持續互動的 E-type system 而言：

- 系統必須持續改變，否則會逐漸不能滿足環境；
- 系統改變時，複雜度傾向增加，除非額外投入工作控制它。

因此存在一個基本張力：

> No Change
> →
> Environmental Misfit

但：

> Continuous Change
> →
> Complexity Pressure

所以軟體不是在：

> 「乾淨」與「髒」之間自由選擇。

而更像是在：

> Adaptation Pressure
> ↔
> Structural Discipline

之間持續取平衡。

如果市場壓力、deadline、使用者需求與外部系統變更速度高於架構治理能力：

> V_(change)
> >
> V_(governance)

那麼即使每個工程師都知道理想架構應該更乾淨，系統仍然可能逐步泥化。

SECTION

5. 第二種生存力：Keep It Working

Big Ball of Mud 的另一個核心模式是：

SECTION

Keep It Working

對一個已經承載真實使用者與真實交易的系統而言：

> Continuity

本身就是高價值屬性。

假設存在兩個方案：

方案 A

架構更漂亮，但需要六個月全面重構。

方案 B

只改局部，架構更醜，但明天可以上線。

在理想工程環境中，可能偏好 A。

但如果：

- 競爭者正在上新功能；
- 法規下週生效；
- 客戶今天就需要；
- 核心交易不能停；

- 公司只有三名工程師；
- 系統缺乏完整測試；

  那麼 B 很可能是現實最優解。

  因此「Keep It Working」並不等於工程師不知道好架構。

  它代表：

> $$V_{(continuity)} > V_{(architectural\ purity)}$$

在某些時間窗口內成立。

SECTION

6. 生存適應度與結構品質不是同一個軸

因此本文提出一個重要區分：

> $$Q_s = \text{Structural Quality}$$

與：

> $$F_v = \text{Viability / Survival Fitness}$$

不能視為同一指標。

一個系統可能：

> $$Q_s \uparrow,\ F_v \uparrow$$

這當然最好。

但也可能：

> Q_s downarrow, F_v uparrow

例如：

- 大量歷史例外讓架構混亂；
- 但它能處理二十年累積的真實業務；
- 新系統架構漂亮；
- 卻還沒學會所有例外。

因此：

> 【Q_s ≠ F_v】

更進一步：

> 【Architectural Elegance
> nRightarrow
> Operational Fitness】

這並不是說漂亮架構沒有價值。

而是說：

> 生存能力需要經過現實環境測試，而不是只由結構形式決定。

SECTION

7. 第三種生存力：它已經累積了「真實世界資料」

一個活了十五年的系統，不只有程式碼。

它還累積了：

- 真實錯誤案例；
- 例外交易；
- 特殊客戶；
- 歷史資料格式；
- regulatory edge cases；
- migration path；
- 部署知識；
- incident response；
- 相容性規則；
- 外部系統怪癖；
- 人類操作知識。

  因此它實際包含：

> K_(embedded)

即：

SECTION

Embedded Operational Knowledge

很多這些知識不是以「文件」形式存在，而是嵌在：

- if/else；
- timeout；
- magic number；
- strange SQL；

- duplicated field；
- old adapter；
- cron；
- retry；
- comment；
- runbook；
- human memory。

  所以某段看起來極醜的程式：

  ```
  if old_customer_type == 7:
    ...
  ```

  可能不是純粹垃圾。

  它也可能是：

> 2013 年某個已經沒人記得原因、但今天仍有客戶依賴的商業規則。

  問題不是這個 if 很漂亮。

  問題是：

> Ugly Code

可能同時攜帶：

> Historical Knowledge

SECTION

8. 熟悉度資本：團隊已經學會如何和它共存

長壽系統還會形成另一種資本：

SECTION

熟悉度資本

SECTION

Familiarity Capital

令：

> K_f

表示團隊對現有系統累積的熟悉度。

它包括：

- 哪裡最容易壞；
- 哪些 warning 可以忽略；
- 哪些檔案不能一起改；
- 哪個 migration 要多跑一次；
- 哪些客戶還在舊版；
- 哪個 service 重啟順序固定；
- 哪段程式碼看似多餘但不能刪。

  這些知識未必是良好工程的證明。

  但它有現實價值。

  因此，一個 legacy system 的總資產可能是：

> $$V_L = V_{(software)} +$$

當人們說：

> 「全部重寫比較快。」

常常只比較：

> C_(new code)

卻忽略必須重新獲得：

> K_(embedded)
> +
> K_f

的成本。

SECTION

9. 第四種生存力：Wrapping 比 Replacement 便宜

SEI 對 legacy-system evolution 的研究很早就指出，interface、wrapping 與 network technology 可以用來繼續利用既有軟體資產，而不是直接丟棄並從頭重建。

這是一個非常務實的選擇。

假設核心系統：

> L

很難改。

可以新增：

> W(L)

作為 wrapper。

於是：

> New World
> →
> W
> →
> L

短期看來，新系統避免直接進入泥巴核心。

代價則是：

> C_(wrapper)
> +
> C_(translation)
> +
> C_(compatibility)

增加。

但只要：

> C_(wrap)
> <
> C_(replace)

組織就有經濟理由繼續包。

於是：

legacy core
→ adapter
→ API
→ compatibility layer
→ new service
→ another adapter
逐漸形成新的有效架構。

這就是為什麼 Big Ball of Mud 可以被「包起來」繼續活。

SECTION

10. Shearing Layers：不是所有部分都用同一速度變化

Foote 與 Yoder 使用 Shearing Layers 來描述不同部分具有不同變動速率。

這個概念對 legacy system 很重要。

例如：

> v_{UI}
> $\gg$
> $v_{core\ ledger}$

前端每幾個月改一次；

帳務核心可能十年都不能大改。

因此，系統很自然會形成：

fast-changing layer
↓
adapter
↓
slow-changing layer

這並不一定是壞設計。

甚至可能是合理隔離。

但是如果這些速度差沒有被正式管理，就會形成大量：

- translation；
- compatibility；
- duplicate model；
- synchronization；
- bridge。

因此 Big Ball of Mud 有時不是單純「大家亂寫」。

而是：

> 不同演化速度的結構長期剪切後形成的沉積物。

SECTION

11. 第五種生存力：局部修改半徑小

乾淨架構有時要求：

> 修改核心概念時，同時修正所有相關抽象。

而泥巴架構很常採取另一種策略：

> 不碰核心，只在旁邊再補一塊。

即：

$$\Delta S_{(local)}$$

很小。

這會降低：

$$R_{(immediate)}$$

即當下修改風險。

所以短期：

$$\text{Patch} \rightarrow \text{Low Blast Radius}$$

是很誘人的。

但長期：

> Σ Patch
> →
> High Structural Entanglement

於是形成：

SECTION

Patch Accumulation Trap

每一次 patch 都因為害怕大修改而合理；

而 patch 越多，大修改越危險；

因此：

> More Patches
> →
> Higher Rewrite Risk
> →
> More Patches

形成正回饋。

SECTION

12. 第六種生存力：補償性完好

第二篇已經建立：

> I_o(t)
> =
> Φ
> (
> I_s(t),
> C(t),

Big Ball of Mud 往往具有非常大的：

> C

包括：

- 人工操作；
- wrapper；
- retry；
- workaround；
- compatibility；
- nightly job；
- manual repair；
- monitoring；
- support knowledge。

因此其存活可能不是因為：

> I_s

很高。

而是：

> C

極其成熟。

換句話說：

> 它不是沒有秩序，而是大量秩序沒有存在於你期待的位置。

有些秩序存在於：

- 操作手冊；
- 人；
- 部署程序；
- 客戶習慣；
- API 相容；
- 每晚批次；
- 監控告警；
- 事故應變。

這也是為什麼只掃 source code 可能嚴重低估系統的真正結構。

SECTION

13. 第七種生存力：選擇壓力保留「能活的醜東西」

這裡可以借用一個演化式的直覺，但不把軟體擬生物化。

長期運行系統中的程式片段不是隨機保留。

它們經過：

- 使用者；
- production；
- incident；
- patch；
- regression；
- deployment；

反覆篩選。

某些特別糟糕的部分會被修掉。

某些雖然醜、但非常可靠的部分反而留下。

所以存活很久的 codebase 具有：

SECTION

Survivorship Filtering

這不能推出：

> 老程式碼一定很好。

但可以推出：

> 一段活了十五年仍然承擔關鍵工作、且經歷無數部署的程式，不能只因外觀難看就假定它毫無價值。
> 它至少證明：

> $F_v > 0$

而且可能很高。

SECTION

14. 重寫悖論：新系統結構更漂亮，但知識更少

假設：

> S_{old}

很亂。

團隊建立：

> S_{new}

新系統擁有：

> Q_s(S_(new))
> >
> Q_s(S_(old))

這很好。

但一開始很可能：

> K_(embedded)(S_(new))
> <
> K_(embedded)(S_(old))

所以：

> F_v(S_(new))

不一定立刻更高。

這就是：

SECTION

Rewrite Paradox

「重新寫」不是把同一系統用更好的程式碼重寫。

實際上往往是：

> 重新發現舊系統二十年來已經學會的所有世界規則。

如果漏掉：

- 少見 exception；
- legacy customer；
- 奇怪 settlement；

- timing behavior；
- hidden API contract；

新系統就會在上線後重新「學習」。

而所謂學習，通常就是：

bug
→ incident
→ patch
→ new exception
新系統可能開始重新走向泥巴。

SECTION

15. 所以 Reconstruction 為什麼很難？

Big Ball of Mud 把 Reconstruction 也列為模式。

這點非常重要。

因為泥巴不是永遠不能重建。

而是重建需要某些條件。

至少要知道：

> A_(effective)

而不只是：

> A_(declared)

還要辨認：

> C

補償機制。

以及：

> K_(embedded)

歷史知識。

如果不知道這三者，就可能：

> Reconstruction
> →
> Lost Capability

因此安全重建不是：

```
old code
→ delete
→ clean new code
```

而應該更接近：

> Recover
> →
> Model
> →
> Verify
> →
> Migrate
> →
> Observe
> →
> Retire

SECTION

16. Big Ball of Mud 作為「生存架構」

本文因此提出：

SECTION

生存架構

SECTION

Survival Architecture

它不是一種推薦設計模式。

定義為：

> 一個系統雖然缺乏高度一致的全局架構，但透過局部適應、歷史知識、補償機制、相容層與組織熟悉度，持續滿足足夠多現實條件而得以長期存活的架構狀態。

其生存適應度可概念化為：

$$
F_v = f(A_l, C, K_e, K_f, R_c, E)
$$

其中：

- A_l：local adaptability；
- C：compensation capacity；
- K_e：embedded knowledge；
- K_f：familiarity capital；

- R_c：continuity／recovery capacity；
- E：environment fit。

因此：

> Q_s downarrow

不必然立刻造成：

> F_v downarrow

但通常會提高未來改變的成本與風險。

SECTION

17. 生存架構的代價

Big Ball of Mud 能活，不代表免費。

它的代價通常表現在：

SECTION

17.1 Change Cost

> C_(Δ)
> uparrow

修改一個功能需要理解越來越多未知關係。

SECTION

17.2 Cognitive Load

> L_(cognitive)
> uparrow

新成員難以建立完整模型。

SECTION

17.3 Compensation Load

> L_c
> uparrow

越來越多 wrapper、manual process 與 exception。

SECTION

17.4 Hidden Coupling

> D_(hidden)
> uparrow

依賴不再只存在於程式碼。

SECTION

17.5 Rewrite Risk

> R_(rewrite)
> uparrow

因為沒有人確定全部行為。

所以：

> 屎山真正的問題不是「不能工作」，而是它把未來自由度逐漸換成現在連續性。

SECTION

18. 生存性—可演化性邊界

因此可以把系統分成四個粗略區域：

Structural Quality	Survival Fitness	狀態
高	高	健康可演化系統
高	低	理論漂亮但現實不適配
低	高	生存型泥巴／成熟 legacy
低	低	接近失敗或等待淘汰

其中最容易被工程敘事誤判的是：

> $Q_s \downarrow, F_v \uparrow$

因為我們看到結構很糟，就容易推論它沒有價值。

但真正合理的治理策略應該是：

> 保留 F_v，提高 Q_s。
>
> 而不是：
>
> 為了提高 Q_s，先把 F_v 摧毀。

SECTION

19. 對 Dynamic MSSP 的啟示：先判定「它為何活著」

這直接影響 Dynamic MSSP。

傳統架構檢查器看到：

cyclic dependency

可能直接給：

ERROR
Dynamic MSSP 更應該問：

- 這個 cycle 是否真的參與 runtime？
- 哪些功能依賴它？
- 它是不是 historical compatibility？
- 是否存在補償？
- 移除後 blast radius 多大？
- 它是偶然耦合，還是已經凝固成 effective contract？
- 是否存在可替換路徑？

  因此：

> Violation
> not⇒
> Immediate Repair

而是：

> Violation
> →
> Role Analysis
> →
> Survival Function Analysis
> →
> Migration Decision

這是 AI 比傳統 Linter 更可能發揮價值的地方。

SECTION

20. 「醜」不是有效分類

軟體工程常使用：

- spaghetti；
- hack；
- ugly；
- legacy；
- shit code；

  這些詞很有溝通效率。

  但研究上太粗。

  同一段 ugly code 可能是：

- 無意義殘留；
- 暫時 workaround；
- historical compatibility；
- performance optimization；
- regulatory edge case；
- human compensation anchor；
- 已凝固的 effective contract。

  因此 Dynamic MSSP 最重要的不是：

> AI 幫我們找醜程式碼。

  而是：

> AI 幫我們判斷醜程式碼到底在系統裡扮演什麼角色。

這比 code smell classification 更接近真正的架構治理。

SECTION

21. 命題 4：生存架構命題

本文提出系列第四個命題：

SECTION

生存架構命題

對長期運行軟體系統 S 而言：

> 【Q_s(S)
> nRightarrow
> F_v(S)】

其中：

- Q_s：Structural Quality；
- F_v：Survival Fitness。

  較低的結構品質不必然導致較低的短期或中期生存適應度。

  Big Ball of Mud 可以透過：

> 【Local Adaptation
> +
> Compensation
> +
> Embedded Knowledge
> +
> Familiarity
> +
> Continuity】

維持相當高的：

> F_v

但這通常以更高的：

> C_(Δ)
> +
> L_c
> +
> D_(hidden)
> +
> R_(rewrite)

為代價。

因此它是一種：

> 能活，但越來越昂貴地活。

SECTION

22. 結論：屎山沒死，是因為它其實一直在適應

本文最終回答：

> 屎山為什麼沒有死？

不是因為結構問題不存在。

也不是因為軟體工程原則都沒有用。

而是因為很多 Big Ball of Mud 具有非常強的局部適應能力。

它們透過：

- piecemeal growth；
- keep it working；

- wrapping；
- compatibility；
- human workaround；
- embedded knowledge；
- familiarity capital；
- compensation；

  持續跟現實世界交換結構品質，以換取可運行性。

  所以更準確的描述不是：

> 「一堆壞東西不知道為什麼還能跑。」

而是：

> 「一個全局結構不佳、但局部適應機制極其成熟的生存系統。」

這也回到本系列最初的疑問。

某些大型軟體確實可能像：

> 由大量局部不完美零件、歷史補丁與補償結構組成，但外部功能依然完好的產品。

但這種狀態不是魔法。

它背後存在可以研究的機制：

> 【Operational Survival
> =
> Local Adaptation
> ⊗
> Accumulated Compensation
> ⊗
> Historical Knowledge】

而理解這些機制，正是安全重構的前提。

下一篇將進一步把目前混在一起使用的幾個詞拆開：

SECTION

〈技術債、架構侵蝕與歷史殘留：三種不同的結構負擔〉

因為不是所有泥巴都是 technical debt。

有些是債；

有些是侵蝕；

有些只是歷史沉積；

還有一些，是問題世界本身不可避免的複雜度。

SECTION

參考文獻

- Foote, B., & Yoder, J. (1997/1999). Big Ball of Mud. Fourth Conference on Pattern Languages of Programs (PLoP '97); later published in Pattern Languages of Program Design 4. https://www.laputan.org/mud/
- Weiderman, N. W., Smith, D. B., & Tilley, S. R. (1997). Approaches to Legacy System Evolution. Carnegie Mellon University Software Engineering Institute, CMU/SEI-97-TR-014.
- Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and Laws of Software Evolution—The Nineties View. Proceedings of the 4th International Software Metrics Symposium.
- Lehman, M. M., & Ramil, J. F. (2003). Software Evolution—Background, Theory, Practice. Information Processing Letters, 88(1–2), 33–44.
- 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.
- Li, R., Soliman, M., Liang, P., & Avgeriou, P. (2022). Symptoms of Architecture Erosion in Code Reviews: A Study of Two OpenStack Projects. arXiv:2201.01184.

- Knieke, C., Rausch, A., & Schindler, M. (2021). Tackling Software Architecture Erosion: Joint Architecture and Implementation Repairing by a Knowledge-based Approach. arXiv:2104.13919.
- Neo.K / EveMissLab.《表觀完好系統》系列第 1–3 篇，以及 MSSP / FPL 既有架構研究。