技術債、架構侵蝕與歷史殘留：三種不同的結構負擔

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

https://thisoneisneok.com/html/papers/ois-05.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：05 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

「技術債」已成為軟體工程中極度普及的詞彙，以至於幾乎所有舊程式碼、架構混亂、文件過時、相容層、特殊分支與難以修改的 legacy system 都可能被統稱為 technical debt。然而，這種用法雖然方便，卻會失去重要的因果差異。

本文主張，至少需要區分三類不同的結構負擔：

- 技術債（Technical Debt）：某項設計、實作或建造方式在短期具有便利或價值，但使未來修改、維護或演化成本增加；
- 架構侵蝕（Architecture Erosion）：軟體在持續演化中逐漸偏離預期、宣告或原有架構原則，導致結構品質、維護性與演化能力下降；

- 歷史殘留（Historical Residue）：過去曾有合理背景、相容需求、技術限制或業務條件的結構，在原始條件改變、消失或失去重要性後仍然保留於現行系統中。

  三者可以重疊，但不是同義詞。技術債描述的是一種跨時間成本關係；架構侵蝕描述的是一種結構偏離與退化過程；歷史殘留則描述一種時間沉積狀態。Parnas 的 software aging 研究尤其指出，成功軟體會同時受到兩種老化力量：未跟上環境變化，以及改動本身造成的結構劣化。這說明「舊」並不必然等於「債」，而且一個系統的結構負擔可能由完全不同的生成機制形成。

  本文進一步提出「結構負擔向量」：

> B_s(t)
> =
> [
> D□,
> E□,
> H□,
> N□
>]

其中 D□ 表示 technical debt，E□ 表示 architecture erosion，H□ 表示 historical residue，N□ 表示 necessary/essential complexity。加入 N□ 的目的，是避免把問題域本身不可消除的必要複雜度誤判為缺陷。

本文最後將這套區分接回 Dynamic MSSP/FPL：智能架構治理不能只輸出「這裡有技術債」，而應辨識負擔的來源、形成時間、仍然存在的理由、目前依賴、可移除性與轉化路徑。不同類型的結構負擔，需要不同治理策略。

關鍵詞：technical debt、architecture erosion、historical residue、software aging、legacy system、accidental complexity、MSSP、FPL、結構負擔、軟體演化

SECTION

1. 「技術債」這個詞被用得太廣了

今天只要看到一個難以理解的舊系統，很容易聽到：

> 「這就是技術債。」

如果有：

- 舊 framework；
- 特殊 API；
- 重複程式碼；
- 奇怪資料表；
- deprecated field；
- compatibility adapter；
- 循環依賴；
- 過時文件；
- legacy protocol；
- 人工流程；

也常全部被歸入：

> Technical Debt

這在工程溝通上很方便。

但理論上會造成一個問題：

> 如果所有不完美都叫債，那麼「債」就不再告訴我們問題究竟是怎麼形成的。

而如果不知道形成機制，就很難知道該怎麼處理。

例如：

- 一個為了趕 deadline 故意跳過抽象層的 shortcut；
- 一個原本正確、後來因二十次修改而逐步偏離的模組；

- 一個 2014 年必須保留、今天已經沒客戶使用的欄位；

  它們表面都可能長得很醜。

  但三者不是同一件事。

SECTION

2. 技術債：核心不是「爛」，而是未來成本

SEI 對 technical debt 的定義非常有助於收斂概念。

一種設計或建造方式，如果：

- 在短期具有便利；
- 但造成未來相同工作成本更高；

  就具有 technical debt 的性質。

  可概念化為：

> $U_{now} > 0$

同時：

> $C_{future} \uparrow$

其中：

- U_{now}：現在取得的效用；
- C_{future}：未來修改、維護、擴展或恢復所需成本。

  因此 technical debt 的關鍵不是：

> 程式碼醜不醜。

而是：

> 今天的選擇是否把成本推遲到未來。
>
> 可以寫成：

$$
D = \text{cal-R}(V_{(now)}, C_{(later)})
$$

其中 cal-R 表示跨時間成本關係。

SECTION

3. 技術債可以是理性的

這一點非常重要。

technical debt 的原始比喻並不等於：

> 做錯事。
>
> 有些技術債是合理交易。
>
> 例如：
>
> 現在：
> 先用單體架構
> → 三週上線 MVP
> 未來：
> 如果流量突破某門檻
> → 再拆服務
> 如果：

$$
P(\text{需要拆})
$$

本來就不高，

那麼提前投入六個月做高度可擴展架構，可能反而是不理性的。

所以：

> D > 0

不等於：

> Bad Decision

真正問題是：

- 有沒有看見它；
- 有沒有估計利息；
- 有沒有 owner；
- 有沒有 repayment trigger；
- 有沒有在環境改變後重新評估。

  這也是 SEI 將 technical debt 強調為「需要顯式管理」而不是單純消滅的原因。

SECTION

4. 技術債的時間方向

technical debt 有一個很重要的時間結構。

令：

> d□

為某個設計決策。

在時間 t□：

> V(d⊠,t⊠)>0

但對未來修改：

> C(d⊠,t>t⊠)
> >
> C(d^ast,t>t⊠)

其中：

> d^ast

代表較完整、較昂貴的替代方案。

這就是「債」。

因此 technical debt 是：

SECTION

Future-Cost Relation

它不必然要求架構已經發生侵蝕。

一個系統可以：

> D>0

但：

> E≈0

例如設計者明確決定：

> 現階段使用 SQLite，未來規模增長再遷移 PostgreSQL。

它可能完全符合當前架構規格。

這是 debt，但不是 erosion。

SECTION

5. 架構侵蝕：核心不是「欠」，而是「偏離與退化」

Architecture erosion 處理另一件事。

相關系統性研究指出，architecture erosion 不只表現為：

- architectural violation；
- structural issue；

  也會造成：

- software quality degradation；
- maintenance difficulty；
- evolution difficulty。

  而且成因不只有技術問題，也包含：

- deadline；
- organization；
- communication；
- developer awareness；
- process；
- ownership。

  因此 erosion 更接近：

> $A(t)$
> $\rightarrow$
> $A(t+\Delta t)$

時，系統逐步偏離：

> A_(intended)

或者逐步破壞原本重要結構性質。

可寫成：

> E_A(t)
> =
> D_A
> (
> A_(effective)(t),
> A_(intended)(t)
>)

其中：

> D_A

是架構差異或偏離程度。

SECTION

6. 技術債與架構侵蝕可以彼此獨立

這是本文第一個重要區分。

SECTION

Case A：有債，沒有明顯侵蝕

團隊有意採取較便宜方案：

single database
instead of
distributed data architecture

而且完全記錄在 ADR 與 FPL 規格中。

此時：

> $D>0$

但：

> $E\approx 0$

因為實作仍符合宣告架構。

SECTION

Case B：沒有明確借債，但發生侵蝕

原始架構規定：

TMS must access User data through UserPort

後來三個開發者分別為了局部便利，直接查 User table。

沒有人說：

> 我們今天要借一筆 technical debt。

但長期形成：

> $A_{effective}$
> $\neq$
> $A_{declared}$

這就是 erosion。

因此：

> $E>0$

但：

> D

不一定是原始形成原因。

SECTION

7. Erosion 是過程，Debt 更像負債狀態

可以進一步說：

> Debt

通常描述某個需要未來付出額外成本的狀態或決策結果。

而：

> Erosion

更強調：

> $(dQ_s)/(dt)<0$

或：

> $(dD_A)/(dt)>0$

亦即：

- 結構品質隨演化下降；
- 或宣告—有效架構距離隨時間增加。

  所以 erosion 具有更強的：

SECTION

Process Semantics

它不是只有：

> 現在有多糟。

而是：

> 系統正在往哪個方向變化。

這對 Dynamic MSSP 很重要。

因為靜態 snapshot 可能顯示：

> E(t□)

仍然不高，

但如果：

> (dE)/(dt)≫0

那麼其風險可能已經很高。

SECTION

8. Parnas：軟體老化有兩種不同來源

David Parnas 在 1994 年的 Software Aging 中提出一個非常重要的觀察：

成功軟體的老化無法完全避免，而且主要有兩種不同來源。

第一類：

> 軟體沒有跟著世界與需求變化。

第二類：

> 為了跟上世界而持續修改，但修改本身讓結構惡化。

可以概念化為：

> Aging
> =
> A_(stasis)
> +
> A_(change)

其中：

> A_(stasis)

來自不改，

而：

> A_(change)

來自改了以後的累積副作用。

這幾乎直接否定了：

> 「只要當初架構設計好，就不會老。」

因為：

> World Changes

本身就會讓過去合理的結構逐漸失去適配性。

這引出本文第三類：

SECTION

歷史殘留

SECTION

Historical Residue

SECTION

9. 歷史殘留：過去合理，不代表今天仍然需要

本文將 Historical Residue 定義為：

> 因過去的技術、業務、組織、相容、法規或環境條件而形成，且在原始條件改變、弱化或消失後，仍持續存在於現行系統中的結構。

例如：

- 舊客戶格式；
- 已停用 protocol 的 adapter；
- 不再使用的 database column；
- 為舊 browser 保留的 workaround；
- 過去法規要求的流程；
- 已不存在 service 的 compatibility layer；
- 舊部署方式留下的 config；
- 遷移期的 dual write；
- 曾經重要但現在沒有 caller 的 API。

它們的共同形式是：

在：

> t□

有：

> J(x,t□)>0

其中 J 表示存在理由。

但在：

> $t_{\square} > t_{\square}$

可能：

> $J(x, t_{\square}) \approx 0$

然而：

> $x \in S(t_{\square})$

仍然成立。

這就是歷史殘留。

SECTION

10. 歷史殘留不一定是技術債

這裡是最容易混淆的地方。

假設 2018 年為支援舊客戶：

API v1

完全合理。

2026 年已經沒有任何 v1 client。

但 API v1 還留著。

這時它可能變成：

> $H > 0$

即 Historical Residue。

但它未必一開始就是 technical debt。

因為當初的設計可能是：

> Correct

甚至：

> Optimal

只是時間改變了。

所以：

> 【Historical Residue
> ≠
> Original Bad Decision】

這個區分非常重要。

否則我們會用今天的條件反過來責怪過去合理的架構。

SECTION

11. 歷史殘留也不一定是架構侵蝕

假設架構文件明確寫著：

Support API v1 and v2
即使 v1 已經幾乎沒人用：

> A_d
> =
> A_e

可能仍然成立。

所以：

> $E \approx 0$

但：

> H>0

也完全可能。

因此：

> 【H
> not⇒
> E】

一個系統可以非常符合自己的架構，

但那個架構本身已經保留了太多歷史沉積。

SECTION

12. 三者的最小區分

可以整理成：

類型 核心問題 時間特性 典型問句

Technical Debt 未來成本 今天換未來 「今天省了什麼，未來多付什麼？」

Architecture Erosion 結構偏離／退化 演化過程 「系統是否逐漸偏離架構？」

Historical Residue 過去條件沉積 過去留到現在 「它現在為什麼還存在？」

因此：

> D≠ E≠ H

但三者存在交集。

SECTION

13. 三者如何互相轉化？

SECTION

13.1 Technical Debt → Architecture Erosion

為趕 deadline，團隊接受：

temporary direct DB access
如果之後大量模組仿效：

> D
> →
> E

一筆局部債變成架構性侵蝕。

SECTION

13.2 Technical Debt → Historical Residue

原本說：

> 上線後再重構。

但十年後仍在。

則：

> D
> →
> H

債變成歷史沉積。

SECTION

13.3 Architecture Erosion → Historical Residue

某次侵蝕造成：

duplicate service
後來新架構已經修好核心問題，

但 duplicate service 仍留下。

則：

> E
> →
> H

侵蝕留下殘骸。

SECTION

13.4 Historical Residue → Technical Debt

某個舊 adapter 原本只是 harmless residue。

但新功能每次都必須兼容它。

於是：

> H
> →
> D

歷史殘留開始對未來工作收取「利息」。

SECTION

14. 三者可以同時存在

考慮：

LegacyAuthAdapter
它可能同時是：

Historical Residue

因為原本的 legacy client 已經消失。

Technical Debt

因為每次修改認證都要多維護一套 adapter。

Architecture Erosion

因為新模組開始直接依賴這個原本應該退出的 adapter，造成依賴方向錯亂。

所以某個 artifact：

> x

可以有：

> D(x)>0

> E(x)>0

> H(x)>0

這不是分類失敗。

而是三個不同維度同時描述同一對象。

SECTION

15. 第四類：必要複雜度不能被誤殺

到這裡還不夠。

因為還有一類東西：

> 它很複雜，但不是債、侵蝕，也不是歷史殘留。

例如：

- distributed consensus；
- tax law；
- financial settlement；
- multi-jurisdiction compliance；
- access-control policy；
- versioned data migration；
- safety constraint。

它們可能來自問題世界本身。

本文用：

> N
> =
> Necessary / Essential Complexity

表示。

這對應經典軟體工程中「essential complexity」與「accidental complexity」的區分精神。

因此：

> Complexity
> ≠
> Debt

而：

> N>0

可能完全健康。

如果 AI 架構治理看見所有複雜度都想簡化，就可能：

> Simplification
> →
> Loss of Required Semantics

SECTION

16. 結構負擔向量

因此本文提出：

> 【B_s(t)
> =
> [
> D□,
> E□,
> H□,
> N□
>]】

其中：

- D□：Technical Debt；
- E□：Architecture Erosion；
- H□：Historical Residue；
- N□：Necessary Complexity。

  這四者不是同一種「壞」。

  真正治理目標不是：

> min
> (D+E+H+N)

因為：

> N

往往不能也不應該消滅。

甚至：

> H

也不一定需要全部刪除。

更合理是：

> min
> (
> D_(harmful),
> E_(uncontrolled),
> H_(unjustified)
>)

同時保留：

> N_(required)

以及必要的 compatibility residue。

SECTION

17. 歷史殘留有三種狀態

並不是所有 residue 都應刪。

可以進一步分為：

SECTION

17.1 Active Residue

仍然有真實依賴。

例如：

old API
仍有 2% 客戶使用。

它雖然歷史化，但仍有功能。

SECTION

17.2 Dormant Residue

目前沒有使用，但可能作為 recovery、migration 或 rare-path 保留。

需要驗證。

SECTION

17.3 Dead Residue

已無依賴、無恢復用途、無合規需求。

這才是最接近：

> Removal Candidate

的類型。

因此 Dynamic MSSP 不能只問：

> 它舊不舊？
>
> 而要問：

> 它還有沒有有效理由？

SECTION

18. Architecture Erosion 的治理方式和 Debt 不一樣

technical debt 通常適合：

- register；
- prioritize；
- estimate interest；
- repay；
- refinance；
- accept。

architecture erosion 更需要：

- detect divergence；
- recover intended architecture；
- identify violating dependency；
- compare declared/effective architecture；
- repair architecture or implementation；
- establish guardrail。

也就是：

> Debt Management
> ≠
> Erosion Control

即使兩者常重疊。

2021 年的 architecture repairing 研究就特別指出，只修 implementation 並不總夠，還需要同時考慮 architecture 與 implementation 的一致修復。

SECTION

19. Historical Residue 的治理方式又不同

對 residue 最危險的做法是：

> 看起來沒用，直接刪掉。

因為它可能仍然是：

- hidden compatibility；
- disaster recovery；
- rare customer；
- migration bridge；
- regulatory artifact；
- compensation anchor。

因此 residue governance 應該是：

> Identify Origin
> →
> Find Current Dependents
> →
> Check Runtime
> →
> Check Recovery
> →
> Check Compatibility
> →
> Retain / Deprecate / Remove

而不是：

> Old
> →
> Delete

SECTION

20. 技術債也有一個常見誤判：未完成不等於負債

假設某功能原本就不需要：

multi-region failover

那麼「沒有 multi-region」不是 technical debt。

只有當：

- 現在或未來確實需要；
- 當前設計使未來建立該能力更昂貴；

  才可能形成 debt。

  因此：

> Missing Feature
> ≠
> Technical Debt

同樣：

> Not Ideal
> ≠
> Debt

否則所有尚未實現的理想架構都會被稱為債。

這會讓技術債變成無限集合。

SECTION

21. Dynamic MSSP：從「問題標籤」變成「生成機制判斷」

傳統 Linter 很可能輸出：

```
WARNING:
Architecture violation detected.
Technical debt candidate.
```

未來 Dynamic MSSP 更適合輸出：

```
structural_burden:
 artifact: legacy_auth_adapter
 classification:
  technical_debt:
   score: 0.71
   evidence:
    - repeated maintenance cost
    - deferred replacement
  architecture_erosion:
   score: 0.43
   evidence:
    - new modules bypass declared AuthPort
  historical_residue:
   score: 0.91
   evidence:
    - original client retired
    - adapter created in 2019
  necessary_complexity:
   score: 0.12
 current_dependents:
  - billing_worker
  - recovery_script
 recommendation:
  action: staged_deprecation
  confidence: 0.79
```

這裡的重點不是數字本身。

而是：

> 同一個 artifact 可以被多個結構維度同時描述。

這比：

TECH_DEBT=true
更接近真實系統。

SECTION

22. 時間導數比靜態分數更重要

如果 Dynamic MSSP 真正動態化，不能只看：

$$D(t),E(t),H(t)$$

還需要：

$$\frac{dD}{dt}$$

$$\frac{dE}{dt}$$

$$\frac{dH}{dt}$$

例如：

System A

$$E=0.6$$

但：

$$\frac{dE}{dt}<0$$

正在改善。

System B

> E=0.2

但：

> (dE)/(dt)≫0

正在快速侵蝕。

那麼 System B 可能更值得立即處理。

這也是「軟體結構動力學」後續會正式展開的部分。

SECTION

23. 從 Parnas 回看：成功本身會產生沉積

Parnas 指出，software aging 幾乎是成功軟體必須面對的問題。

因為如果一個產品根本沒有活多久：

> H≈0

很容易。

它沒有時間形成歷史。

但成功產品：

> T_(lifetime)uparrow

通常意味著：

> Environment Changes
> uparrow

> Requirements Changes
> uparrow

> Compatibility History
> uparrow

> Organizational Turnover
> uparrow

所以：

> H

幾乎自然有成長壓力。

這不是失敗的證明。

它是時間的結果。

真正的問題是：

> 系統有沒有能力持續辨識哪些歷史還有價值、哪些只剩負擔。

SECTION

24. 命題 5：結構負擔非同質命題

本文提出系列第五個命題：

SECTION

結構負擔非同質命題

軟體系統中的結構負擔不能被單一「技術債」概念充分描述。

至少存在：

> 【B_s(t)
> =
> [D⊠,E⊠,H⊠,N⊠]】

其中：

> D⊠
> ≠
> E⊠
> ≠
> H⊠
> ≠
> N⊠

且不同維度可以：

- 重疊；
- 轉化；
- 共存；
- 以不同速度增減。

因此：

> 【Same Symptom
> not⇒
> Same Cause】

以及：

> 【Same Cause
> not⇒
> Same Remedy】

SECTION

25. 結論：不要再把所有舊東西叫技術債

本文最終不是要貶低 technical debt 這個概念。

恰恰相反。

正因為 technical debt 很有用，才不應讓它吞掉所有問題。

可以把三者濃縮成：

SECTION

技術債

> 今天的便利，讓未來變貴。

D
=
Deferred Cost Relation

SECTION

架構侵蝕

> 系統在演化中逐步偏離或破壞原本重要的結構。

E
=
Structural Divergence Process

SECTION

歷史殘留

> 過去曾有理由存在的東西，在條件改變後仍然留在今天。

> H
> =
> Temporal Residue

而必要複雜度：

> 不是歷史錯誤，而是問題本身就真的難。

> N
> =
> Essential Complexity

所以，當我們打開一座屎山時，不能只問：

> 「哪裡有 technical debt？」

更好的問題是：

> 這一層到底是欠下來的、侵蝕出來的、歷史留下來的，還是本來就無法消除的複雜度？

只有回答這個問題後，我們才知道：

- 該 repay；
- 該 repair；
- 該 deprecate；
- 該 preserve；
- 還是根本不該碰。

這也為下一篇做好準備。

如果某個 workaround、compatibility layer 或 temporary patch 長期存在，其他模組開始依賴它，甚至新架構都必須圍著它設計，那麼它就不再只是「債」或「殘留」。

它開始發生另一種變化：

> C□
> →
> A_(t+1)

也就是：

> 今天的 workaround，成為明天的 architecture。

下一篇將正式進入：

SECTION

〈今天的 Workaround，明天的 Architecture：補償凝固命題〉

SECTION

參考文獻

- 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/
- Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. IEEE Software, 29(6), 18–21.
- 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., Liang, P., Soliman, M., & Avgeriou, P. (2021). Understanding Architecture Erosion: The Practitioners' Perspective. arXiv:2103.11392.

- 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.
- Parnas, D. L. (1994). Software Aging. Proceedings of the 16th International Conference on Software Engineering (ICSE), 279–287. DOI: 10.1109/ICSE.1994.296790.
- Brooks, F. P. Jr. (1987). No Silver Bullet—Essence and Accidents of Software Engineering. Computer, 20(4), 10–19.
- Rios, N., Mendonça Neto, M. G., Spínola, R. O., & Seaman, C. (2020). A Survey of Self-Admitted Technical Debt. Journal of Systems and Software / Information and Software Technology literature stream.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–4 篇，以及 MSSP / FPL 既有架構研究。