複雜度配置論：程式語言沒有消滅的複雜度去了哪裡？

The Allocation of Complexity in Programming Language Design: Where Does the Complexity Go?

v1.0 / 2026-07-30 / Neo.K / 公開版／方法論基礎論文

https://thisoneisneok.com/html/papers/pldst-002.html

英文名稱： The Allocation of Complexity in Programming Language Design: Where Does the Complexity Go?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-002

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／方法論基礎論文

SECTION

摘要

程式語言設計經常以「更簡單」「更安全」「更自然」「零成本」「免管理」描述自身優勢。然而，一個介面變得簡單，並不代表支撐它的全部計算、規則、例外、工具與治理成本已經消失。垃圾回收降低程式設計者的手動記憶體管理負擔，卻增加運行時、延遲分析與實作者責任；型別推導減少顯式標註，卻提高編譯器推理、錯誤解釋與工具實作負擔；所有權系統能在沒有垃圾回收器的情況下提供記憶體安全保證，卻要求編譯器與使用者共同處理移動、借用及生命週期規則；零額外成本抽象限制了運行時成本，卻沒有宣稱編譯時間、語言規格與泛型診斷同樣為零；向後相容降低既有使用者的遷移成本，卻可能將歷史例外長期保存在語言、標準與工具中。[R3][R4][R5][R6]

Frederick Brooks 將軟體困難區分為本質性與偶發性，並主張軟體實體的複雜性具有不可約的本質部分；Niklaus Wirth 則批評硬體資源增長掩護了軟體膨脹，主張回到必要功能、嚴謹方法與可理解系統；Rob Pike 對 Go 的回顧顯示，語言特徵的減少可能是為了降低大型組織中的建置、閱讀、依賴與工具複雜度；Guy Steele 則指出，現代語言過於龐大，不宜一次設計完成，而應設計可由使用者與制度逐步生長的機制。[R1][R2][R3][R7]

本文提出 複雜度配置論（Complexity Allocation Theory, CAT-PL）。它拒絕兩種極端說法：

- 複雜度永遠嚴格守恆，任何簡化都只是騙局；
- 新語言機制可以無代價地消滅所有複雜度。

  本文將複雜度視為一組分布於語法、語義、編譯器、運行時、程式庫、工具、使用者、組織、生態與治理中的異質負擔，並區分九種流變：

- 消除；
- 壓縮；
- 封裝；
- 轉移；
- 延後；
- 攤銷；
- 複製；
- 隱藏；
- 制度化。

  本文建立複雜度配置向量、流變矩陣、可見性、支付時機、責任主體與失敗外溢等分析工具，並將其轉譯為 PLDST 設計師風格研究與 SKILL 可執行規格。

  核心命題是：

> 【表面簡單
> not⇒
> 系統總體簡單】

但同時：

> 【複雜度轉移
> ≠
> 複雜度必然守恆】

好的語言設計確實可能透過統一概念、消除重複狀態、限制非法組合與建立更好的抽象，真正降低部分總體複雜度；問題不在於「所有簡化都是轉移」，而在於必須說清楚：

> 哪一種複雜度被降低、哪一種被新增、由誰承擔、何時支付、是否可見，以及失敗時會外溢到哪裡。

關鍵詞： 程式語言設計、複雜度配置、偶發複雜度、本質複雜度、垃圾回收、所有權、零額外成本、語言演化、PLDST

SECTION

一、設計宣稱中的「簡單」

程式語言中的「簡單」至少可能指：

- 語法字元少；
- 核心概念少；
- 新手容易開始；
- 常見程式短；
- 編譯器容易實作；
- 執行成本容易預測；
- 錯誤容易理解；
- 大型程式容易維護；
- 語言版本容易演化；
- 生態系統容易治理。

這些含義並不相同。

一種語言可能：

- 語法很小，但巨集系統極複雜；
- 初學簡單，但大型系統難以重構；
- 使用簡單，但 Runtime 極複雜；
- 執行模型清楚，但型別系統難學；
- 語言規格簡潔，但程式庫慣例龐大；
- 單一程式容易理解，但跨版本演化困難。

因此，任何「這個設計更簡單」的主張，都必須回答：

> Simple for whom, in what activity, at what time?

SECTION

二、從總量問題轉成配置問題

常見提問是：

> 複雜度是否守恆？
>
> 這個問題過於粗糙，因為不同複雜度不能直接相加成一個無單位總量。
>
> 例如：
>
> - 一條額外語法規則；
> - 十秒編譯時間；
> - 一次難以理解的型別錯誤；
> - 一個不可預測的 GC 停頓；
> - 一年一次的版本遷移；

- 一個標準委員會的十年相容負擔；

  它們都可以被稱為複雜度，卻不是同一種量。

  更合適的問題是：

- 複雜度位於哪一層？
- 由誰承擔？
- 在哪個生命週期階段支付？
- 是否能被觀察與診斷？
- 是否被所有使用者支付？
- 是否只支付一次？
- 是否因規模增長而非線性放大？
- 失敗是否會跨層外溢？

  因此，本文將問題改寫為：

> 【複雜度配置
> =
> 位置
> +
> 責任
> +
> 時間
> +
> 可見性
> +
> 外溢】

SECTION

三、Brooks 的區分

Brooks 在〈No Silver Bullet〉中把軟體困難分成本質與偶發兩類，並指出軟體實體由大量彼此不同、以非線性方式互動的元素組成；抽掉這些關係，有時也會抽掉軟體問題本身的本質。[R1]

本文將其轉譯為：

3.1 本質複雜度

來自問題領域本身必須區分的狀態與關係，例如：

- 稅務制度中的例外；
- 分散式系統中的部分失敗；
- 權限系統中的角色與範圍；
- 即時控制中的時序限制；
- 金融帳務中的不可重複與可追溯要求。

3.2 偶發複雜度

來自表示、工具、歷史或實作方式，例如：

- 冗長樣板；
- 人工記憶體釋放；
- Header 重複解析；
- 不一致建置系統；
- 缺乏模組；
- 不必要格式差異；
- 缺少靜態檢查。

SECTION

四、二分法仍然不足

本質與偶發並不是永遠固定的分類。

某項複雜度可能：

- 在一個語言中屬偶發，在另一個執行模型中成為必要；
- 在單機程式中可忽略，在分散式環境中成為本質；
- 對語言實作者是本質，對使用者是封裝後的偶發；
- 隨硬體、工具與組織條件改變。

例如記憶體生命週期是實際問題，但「由使用者顯式 free、由 GC 追蹤、由所有權規則靜態檢查」是不同配置方式。

所以本文不把每項負擔永久標記為 essential 或 accidental，而記錄：

問題域是否要求
當前模型是否要求
當前實作是否引入
能否由其他機制取代

SECTION

五、不可約與可重構

本質複雜度並不表示：

- 每個使用者都必須直接看到它；
- 每次操作都要重新支付；
- 不可建立抽象；
- 不可由工具協助；
- 不可限制狀態空間。

好的設計可以把本質複雜度：

- 集中；
- 命名；

- 模組化；
- 建立不變量；
- 提供局部視圖；
- 使非法狀態不可表示；
- 透過一次性實作供多人重用。

它不能消除問題必需的差異，但可以降低人類同時掌握它們的負擔。

SECTION

六、表面複雜度 C_s

包括語法、標點、關鍵字、顯式標註、樣板、版面規則與儀式性程式碼。表面複雜度最容易被看見，因此也最容易被過度重視。短程式不一定代表簡單語義；長程式也不一定代表難以推理。

SECTION

七、語義複雜度 C_m

包括名稱解析、型別規則、隱式轉換、求值順序、生命週期、作用域、例外、並行、未定義行為與特徵交互作用。它決定讀者需要掌握多少規則才能預測程式的意思。

SECTION

八、編譯器複雜度 C_c

包括解析、型別推導、借用檢查、最佳化、單態化、巨集展開、增量編譯、診斷與跨模組分析。把責任交給編譯器，可能降低使用者重複工作，但會提高實作者成本與工具依賴。

SECTION

九、運行時複雜度 C_r

包括垃圾回收、動態派發、JIT、排程器、協程、反射、動態載入、安全檢查與例外展開。運行時複雜度常被日常語法隱藏，直到延遲、記憶體、暫停、部署或故障成為問題。

SECTION

十、程式庫複雜度 C_l

語言核心小，不代表使用者接觸的概念少。複雜度可能進入標準程式庫、框架、套件、設計模式、型別類別、慣例與 DSL。語言與程式庫的邊界本身就是設計選擇。

SECTION

十一、工具複雜度 C□

包括 IDE、Linter、Formatter、Build tool、Package manager、Language server、Debugger、Profiler 與 Migration tool。工具可吸收大量偶發複雜度，也可能使語言離開工具後難以使用。

SECTION

十二、使用者複雜度 C_u

包括學習、記憶、局部推理、除錯、效能分析、遷移、認知切換與團隊溝通。同一負擔對不同使用者群可能完全不同：

> C_u=f(Experience,Task,Scale,Tooling,PriorKnowledge)

SECTION

十三、組織複雜度 C_o

包括大型程式碼庫、團隊風格差異、建置時間、依賴管理、Code review、版本偏移、人員流動與跨語言整合。Go 的設計回顧明確將許多問題定位在這一層，而不只是個別程式的語法層。[R3]

SECTION

十四、生態複雜度 C_e

包括套件數量、重複功能、依賴樹、安全漏洞、發布流程、版本衝突、平台差異與文件碎片化。一種容易擴展的語言，可能降低核心壓力，卻讓生態多樣性快速增長。

SECTION

十五、治理與演化複雜度 C_g

包括提案程序、標準、向後相容、棄用、Edition、多實作一致性、委員會折衷與長期維護。一個功能發布後，設計成本可能在未來數十年持續產生相容、教學、規格、工具與特徵交互成本。

SECTION

十六、配置向量

本文將某個設計 d 的複雜度配置表示為：

> C(d)=
> (C_s,C_m,C_c,C_r,C_l,C□,C_u,C_o,C_e,C_g)

這不是把所有維度壓成總分，而是保存分布。

SECTION

十七、消除 Elimination

真正移除不必要狀態、重複與規則，例如統一類別與型別模型、消除重複宣告、取消可由單一規則涵蓋的特例，或將非法狀態從資料模型中移除。若原本有多個彼此交互的特例，統一規則可能同時降低多層複雜度。

SECTION

十八、壓縮 Compression

以更一般的概念表示多個案例，例如泛型、高階函數、模式匹配、參數化模組、Traits 與代數資料型別。壓縮降低重複，但可能提高抽象理解與編譯器成本。

SECTION

十九、封裝 Encapsulation

複雜度仍存在，但透過模組、抽象資料型別、資源類別、Runtime 或標準程式庫，使每位使用者不必直接處理。封裝的成功條件是介面穩定、洩漏有限、錯誤可診斷且效能模型足夠透明。

SECTION

二十、轉移 Displacement

複雜度從一個載體移至另一個載體。例如：

> C_udownarrow, C_ruparrow

可能表示使用者少做記憶體管理，而 GC Runtime 承擔追蹤與回收。又如：

> C_sdownarrow, C_c+C⊠uparrow

表示程式少寫型別，編譯器與 IDE 承擔推導與錯誤解釋。

SECTION

二十一、延後 Deferral

當前階段變簡單，但成本在之後支付。例如動態型別將部分錯誤延至測試或運行、不固定依賴版本把決策延至部署、不處理棄用把成本延至重大版本。延後不一定錯誤；探索階段延後約束可能提高速度，但需要清楚的後續支付機制。

SECTION

二十二、攤銷 Amortization

一次支付高成本，讓大量使用者受益，例如編譯器實作者實作型別推導、標準程式庫提供安全容器、Formatter 統一風格、Runtime 實作排程。若一次性成本由少量專家承擔並可靠服務大量使用者，總體負擔可能真正下降。

SECTION

二十三、複製 Duplication

為了局部簡單，系統在多處重複表示同一概念，例如 Schema、型別與文件分開維護，Client／Server 重複模型，或多種套件工具各自保存依賴資料。局部簡化可能造成全域同步負擔。

SECTION

二十四、隱藏 Concealment

複雜度仍存在，但使用者無法觀察，例如隱式網路請求、不可見配置搜尋順序、無法解釋的型別錯誤、Runtime 自動重試與隱式全域狀態。隱藏與封裝不同：封裝提供穩定邊界；隱藏使原因與成本難以診斷。

SECTION

二十五、制度化 Institutionalization

複雜度被移入 RFC、標準委員會、相容性審查、Edition、Feature gate 與 Deprecation policy。制度化降低個別使用者自行協調的成本，但增加決策時間與程序負擔。

SECTION

二十六、流變矩陣

對一項語言決策 q，定義：

> $\Delta C_q = C_{after} - C_{before}$

描述載體間流動：

> $F_q = [f_{ij}]$

其中：

- $f_{ij} > 0$：負擔由載體 i 移向 j；
- $f_{ii} < 0$：該層負擔被消除或壓縮；
- 新增複雜度可由外部來源節點 $\varnothing$ 進入；
- 真正消除可流向終止節點 $\perp$。

SECTION

二十七、責任向量

> R(q)=
> (R_(designer),R_(implementer),R_(tool),R_(programmer),R_(operator),R_(organization)
> ,R_(ecosystem))

同一機制可能把複雜度從數百萬使用者攤銷到少量實作者，因此即使實作者成本上升，社會總負擔仍可能下降。

SECTION

二十八、支付時間

> T=(Design,Learn,Write,Compile,Test,Run,Debug,Deploy,Migrate,Govern)

某項機制可能增加學習與編譯成本、降低測試與運行事故、增加版本治理並降低部署除錯。因此不能只看「寫第一個範例需要幾行」。

SECTION

二十九、可見性

定義 V_c∈[0,1] 表示負擔對承擔者的可觀察程度。高可見複雜度不一定比較糟；明確錯誤、顯式 Capability 與可讀效能成本，可能比低可見但會在生產環境爆發的複雜度更容易治理。

SECTION

三十、外溢性

定義：

> S_c=failure spillover radius

語法錯誤通常局部；型別系統缺陷可能影響整個編譯；GC 暫停可能影響服務延遲；依賴供應鏈問題可能影響整個生態；相容性決策可能延續數十年。設計評估不能只看平均成本，也要看失敗半徑。

SECTION

三十一、垃圾回收：從使用者移到 Runtime

手動管理模式：

> C_u+C_(debug)uparrow, C_rdownarrow

垃圾回收模式：

> C_udownarrow, C_r+C_(latency)+C_(impl)uparrow

它可能真正降低總體複雜度，因為釋放邏輯不再散布於所有程式，大量 use-after-free 與 double-free 路徑被移除，回收器成本由少數實作者攤銷。但它也可能引入暫停、記憶體峰值、Root 掃描、Finalizer 語義與外部資源不同步。因此「免記憶體管理」是介面層說法，不是系統層事實。

SECTION

三十二、Rust 所有權：從 Runtime 移到規則、編譯器與學習

Rust 官方文件說明，所有權讓 Rust 在不依賴垃圾回收器的情況下提供記憶體安全保證。[R5]

其配置近似：

> C_rdownarrow

> C_m+C_c+C_u^(learn)uparrow

同時：

> C_u^(memory bugs)+C_(debug)^(memory)downarrow

所有權不是單純「更複雜」或「更簡單」，而是增加前置規則、編譯器分析與初期認知負擔，換取大量運行期記憶體錯誤的減少與較直接的效能模型。其價值取決於錯誤成本與目標領域。

SECTION

三十三、C++ 零額外成本：限制 Runtime，不代表所有成本為零

Stroustrup 將輕量抽象描述為：其時間與空間開銷不應高於針對同一抽象仔細手寫的實作，並用「不用的不付費，使用的難以手寫得更好」概括零額外成本原則。[R4]

這主要約束：

> C_r^(time)+C_r^(space)

但不表示：

> C_c=C_m=C⌧=C_u=0

模板、泛型、最佳化與靜態抽象可能增加編譯時間、錯誤訊息、語言規格、工具、學習與二進位大小。零額外成本是特定維度承諾，不應被誤讀成全部複雜度為零。

SECTION

三十四、Go：減少語言自由以降低組織複雜度

Go 的設計回顧把痛點列為慢建置、不受控依賴、每位程式設計者使用不同語言子集、程式難讀、更新成本、版本偏移、工具困難與跨語言建置。[R3]

因此某些限制可表示為：

> C_s^(freedom)+C_l^(choice)downarrow

換取：

> C_o^(coordination)+C⌧^(automation)+C_u^(reading)downarrow

這不是證明 Go 對所有任務最簡單，而是說明其設計目標位於大型組織工程。

SECTION

三十五、Wirth 與 Lean Software：拒絕硬體替軟體膨脹買單

Wirth 批評軟體大小超過功能成長，指出更快硬體與更多記憶體容易掩護鬆散設計；他以 Oberon 說明可透過必要功能、嚴謹語言與可理解系統降低資源與概念負擔。[R2]

其配置主張不是把成本移到更強硬體，而是減少非必要功能、限制特徵交互、讓編譯器協助發現概念錯誤，並讓系統能被少數人完整理解。這屬於真正消除與壓縮的企圖，而不是純轉移。

SECTION

三十六、Steele 的 Growing a Language：把完整性問題改寫成演化問題

Steele 指出，現代語言與需求已大到難以一次設計與建成，因此應為使用者擴展與後續生長設計機制。[R7]

配置是：

> C_(initial design)downarrow

但：

> C_g+C_e+C_(compatibility)uparrow

可成長語言降低中央設計者一次預知所有需求的負擔，卻需要擴展規則、模組化、社群篩選、相容性、合併與修剪機制。「可生長」不是免費功能，而是把一次性完整設計轉為長期制度化演化。

SECTION

三十七、命題一：複雜度非嚴格守恆

若新機制消除重複狀態、非法組合與交互特例，則在某種經合理定義的負擔度量下可能成立：

> Σ☒C☒^(after)<Σ☒C☒^(before)

因此「任何簡化都只是轉移」過強。

SECTION

三十八、命題二：局部簡化可能增加全域複雜度

若每個局部元件都建立自己的便利抽象，可能造成重複概念、不一致錯誤、多套配置、工具碎片與依賴衝突。因此：

> $\forall\, i, \Delta\, C\boxtimes^{(local)} < 0$

不推出：

> $\Delta\, C^{(global)} < 0$

SECTION

三十九、命題三：被攤銷的複雜度可能是好複雜度

若複雜度由少量專家一次承擔，並可靠服務大量使用者：

> $C_{(implementer)} \uparrow$

但：

> $N \cdot C_{(user)} \downarrow$

當 N 足夠大時，總社會成本可能下降。

SECTION

四十、命題四：不可見複雜度具有利息

若複雜度被隱藏而非封裝，可用啟發式表示：

> $I_c(t) = P_c \cdot r \cdot t$

其中 P_c 是隱藏複雜度本金，r 是交互與變更率，t 是存在時間。這不是財務精確公式，而是表示無法觀察的規則會隨功能交互與人員更替增加診斷成本。

SECTION

四十一、命題五：責任配置必須匹配能力

如果某主體承擔責任，卻沒有足夠可見性與控制能力，則形成失衡：

> Responsibility(a)>Authority(a)+Observability(a)

例如要求使用者保證效能，卻隱藏配置與 Runtime 行為；要求程式庫維護相容性，卻無法控制語言變更；要求編譯器保證安全，卻允許無邊界外部副作用。

SECTION

四十二、命題六：相容性是時間上的複雜度配置

向後相容可降低當前使用者遷移成本：

> C_u^(migrate)downarrow

但增加：

> C_m+C_c+C⊠+C_g

並將部分成本分散到未來版本。相容性既不是純美德，也不是純負債，而是跨時間與跨使用者世代的配置選擇。

SECTION

四十三、不要問「哪個最簡單」

應分別問：

- 初學者第一次成功需要什麼？
- 專家預測成本需要什麼？
- 大型團隊協作需要什麼？
- 編譯器與工具實作者承擔什麼？
- 運行時支付什麼？
- 版本演化支付什麼？

- 故障時誰診斷？
- 離開完整工具鏈後還剩什麼？

SECTION

四十四、配置卡

每項設計機制使用：

機制：
解決的原始問題：
被降低的複雜度：
新增的複雜度：
承擔者：
支付時間：
可見性：
失敗外溢：
是否可攤銷：
是否可逆：
規模效應：
證據：

SECTION

四十五、負擔矩陣

載體／階段	學習	撰寫	編譯	運行	除錯	遷移	治理
語法							
語義							
編譯器							
Runtime							
工具							
使用者							
組織							
治理							

分數只作比較索引，必須附實際證據。

SECTION

四十六、不同尺度分開評估

定義：

> C(d,n)

其中 n 可以是程式大小、團隊人數、依賴數、存活年限、部署節點或語言版本數。某設計在 n 小時簡單，在 n 大時可能急遽惡化。

SECTION

四十七、設計師在保護誰

分析一位設計者時，應觀察他反覆降低哪一方成本：初學者、專家、Runtime、編譯器、大型組織、生態維護者、硬體或未來版本。這形成 保護對象指紋。

SECTION

四十八、設計師把複雜度移到哪裡

可形成：

> Π_d=(PreferredSinks,RejectedSinks,HiddenSinks)

例如願意增加編譯器複雜度、拒絕不可預測 Runtime、接受語法冗長、拒絕使用者手動資源管理、接受長期治理程序，或拒絕語言核心持續增長。

SECTION

四十九、設計師何時支付成本

前置支付型

- 學習；
- 型別；
- 設計；
- 編譯。

運行支付型

- 動態檢查；
- GC；
- JIT；
- 反射。

事故支付型

- 測試；
- 除錯；
- 生產故障。

演化支付型

- 相容；
- 遷移；
- 標準；
- 棄用。

設計風格可被描述為支付時機偏好。

SECTION

五十、對 PLDST-001 的擴展

PLDST-001 的複雜度配置向量，現擴展為：

> cal-A_(d,t)=(C,F,R,T,V_c,S_c)

其中：

- C：載體分布；
- F：流變矩陣；
- R：責任；
- T：支付時間；
- V_c：可見性；
- S_c：外溢半徑。

SECTION

五十一、決策抽取

SKILL 對每個語言機制抽取：

```
{
 "decision": "ownership",
 "problem": "memory safety without mandatory GC",
 "reduced_complexity": [
  "runtime memory faults",
  "manual lifetime debugging"
 ],
 "increased_complexity": [
  "static semantics",
  "compiler analysis",
  "learning"
 ],
 "responsible_actors": [
  "compiler implementer",
  "programmer"
 ],
 "payment_time": [
  "learning",
  "compile"
```

```
 ],
 "visibility": "high",
 "spillover": "mostly compile-time",
 "evidence": []
}
```

SECTION

五十二、禁止簡單守恆輸出

SKILL 不得只輸出：

複雜度沒有消失，只是轉移。

它必須判斷是否有真正消除、是否只是局部轉移、是否因攤銷降低總成本、是否隱藏、是否延後、是否複製，以及是否制度化。

SECTION

五十三、比較輸出

比較兩位設計者時，應輸出共同問題、不同配置、不同承擔者、不同支付時間、不同失敗模式、不同規模假設與不同相容性策略，而不是只比較特徵多少。

SECTION

五十四、信心標記

每項配置推論標記：

[F] 可確認機制
[Q] 設計者直接說明
[D] 可辨識決策
[I] 本文配置推論
[C] 反例
[U] 證據不足

SECTION

五十五、語法短即簡單

高度隱式語法可能減少字元，卻增加名稱解析、型別推斷、錯誤來源與工具依賴。短只能證明表面長度下降。

SECTION

五十六、靜態檢查一定增加總複雜度

靜態系統雖增加規則與編譯器，但可能刪除防禦性程式碼、測試狀態、生產事故、文件約定與人工 Code review 項目。應做全生命週期比較。

SECTION

五十七、動態系統一定比較簡單

動態系統可能適合探索與異質資料，但大型演化可能把負擔移到測試、Runtime、文件、人工推理與重構工具。不能用起步速度代表全部週期。

SECTION

五十八、零成本就是沒有代價

零額外成本通常指特定 Runtime 時間與空間維度，不包括編譯時間、規格、診斷、認知、工具與二進位膨脹。

SECTION

五十九、核心小就是系統小

小核心可把複雜度移至巨集、程式庫、DSL、社群慣例與擴展治理。必須分析整個有效語言：

> $L_{(effective)}=Core+Library+Tooling+Conventions+Extensions$

SECTION

六十、本文不是複雜度物理學

本文公式是分析表示，不主張複雜度具有像能量一樣嚴格可測與守恆的物理量。它的用途是迫使研究者說清楚減少的是什麼、增加的是什麼、負擔去了哪裡、誰支付與何時支付。

SECTION

六十一、不同負擔不能任意加總

除非先定義任務、使用者、權重、時間範圍、風險與規模，否則：

> $C_s+C_r+C_g$

只是一個符號集合，不是可直接比較的總量。

SECTION

六十二、良好設計可能真正降低複雜度

本文反對「所有設計只是搬運成本」的犬儒觀點。真正降低可能來自統一、消除重複、更好的資料模型、更小狀態空間、局部推理、更強模組邊界、可重用工具、自動化攤銷，以及讓非法狀態不可表示。

SECTION

六十三、簡單具有情境性

某機制可能對小程式過重、對大型系統必要；對初學者困難、對專家節省大量事故；對低延遲系統不適合、對商業後台極有效。PLDST 必須避免無條件簡單排行。

程式語言設計中的複雜度問題，不應被縮成兩句口號：

> 「複雜度都不會消失。」

或：

> 「這項機制讓一切變簡單。」

兩者都忽略了語言設計真正的工作：選擇哪些差異必須保留、哪些可以統一、哪些應由專家一次處理、哪些應對使用者顯式、哪些成本可以延後，以及哪些歷史負擔值得長期保存。

本文提出：

> C(d)=
> (C_s,C_m,C_c,C_r,C_l,C□,C_u,C_o,C_e,C_g)

並以九種流變描述設計決策：

> 【消除
> +
> 壓縮
> +
> 封裝
> +
> 轉移
> +
> 延後
> +
> 攤銷
> +
> 複製
> +
> 隱藏
> +
> 制度化】

一項好的語言設計，不是讓某個展示範例看起來最短，而是使複雜度：

- 出現在有能力處理它的位置；
- 由擁有相應控制權的主體承擔；
- 在適當階段支付；
- 保持足夠可見；
- 不造成不可接受的失敗外溢；
- 能透過重用與攤銷降低整體負擔；

- 不把今日便利轉成無限期的未來負債。

因此，PLDST 對設計者最關鍵的問題之一，不是：

> 他喜歡簡單還是複雜？

而是：

> 他認為哪一種複雜度是必要的、哪一種可以消除、哪一種應由誰承擔，以及他願意把代價推到什麼時間與系統層級？

這一配置模式，正是程式語言設計師風格最穩定、也最能跨越不同語言作品的深層指紋之一。

設計機制：
語言／版本：
原始問題：
目標使用者：
本質負擔：
偶發負擔：
消除：
壓縮：
封裝：
轉移：
延後：
攤銷：
複製：
隱藏：
制度化：
新增負擔：
主要承擔者：
支付階段：
可見性：
失敗外溢：
規模效應：
相容性影響：
證據：
反例：
信心：

[R1] Frederick P. Brooks, Jr., “No Silver Bullet: Essence and Accidents of Software Engineering,” UNC Technical Report TR86-020, 1986; later published in Computer, 1987.

— 本質與偶發困難、軟體狀態與交互複雜性、組織與理解負擔。

[R2] Niklaus Wirth, “A Plea for Lean Software,” Computer, Vol. 28, No. 2, 1995, pp. 64–68.

— 軟體膨脹、硬體資源掩護、必要功能、嚴謹方法與 Oberon 案例。

[R3] Rob Pike, “Go at Google: Language Design in the Service of Software Engineering,” SPLASH 2012, Go official website.

— 大型軟體工程、建置、依賴、閱讀、工具與組織複雜度。

[R4] Bjarne Stroustrup, “Foundations of C++,” ETAPS 2012 keynote draft.

— 直接硬體映射、輕量抽象與零額外成本原則。

[R5] Steve Klabnik and Carol Nichols, The Rust Programming Language, “Understanding Ownership,” official Rust documentation.

— 所有權在不需要垃圾回收器的情況下提供記憶體安全保證。

[R6] The Rust Project, “Experiment with ergonomic ref-counting,” Rust Project Goals, 2025H1.

— 顯式參考計數複製對部分應用形成偶發複雜度的官方問題描述。

[R7] Guy L. Steele Jr., “Growing a Language,” invited talk at OOPSLA 1998; final version in Higher-Order and Symbolic Computation, Vol. 12, No. 3, 1999.

— 現代語言規模、一次性完整設計的困難、使用者擴展與制度化生長。

[R8] Thomas R. G. Green and Marian Petre, “Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework,” Journal of Visual Languages & Computing, 7(2), 1996.

— 以多維權衡而非單一總分分析記號系統。

[R9] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine, “Interdisciplinary Programming Language Design,” Onward! ’18, 2018, pp. 133–146, DOI: 10.1145/3276954.3276965.

— 依使用者、活動與情境選擇語言品質屬性的方法論。

本文完成初稿後，再次以原始論文、官方語言文件與正式會議資料核對下列事項：

- Brooks 的年代與命題範圍

  UNC 技術報告 TR86-020 的日期為 1986 年 9 月；文章後於 1987 年刊於 Computer。本文只把「本質／偶發」用作分析起點，沒有把 Brooks 的論證擴張成「所有軟體複雜度均不可降低」。

- Wirth 的書目與原意

〈A Plea for Lean Software〉刊於 Computer 28(2)，1995 年 2 月，頁 64–68。Wirth 確實批評軟體膨脹、硬體資源掩護與無節制功能增長，並以 Oberon 主張回到必要功能與嚴謹設計。本文將其歸類為消除與壓縮的企圖，而不是聲稱 Oberon 已消除所有複雜度。

- Go 的發表情境

  〈Go at Google〉是 Rob Pike 在 2012 年 10 月 25 日 SPLASH 大會 keynote 的修改稿。其官方文章明確列出慢建置、不受控依賴、語言子集差異、閱讀、更新與工具等痛點。本文因此把 Go 解讀為組織複雜度配置，而不是泛稱它在所有情境中最簡單。

- C++ 零額外成本的限定範圍

  Stroustrup 的 Foundations of C++ 是 ETAPS 2012 keynote 稿，將輕量抽象限定於不超過仔細手寫實作的時間與空間開銷。本文已明確區分 Runtime 成本與編譯時間、規格、工具及學習成本，沒有把 zero-overhead 誤寫成 zero-total-cost。

- Rust 所有權與人體工學負擔

  Rust 官方《The Rust Programming Language》確實說明所有權可在不需要垃圾回收器的情況下提供記憶體安全保證。Rust 2025H1 的官方 ergonomic reference-counting 目標也直接將顯式複製參考計數值描述為許多應用中的 significant accidental complexity。本文據此說明同一系統內仍會持續重新配置複雜度，沒有宣稱 Rust 所有權能排除所有記憶體洩漏或所有 unsafe 風險。

- Steele 的出版資訊與主張

  〈Growing a Language〉源自 1998 年 OOPSLA invited talk，正式版本刊於 1999 年 Higher-Order and Symbolic Computation 12(3)，頁 221–236。本文只採用其「大型語言不宜一次設計完成、應規劃生長」的論點，並自行分析其治理與相容成本；後者屬 [I] 推論，不是 Steele 的逐字原話。

- 跨學科語言設計的書目修正

  Coblenz、Aldrich、Myers 與 Sunshine 的〈Interdisciplinary Programming Language Design〉正式發表於 Onward!’18，不是名為「PL’18」的獨立會議。參考文獻已修正。

- 複雜度守恆的限制

  本文沒有把複雜度視為可跨維度直接相加的物理量。所有總和公式都附帶「須先定義任務、使用者、權重、時間與規模」的限制；九種流變是分析詞彙，不是經驗定律。

- 垃圾回收案例的歸因限制

本文對垃圾回收的成本配置是一般工程分析，不歸因於某一位設計師，也不主張所有 GC 都具有相同延遲、吞吐或記憶體特性。

- 史實與本文推論分層

  Brooks、Wirth、Pike、Stroustrup、Rust 與 Steele 的直接主張由 [R] 來源支撐；「九種流變」「十個載體」「責任向量」與六個命題均為本文提出的方法論模型，不應被誤讀為這些作者共同主張的既有理論。