程式語言設計師風格譜系總論：設計自由、複雜度與代價

A General Theory of Programming Language Designer Styles: Design Freedom, Complexity, and Cost

v1.0 / 2026-07-30 / Neo.K / 公開版／第一批 30 篇封頂總論

https://thisoneisneok.com/html/papers/pldst-030.html

英文名稱： A General Theory of Programming Language Designer Styles: Design Freedom, Complexity, and Cost

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-030

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第一批 30 篇封頂總論

系列進度： 30／30

建議書名： 《設計語言的人：程式語言設計師的風格、選擇與代價》

SECTION

摘要

程式語言設計史通常以範式、語法、型別系統、執行模型與著名語言為主線。這些分類可以說明一門語言「具有什麼」，卻較難回答另一組問題：

- 為何不同設計者面對相似限制，會反覆保護不同價值？
- 為何一門語言的「簡單」常伴隨 Compiler、Runtime、工具、函式庫或治理複雜度的增加？

- 為何安全、自由、可讀性、機器透明、相容性與快速演化無法被同時無限最大化？
- 為何創始者原始思想、共同設計、參考實作、現代治理及生態結果必須分開？
- 為何同一設計者在不同時期可能呈現不同風格？
- AI 又如何在不冒充歷史人物的前提下，使用這些風格分析新問題？

PLDST 第一批三十篇的核心結論是：

> 程式語言不是抽象功能的中性容器，而是一份關於自由、複雜度、責任、時間與權力如何被配置的技術—制度契約。

本文把程式語言設計表示為：

> 【mathfrakL_(d,t)
> =
> (
> C,
> P,
> V,
> F,
> B,
> R,
> E,
> G,
> T,
> O
>)】

其中：

- C：Context，歷史、硬體、組織與生態背景；
- P：Problem framing，設計者如何定義問題；
- V：Value ordering，價值衝突時的優先序；

- F：Freedom allocation，自由被配置給誰；
- B：Burden allocation，複雜度與成本被配置到哪裡；
- R：Responsibility allocation，誰負責預防、偵測、修復與承擔後果；
- E：Evidence standard，何種證據足以支持或拒絕功能；
- G：Governance，誰能提出、裁決、實作、發布及接班；
- T：Temporal evolution，風格、語言與制度如何跨時間變化；
- O：Outcome，實際實作、生態與長期結果。

  因此，設計風格不是語言特徵總和，也不是心理人格：

> 【DesignStyle
> =
> RepeatedTradeoffRules
> +
> BurdenAllocation
> +
> ResponsibilityAllocation
> +
> EvidenceStandard
> +
> Governance
> +
> TemporalRevision】

本文進一步提出五項總論命題。

第一，自由是複數的。作者表達自由、讀者理解自由、實作者最佳化自由、使用者遷移自由、領域社群延展自由與治理參與自由可能彼此衝突。

第二，複雜度可以被真正刪除，也可以被壓縮、封裝、轉移、延後、攤銷、複製或隱藏。PLDST 不主張粗糙的物理式「複雜度永恆守恆」，而主張所有「變簡單」的設計都必須追問剩餘負擔及新負擔的位置。

第三，責任應與控制權及可觀測性對齊。如果使用者沒有足夠資訊與控制，卻被要求承擔錯誤；或 Compiler、Runtime、工具擁有控制權，卻不提供可解釋回饋，設計便形成責任錯置。

第四，相容性把過去轉成現在的共同作者。一門成熟語言的設計者不只與當代需求協商，也與舊程式、ABI、工具、教材、組織知識和使用者信任協商。

第五，治理是語言本體的一部分。PEP、RFC、Steering Council、Core Team、企業工程資源與標準委員會決定什麼能成為下一版語言，因此治理不是附錄，而是演化語義的控制層。

PLDST 最終不是建立「誰最好」的排行榜，而是建立一套可回答下列問題的分析語言：

> 當設計者面對不可同時滿足的要求時，他保護什麼、限制什麼、把代價交給誰、要求什麼證據，又如何讓這些選擇跨越時間形成可辨識的設計風格？

關鍵詞： PLDST、程式語言設計、設計師風格、設計自由、複雜度配置、責任配置、相容性、語言治理、設計決策、程式語言史、AI 模擬

SECTION

一、從語言特徵轉向決策規則

傳統分類常使用：

命令式
函數式
物件導向
邏輯式
動態型別
靜態型別
編譯式
直譯式
系統語言
腳本語言
這些標籤描述語言結果，但不能直接解釋設計者。

SECTION

二、相同特徵可以來自不同理由

兩門語言都可能使用型別推導，但一門是為了：

- 減少樣板；

另一門可能是為了：

- 保持形式推理；
- 提升工具；
- 讓既有動態生態逐步加入證據；
- 不破壞原始表面。

  因此：

> SameFeature
> not⇒
> SameStyle

SECTION

三、不同特徵可以來自同一深層風格

同一設計者可能在不同語言中採取不同機制，卻維持相似決策規則。

例如工具驅動風格可以在：

- 快速編譯器；
- IDE；
- 漸進型別；
- Metadata；
- Language service；

  中呈現，而不必綁定單一語法。

SECTION

四、最小研究單位

PLDST 的最小單位不是人物評語，而是 Design Decision Record：

> DDR
> =
> (
> Actor,
> Time,
> Problem,
> Constraints,
> Options,
> Decision,
> Rationale,
> Implementation,
> Consequences,
> Evidence
>)

SECTION

五、風格是多筆決策的聚合

> Style(d,τ)
> =
> Aggregate
> (
> DDR⊠,…,DDR⊠
>)

一段名言最多只能成為一項證據。

SECTION

六、風格具有時間索引

> Σ_d
> =
> {
> Σ_(d,t□),
> Σ_(d,t□),
> ⋯,
> Σ_(d,t□)
> }

所以：

> LanguageToday
> not⇒
> FounderAlwaysBelievedThis

SECTION

七、語言設計沒有單一目標函數

語言可能同時追求：

- 可讀；
- 可寫；
- 可學；
- 安全；
- 快速；
- 可攜；
- 可延展；
- 可推理；
- 可最佳化；

- 可部署；
- 可相容；
- 可治理。

SECTION

八、多目標形式

> J
> =
> w_RR
> +
> w_WW
> +
> w_SS
> +
> w_PP
> +
> w_CC
> +
> w_EE
> +
> w_GG
> -
> Cost

權重 w☒ 不只是技術常數，而是設計風格的核心。

SECTION

九、真正差異在衝突時出現

所有設計者都可能支持「簡單」「實用」「安全」。

只有在以下情況才看見風格：

安全與互操作衝突
可讀與短碼衝突
相容與一致性衝突
抽象與成本透明衝突
個人品味與社群需求衝突

SECTION

十、設計風格是排序而非口號

> V_d
> =
> ValueOrdering_d

不是：

> V_d
> =
> ListOfValues_d

SECTION

十一、作者表達自由

作者能否：

- 使用多種慣用法；
- 建立 DSL；
- 使用 Metaprogramming；
- 控制記憶體；
- 連接外部系統；
- 建立新抽象。

SECTION

十二、讀者理解自由

讀者能否不依賴：

- 隱藏 Context；
- 私有框架知識；
- 過多特殊符號；
- 全域搜尋；
- 執行後才知道的語義；

  而理解程式。

SECTION

十三、實作者自由

Compiler／Runtime 能否：

- 最佳化；
- 更換表示；
- 平行執行；
- 延遲求值；
- 使用不同 GC；
- 建立多實作。

SECTION

十四、使用者遷移自由

使用者能否：

- 保持舊程式；
- 逐步採用；
- 使用舊版；
- 由工具遷移；
- 避免被單一 Runtime 鎖定。

SECTION

十五、生態延展自由

Library、Framework 與領域社群能否：

- 建立新協定；
- 擴充類型；
- 形成 DSL；
- 發布套件；
- 替換工具。

SECTION

十六、治理參與自由

誰能：

- 提案；
- 討論；
- 審查；
- 裁決；

- 實作；
- 發布；
- 分叉。

SECTION

十七、自由可能互相衝突

> Freedom_(author)uparrow
> ⇒
> Vocabulary_(reader)uparrow

> Freedom_(implementation)uparrow
> ⇒
> Predictability_(machine-cost)downarrow

> Freedom_(migration)uparrow
> ⇒
> Freedom_(redesign)downarrow

SECTION

十八、限制也可能創造自由

- 靜態限制可創造重構自由；
- 共同 Style 可創造跨團隊閱讀自由；
- 不可變值可創造並行分享自由；
- Edition 可創造語言演化自由；
- 明確 Unsafe 邊界可創造安全核心內的高階自由。

SECTION

十九、複雜度不只有一種

> cal-C
> =
> (
> C_(problem),
> C_(language),
> C_(author),
> C_(reader),
> C_(compiler),
> C_(runtime),
> C_(tool),
> C_(library),
> C_(governance),
> C_(migration)
>)

SECTION

二十、本質與偶發

部分複雜度來自問題本身：

- 並行；
- 分散式失敗；
- 資源生命週期；
- 相容；
- 安全；
- 時間；

- 不確定性。

  部分來自歷史表示、工具不足或糟糕介面。

SECTION

二十一、複雜度可以真正刪除

若一項負擔只是：

- 重複樣板；
- 無必要轉換；
- 偶發語法差異；
- 人工記帳；
- 可由工具可靠推導的資訊；

  它可以被真正消除或攤銷。

  因此 PLDST 不主張：

> Complexity_(total)=永恆常數

SECTION

二十二、剩餘負擔仍需追蹤

設計宣稱「簡單」時，至少檢查：

刪除了什麼？
轉移了什麼？
新增了什麼？
誰支付？
何時支付？
是否可見？
規模增長後是否仍成立？

SECTION

二十三、九種複雜度操作

Eliminate
Compress
Encapsulate
Transfer
Defer
Amortize
Duplicate
Hide
Institutionalize

SECTION

二十四、垃圾回收例子

垃圾回收：

- 降低作者手工釋放責任；
- 增加 Runtime、延遲、記憶體與實作者負擔；
- 可能提高安全及生產力；
- 不表示所有資源生命週期都被處理。

SECTION

二十五、型別推導例子

型別推導：

- 減少標註；
- 增加 Compiler 推理；
- 增加診斷與工具責任；
- 可能使公開介面受實作細節影響。

SECTION

二十六、零額外成本例子

零額外成本通常約束：

- Runtime；
- Code size；
- Data representation；

  不自動保證：

- 編譯時間；
- 規格大小；
- 錯誤訊息；
- 學習成本；
- 委員會成本；

  也為零。

SECTION

二十七、責任不是「誰犯錯」

責任包含：

> cal-R
> =
> (
> Prevention,
> Detection,
> Localization,
> Containment,
> Handling,

SECTION

二十八、責任主體

使用者
語言規則
型別系統
Compiler
Runtime
Library
Tool
Framework
部署平台
治理共同體

SECTION

二十九、控制—責任對齊

> Responsibility(a)
> ≤
> Control(a)
> +
> Observability(a)
> +
> Capability(a)

若沒有控制、資訊與能力，責任便不應完全交給該主體。

SECTION

三十、Compiler 前移責任

所有權、型別、效果、可空性與資料流分析可把部分錯誤前移。

代價是：

- 規則；

- 診斷；
- 編譯時間；
- 標註；
- 學習；
- Escape hatch。

SECTION

三十一、Runtime 承擔責任

GC、Exception、Dynamic dispatch、JIT、Actor supervision 會把部分責任放到執行期。

其優勢是：

- 動態資訊；
- 彈性；
- 恢復能力。

其代價是：

- 延遲；
- 部署複雜度；
- 執行期失敗；
- 可預測性。

SECTION

三十二、工具承擔責任

Type checker、Linter、IDE、Formatter、Migration tool 可在不完全改變 Runtime 語義的情況下提供保證。

工具不是語言之外的中性附加物，而是實際使用契約的一部分。

SECTION

三十三、顯式不是越多越好

過度顯式會造成：

- 樣板；
- 重複；
- 視覺噪音；
- 資訊漂移；
- 維護負擔。

SECTION

三十四、隱式不是越少越好

可靠推導可以：

- 減少錯誤；
- 保持意圖；
- 提升流暢；
- 讓程式聚焦問題。

SECTION

三十五、關鍵是可恢復性

> ImplicitnessQuality
> =
> Recoverability
> +

SECTION

三十六、局部恢復與工具恢復

隱式資訊可由：

- 局部語法；
- 型別；
- IDE；
- Trace；
- Compiler message；
- Runtime inspection；

  恢復。

  不同設計者對工具依賴的容忍不同。

SECTION

三十七、公共 API 與局部實作

局部程式可容許更多推導；公共介面通常需要更高穩定性及可讀性。

所以顯式—隱式不是單一全域開關。

SECTION

三十八、機器模型不能完全消失

所有程式最終受：

- 記憶體；
- 延遲；

- 帶寬；
- Cache；
- 排程；
- 故障；
- 資料表示；
- 能源；

  限制。

SECTION

三十九、也不必全部寫在表面

Compiler、Runtime 與工具可以封裝大量機器細節。

問題不是「抽象是否隱藏機器」，而是：

> 隱藏到什麼程度、何時洩漏、誰能恢復、失敗時是否可解釋？

SECTION

四十、Ritchie 型透明度

C 的力量來自提供一個相對薄、可攜且可預測的系統層，而不是等同特定組合語言。

SECTION

四十一、Stroustrup 型抽象

高階抽象可存在，但：

- 不使用不支付；
- 使用時成本可接近手寫低階方案；

- 資料表示與系統互操作仍重要。

SECTION

四十二、Hejlsberg 型工具化

有些複雜度可由 Compiler 與 IDE 共同承擔，使使用者逐步獲得靜態證據，而不必一次重寫生態。

SECTION

四十三、人類可讀性也是成本模型

讀者時間、團隊協調、除錯、Code review 與錯誤修正，都是真實工程成本。

SECTION

四十四、語法簡單

表面規則少、符號少、例外少。

SECTION

四十五、核心簡單

少量原語具有高生成力。

SECTION

四十六、代數簡單

程式可由組合律、等式與轉換規則理解。

SECTION

四十七、表示簡單

資料、程式與語義使用統一表示。

SECTION

四十八、去交纏簡單

值、身分、狀態、時間、來源與執行策略保持可分離。

SECTION

四十九、實作簡單

Compiler、Runtime 或完整系統可由人理解和重建。

SECTION

五十、治理簡單

提案、裁決、實作、發布與接班權限清楚。

SECTION

五十一、Backus、McCarthy、Hickey

> 【Backus：代數轉換型簡單性;
> McCarthy：生成符號型簡單性;
> Hickey：去交纏結構型簡單性】

三者都重視函數與簡單，卻不是同一路線。

SECTION

五十二、以讀者為中心

Guido 式人本重視：

- 可讀；
- 明示；
- 慣例收斂；
- 團隊；
- 下一位維護者。

SECTION

五十三、以作者體驗為中心

Matz 式人本重視：

- 自然；
- 流動；
- 和諧；
- 創造；
- 程式設計者幸福。

SECTION

五十四、以異質求解者為中心

Wall 式人本重視：

- 多種慣用法；
- Context；
- 領域差異；
- 表達壓縮；
- 社群文化。

SECTION

五十五、人本不是單一友善度

> HumanCentered
> =
> ForWhom
> +
> WhichTask
> +
> WhichTimeScale
> +
> WhoPays

SECTION

五十六、三種自由

Python：共同理解的自由
Ruby：創造流動的自由
Perl：多元表達的自由

SECTION

五十七、安全不是禁止一切

安全是對特定錯誤建立可說明保證。

SECTION

五十八、自由不是任意位元操作

自由也包括：

- 建立高階抽象；
- 安全重構；

- 可靠並行；
- 連接外部世界；
- 在限制外建立受控 Escape hatch。

SECTION

五十九、安全核心與危險邊界

成熟設計常使用：

> SafeCore
> +
> ExplicitEscape

而不是：

> EverythingSafe
> 或
> EverythingUnrestricted

SECTION

六十、Escape hatch 的責任

逃生口應具有：

- 明確名稱；
- 小範圍；
- 文件；
- Code review；
- 測試；

- 責任轉移標記。

SECTION

六十一、相容性不是保守情緒

相容性保護：

- 程式；
- 資料；
- ABI；
- 套件；
- 教材；
- 技能；
- 部署；
- 使用者信任。

SECTION

六十二、相容性也是永久成本

> HistoricalChoice
> →
> CompatibilityObligation
> →
> CurrentComplexity

SECTION

六十三、設計一致性的誘惑

Clean slate 可以：

- 移除例外；
- 重整語義；
- 提升安全；
- 簡化教學。

但可能造成：

- 生態分裂；
- 遷移失敗；
- 使用者流失；
- 組織成本。

SECTION

六十四、相容性化石

現代語言保留某功能，不代表現代治理仍喜歡它。

它可能只是：

- 既有程式；
- 標準；
- ABI；
- 工具；
- 社群依賴；

形成的化石。

SECTION

六十五、遷移工具是設計機制

Edition、Deprecation、Formatter、Codemod、Type migration 會把破壞成本分期與攤銷。

SECTION

六十六、歷史成為共同作者

成熟語言可表示為：

> Language□
> =
> CurrentDesign
> +
> AccumulatedCommitments_(<t)

SECTION

六十七、創始者偏誤

語言史常把：

- 創始人；
- 共同設計者；
- Compiler team；
- 標準委員會；
- Framework；
- 使用者生態；

壓成一個名字。

SECTION

六十八、五種歸因

> Credit
> ≠
> Causality
> ≠
> Authority
> ≠
> Accountability
> ≠
> Maintenance

SECTION

六十九、提案者不等於裁決者

一項功能可能由：

- A 提出；
- B 設計；
- C 實作；
- D 批准；
- E 維護；
- F 形成生態慣例。

SECTION

七十、個人風格與制度風格

Guido 的 BDFL 風格不能直接代表 2018 年後所有 Python 決策。

Rust 的現代語言風格也不能全部歸因於 Graydon Hoare。

SECTION

七十一、人物個案的正確格式

每篇人物研究應同時處理：

- 直接決策；
- 共同作者；
- 實作；
- 組織；
- 治理；
- 生態；
- 後期變化。

SECTION

七十二、誰能改變語言

治理決定：

誰能提出？
誰能進入議程？
誰能要求證據？
誰能拒絕？
誰能合併？
誰能發布？
誰能接班？

SECTION

七十三、個人工作室

優勢：

- 一致；
- 快速；
- 可刪除；
- 整體實作。

風險：

- 單點；
- 不可擴張；
- 隱性知識；
- 接班。

SECTION

七十四、BDFL

> DistributedContribution
> +
> CentralFinalJudgment

它把探索分散，把最終性集中。

SECTION

七十五、RFC 聯邦

> OpenDeliberation
> +
> TeamResponsibility
> +
> DocumentedDecision

RFC 不等於全民投票。

SECTION

七十六、Steering Group

公開提案與小型專業裁決群體並存。

企業資源可能形成正式程序之外的實際議程權。

SECTION

七十七、標準委員會

多實作、國家與公司代表、正式 Paper 和共識，交換較高協調成本以取得長期產業承諾。

SECTION

七十八、治理不是道德排名

不同制度適合不同：

- 使用者規模；
- 風險；
- 實作數量；
- 相容責任；
- 資源；
- 創始者狀態。

SECTION

七十九、接班是設計問題

接班需要轉移：

- 品味；
- 權威；
- 文件；
- 資產；
- Release；
- 衝突處理；
- 社群信任。

SECTION

八十、設計主張需要何種證據

可使用：

- 原始論文；
- 語言報告；
- 設計 FAQ；
- PEP／RFC；
- Compiler commit；
- 口述歷史；
- 訪談；
- 標準文件；
- 實作結果；
- 後期回顧。

SECTION

八十一、第一手來源不是絕對真理

設計者可能：

- 事後合理化；
- 忽略協作者；
- 重寫歷史；
- 只描述目標而非結果。

SECTION

八十二、實作也是證據

如果正式哲學與實際 Compiler、Runtime、Library 長期不符，PLDST 必須記錄差距。

SECTION

八十三、拒絕也是資料

被拒提案最能顯示：

- 邊界；
- 優先序；
- 證據門檻；
- 相容責任。

SECTION

八十四、反證不是破壞人物形象

反證用來：

- 限定範圍；
- 區分時期；
- 修正歸因；
- 防止口號化；
- 發現自我修正。

SECTION

八十五、Backus：從實用編譯器到自我批判

Backus 先將機器最佳化負擔集中到 FORTRAN Compiler，後來又批判逐字、變數、賦值與狀態支配的馮紐曼風格。

其重要性不只是轉向函數式，而是展示：

> Designer
> can revise
> DesignerStyle

SECTION

八十六、McCarthy：小型符號核心

McCarthy 以 S-expression、遞迴、Lambda、apply 與 eval 建立生成性符號系統。

小核心可以形成巨大語言空間，也會產生 Runtime、Macro、方言與治理負擔。

SECTION

八十七、Kay：語言作為動態媒介

Alan Kay 的物件不是只為企業資料封裝，而服務：

- 訊息；
- 個人運算；
- 教育；
- 圖形環境；
- 可延展媒介。

SECTION

八十八、Wirth：可完整理解的系統

Wirth 的簡潔不是少寫字，而是讓語言、Compiler、OS 與教學形成可被完整掌握的系統。

SECTION

八十九、Ritchie：克制的機器抽象

C 提供足夠薄、足夠可攜、足夠可預測的系統語言層。

SECTION

九十、Stroustrup：抽象與既有世界共存

C++ 的核心不是「功能很多」，而是讓高階抽象與 C、硬體、工業生態和長期相容同時存在。

代價是規格、工具與學習複雜度。

SECTION

九十一、Guido：公共文本

Python 將可讀性、明示、慣例收斂與實用 Escape hatch 結合。

SECTION

九十二、Matz：認知環境

Ruby 願意讓內部系統更複雜，以換取程式設計者的自然表達與持續流動。

SECTION

九十三、Wall：異質性容納

Perl 接受 Context、多義性與多種慣用法，再以文化、文件、strict、warnings 及生態治理自由。

SECTION

九十四、Hejlsberg：語言—工具—平台整體

Turbo Pascal、Delphi、C#、TypeScript 顯示其穩定風格不是某種語法，而是降低真實開發者從編輯、檢查到部署的整體摩擦。

SECTION

九十五、Hickey：解除交纏

Clojure 把 Value、Identity、State、Reference 與 Time 分開，讓 Runtime 承擔 Persistent data 的結構複雜度。

SECTION

九十六、Hoare 與 Rust 共同體：制度工程

Rust 從個人原型變成由團隊、RFC、Compiler、Library、Edition 與治理共同維持的安全系統語言。

其設計風格不能再由單一人物概括。

SECTION

九十七、系統語言三角

Wirth、Ritchie、Stroustrup 分別代表：

可理解的完整系統
可攜的機器控制
相容約束下的高階抽象

SECTION

九十八、人本語言三角

Guido、Matz、Wall 分別保護：

讀者與團隊
正在編程的作者
異質領域與表達者

SECTION

九十九、簡單性三角

Backus、McCarthy、Hickey分別追求：

代數轉換
符號生成
責任分離

SECTION

一百、治理五型

個人工作室
BDFL
RFC 團隊聯邦
Steering 混合制
標準委員會

SECTION

一百零一、比較的真正價值

比較不是決定誰較好，而是顯示：

- 同一口號的不同含義；
- 同一功能的不同理由；
- 不同成本配置；
- 不同適用條件；

- 不可直接合併的價值衝突。

SECTION

一百零二、完整分析單位

> 【mathfrakD
> =
> (
> Actor,
> Decision,
> Context,
> Time,
> Values,
> Freedom,
> Complexity,
> Responsibility,
> Evidence,
> Governance,
> Outcome
>)】

SECTION

一百零三、風格向量

> Σ_(d,t)
> =
> (
> P,
> V,
> F,
> B,
> R,
> E,

其中 X 表示演化與修正模式。

SECTION

一百零四、語言狀態

> L□
> =
> Design□
> +
> Implementation□
> +
> Compatibility_(<t)
> +
> Ecosystem□
> +
> Governance□

SECTION

一百零五、設計結果

> Outcome
> =
> f(
> Style,
> Constraints,
> Implementation,
> Adoption,
> History,
> Institutions
>)

所以：

> SuccessfulLanguage
> not⇒
> EveryDecisionWasCorrect

SECTION

一百零六、代價配置契約

> 【LanguageDesign
> =
> Affordances
> +
> Prohibitions
> +
> CostModel
> +
> ResponsibilityModel
> +
> EvolutionConstitution】

SECTION

一百零七、軸只是索引

PLDST-027 的十八軸可協助：

- 比較；
- 搜尋；
- 視覺化；
- 聚合；

- AI 分析。

SECTION

一百零八、軸不是人物本體

> Profile
> ≠
> Person

SECTION

一百零九、數值不是精確真理

4.2 不表示風格可被自然測量到小數點。

必須同時顯示：

- Coverage；
- Confidence；
- Time；
- DDR count；
- Counterevidence。

SECTION

一百一十、矩陣與語料不可分離

> MatrixWithoutCorpus
> =
> Labeling

> CorpusWithoutMatrix
> =
> ArchiveWithoutComparison

SECTION

一百一十一、SKILL 是研究管線

> Request
> →
> Search
> →
> Evidence
> →
> DDR
> →
> Challenge
> →
> Assessment
> →
> Profile

SECTION

一百一十二、結構正確不等於史實正確

> SchemaValid
> not⇒
> HistoricallyTrue

SECTION

一百一十三、AI 預設只能生成 Candidate

高影響紀錄需要：

- 來源恢復；
- 歸因檢查；
- 時間檢查；
- 反證；
- 人類覆核。

SECTION

一百一十四、重新搜尋是必要規則

每篇新 PLDST 文章或新研究 Run 都應重新搜尋。

舊 Corpus 是起點，不是唯一依據。

SECTION

一百一十五、模擬不是身份

> Simulation
> ≠
> Identity
> ≠
> Prediction
> ≠
> Impersonation

SECTION

一百一十六、表面與決策

> SurfaceStyle=Rhetoric

> DecisionStyle
> =
> Values
> +
> Heuristics
> +
> Burden
> +
> Evidence
> +
> Governance

SECTION

一百一十七、混合不是平均

> Hybrid
> ≠
> Average

合理混合需要：

- 分層；
- Gate；
- Council；
- Constraint reviewer；
- Pareto set；
- 衝突公開。

SECTION

一百一十八、跨時代轉譯

使用歷史風格分析現代問題時，必須分開：

- 穩定價值；
- 歷史限制；
- 現代限制；
- 治理權變化；
- 不確定性。

SECTION

一百一十九、好模擬應更可追蹤

> GoodSimulation
> =
> MoreTraceableThanOrdinaryGeneration

SECTION

一百二十、設計前提卡

新語言應先寫：

主要使用者：
主要問題：
非目標：
機器成本：
相容邊界：
核心大小：
擴張位置：
錯誤責任：
治理方式：

SECTION

一百二十一、自由配置卡

作者可做什麼？
讀者可假設什麼？
Compiler 可改寫什麼？
Runtime 可延後什麼？
Library 可擴張什麼？
治理可拒絕什麼？

SECTION

一百二十二、複雜度配置卡

消除：
轉移：
封裝：
延後：
攤銷：
新增：
隱藏：
制度化：

SECTION

一百二十三、責任配置卡

誰預防？
誰偵測？
誰定位？
誰圍堵？
誰恢復？
誰負責遷移？
誰負責長期維護？

SECTION

一百二十四、證據卡

原型：
Benchmark：
使用案例：
反例：
多實作：

工具影響：
遷移成本：
未解問題：

SECTION

一百二十五、治理卡

誰可提案？
誰設定議程？
誰裁決？
誰實作？
誰發布？
誰接班？
如何申訴？
如何分叉？

SECTION

一百二十六、不可能無條件最大化所有自由

> max
> (
> AuthorFreedom,
> ReaderFreedom,
> ImplementationFreedom,
> CompatibilityFreedom,
> GovernanceFreedom
>)

通常受到共同限制。

SECTION

一百二十七、但也不是純零和

更好的 Compiler、資料結構、工具與制度可以同時改善多項維度。

因此 PLDST 不主張所有改進必須犧牲另一項同等價值。

SECTION

一百二十八、Pareto 前沿

更合理的問題是：

> 此設計是否在現有限制下支配其他方案，或位於可說明的 Pareto 前沿？

SECTION

一百二十九、代價應公開

不能只說：

更安全
更簡單
更自然
更快

還應說：

對誰？
相較何物？
在哪個尺度？
由誰支付？
是否可逆？

SECTION

一百三十、不是範式接力史

語言史不是：

命令式被物件導向取代
物件導向被函數式取代
動態被靜態取代

SECTION

一百三十一、是反覆重配負擔

同一問題會在不同年代被重新配置：

- 記憶體；
- 型別；
- 並行；
- 模組；
- 相容；
- 工具；
- 治理。

SECTION

一百三十二、舊設計不必然落後

某些歷史設計在原始條件下是高品質答案。

PLDST 避免以今日工具對歷史人物作廉價評判。

SECTION

一百三十三、新設計不必然進步

新功能可能只把：

- 複雜度；
- 錯誤；
- 相容成本；
- 工具負擔；

推到未來。

SECTION

一百三十四、語言史是制度史

成熟語言的演化由：

- 公司；
- 標準；
- Core team；
- 基金會；
- Package 生態；
- 教育；
- 部署平台；

  共同塑造。

SECTION

一百三十五、公開資料不完整

部分決策只存在：

- 私人信件；
- 未保存會議；
- 非正式對話；
- 已失效來源。

SECTION

一百三十六、人物與環境難以完全分離

設計者受：

- 硬體；
- 組織；

- 經濟；
- 同事；
- 既有語言；
- 使用者；

限制。

SECTION

一百三十七、評估者也有風格

研究者可能偏好：

- 極簡；
- 安全；
- 函數式；
- 動態；
- 系統透明。

因此必須保存：

- 軸定義；
- 來源；
- 反證；
- 編碼者；
- 信心；
- 分歧。

SECTION

一百三十八、第一批樣本不完整

第一批未完整覆蓋：

- ML／Haskell；
- Ada；
- Java；
- Go 創始團隊；
- Erlang；
- Lua；
- Scheme；
- Prolog；
- APL；
- SQL；
- GPU／Shader 語言；
- WebAssembly；
- AI 原生語言。

SECTION

一百三十九、風格不是因果全部

> Style
> ⊂
> CausalSystem

不是：

> Style
> =
> CausalSystem

SECTION

一百四十、第一部：方法論地基

PLDST-001 至 005 建立：

- 決策風格；
- 複雜度配置；
- 責任配置；
- 多主體歸因；
- 時間相位。

SECTION

一百四十一、第二部：核心衝突

PLDST-006 至 010 建立：

- 核心與擴張；
- 顯式與推導；
- 機器與人；
- 安全與自由；
- 一致性與相容。

SECTION

一百四十二、第三部：設計者個案

PLDST-011 至 022 用十二組個案驗證方法能穿越：

- 系統語言；
- 函數與符號；
- 物件；
- 動態語言；
- 工具平台；
- 安全共同體。

SECTION

一百四十三、第四部：跨案例比較

PLDST-023 至 026 證明 PLDST 不是人物傳記合集，而能比較：

- 技術現實；
- 人本價值；
- 簡單性；
- 治理。

SECTION

一百四十四、第五部：方法落地

PLDST-027 至 030 建立：

- 評估矩陣；

- 決策語料；
- SKILL；
- AI 模擬；
- 總論。

SECTION

一百四十五、理論成果

PLDST 已建立：

設計決策人格
複雜度配置論
控制責任配置論
時間相位
多主體歸因
自由—安全模型
核心—擴張模型
治理風格分類

SECTION

一百四十六、歷史成果

已對十二組設計者或共同體建立正式個案。

SECTION

一百四十七、比較成果

已建立四篇跨設計者比較，避免人物研究碎片化。

SECTION

一百四十八、工程成果

已建立：

- DDR Schema；
- 十八軸 Vocabulary；
- Corpus 規格；
- PLDST SKILL；
- Simulation Contract；
- Validator；
- Contract tests。

SECTION

一百四十九、成書成果

三十篇可以重組為：

- 方法；
- 衝突；
- 人物；
- 比較；
- AI 與未來。

SECTION

一百五十、不要問「此語言是否簡單」

應問：

哪一種簡單？
對誰簡單？
在哪一層簡單？
代價在哪裡？

SECTION

一百五十一、不要問「此語言是否自由」

應問：

誰的自由？
控制什麼？
限制誰？
是否有 Escape hatch？

SECTION

一百五十二、不要問「此語言是否安全」

應問：

防止哪一類錯誤？
在哪個階段？
由誰保證？
保證邊界在哪裡？

SECTION

一百五十三、不要問「此設計者是否務實」

應問：

接受哪種證據？
願意犧牲哪種純度？
對哪些歷史承諾負責？

SECTION

一百五十四、不要問「誰發明了這門語言」

應問：

誰提出？
誰設計？
誰實作？
誰裁決？
誰維護？
誰讓它成為生態？

SECTION

一百五十五、設計風格公式

> 【Σ_(d,t)
> =
> (
> ProblemFraming,
> ValueOrdering,
> FreedomAllocation,
> BurdenAllocation,
> Responsibility,
> Evidence,
> Governance,
> Evolution
>)】

SECTION

一百五十六、語言公式

> 【L⊠
> =
> Syntax
> +
> Semantics
> +
> Implementation
> +
> Library
> +
> Tooling
> +
> Compatibility
> +

SECTION

一百五十七、代價公式

> 【Cost
> =
> WhoPays
> +
> WhatIsPaid
> +
> WhenPaid
> +
> Visibility
> +
> FailureExternality】

SECTION

一百五十八、自由公式

> 【Freedom
> =
> AvailableActions
> +
> PredictableConsequences
> +
> RecoverableInformation
> +
> ExitOptions】

SECTION

一百五十九、良好配置

【GoodAllocation
=
CostAtCapableLayer
+
ResponsibilityWithControl
+
VisibleTradeoffs
+
BoundedFailure
+
RevisableGovernance】

SECTION

一百六十、語言設計不是消滅代價

設計的核心是：

> 將不可避免及新產生的代價，配置到最能承擔、最能重用、最能觀測且最不會造成不可接受外溢的位置。

SECTION

一百六十一、語言設計不是最大化自由

設計的核心是：

> 決定哪些自由應受保護、哪些自由需受限制，以及限制是否創造更高階、更長期的自由。

SECTION

一百六十二、語言設計不是創始者獨白

設計會逐步成為：

> Founder
> +
> CoDesigners
> +
> Implementers
> +
> Users
> +
> Institutions
> +
> History

SECTION

一百六十三、語言設計不是一次性作品

每個版本都重新回答：

- 什麼仍值得保護；
- 什麼已成化石；
- 什麼可以遷移；
- 什麼必須拒絕；
- 誰仍具有權力。

SECTION

一百六十四、PLDST 的總命題

> 【ProgrammingLanguageDesign
> =
> TheConstitutionalAllocation
> (
> Freedom,

SECTION

一百六十五、PLDST 不是排行榜

它不宣告：

- Wirth 優於 Stroustrup；
- Python 優於 Perl；
- 函數式優於命令式；
- 委員會優於 BDFL；
- 安全優於自由。

SECTION

一百六十六、PLDST 是比較語言

它讓研究者能精確說：

- 哪種價值被優先；
- 哪種成本被轉移；
- 哪種責任被前移；
- 哪種歷史被保護；
- 哪種證據被接受；
- 哪種權力被制度化。

SECTION

一百六十七、PLDST 是設計鏡子

研究歷史設計者的目的，不是複製他們，而是讓當代設計者看見自己的隱性取捨。

SECTION

一百六十八、PLDST 是 AI 的限制器

AI 可以生成大量設計方案。

PLDST 要求它同時回答：

來源？
代價？
反證？
時間？
歸因？
責任？
治理？

SECTION

一百六十九、第一批完成

> 【PLDST First Batch
> =
> 30/30】

SECTION

一百七十、最終封頂命題

> 程式語言設計的真正風格，不存在於設計者最常說出的口號，而存在於他面對無法同時滿足的要求時，反覆選擇讓誰自由、讓誰受限、讓誰承擔複雜度，又願意為哪些歷史與未來負責。

SECTION

第一部　方法論地基

- PLDST-001　程式語言設計師風格譜系：從語言特徵分類到設計決策人格
- PLDST-002　複雜度配置論：程式語言沒有消滅的複雜度去了哪裡？
- PLDST-003　控制責任配置論：使用者、編譯器、Runtime 與工具誰應承擔錯誤？
- PLDST-004　設計者共同體與制度：如何避免程式語言史的創始人歸因偏誤
- PLDST-005　風格的時間相位：設計師思想、語言演化與社群治理如何分離

SECTION

第二部　核心風格原型

- PLDST-006　極簡核心與功能擴張：語言應保持多小，又能成長到多大？
- PLDST-007　顯式控制與自動推導：設計者應要求使用者說多少，又替使用者猜多少？
- PLDST-008　機器效率與人類可讀性：成本模型應寫在語言表面，還是藏在編譯器之後？
- PLDST-009　安全約束與表達自由：語言應禁止多少錯誤，又應允許多少逃生？
- PLDST-010　設計一致性、相容性與社群演化：語言何時應堅持原則，何時應接受歷史？

SECTION

第三部　設計師與共同體個案

- PLDST-011　John Backus：從 FORTRAN 到函數級程式設計的自我反省
- PLDST-012　John McCarthy：極小核心、符號計算與語言可延展性
- PLDST-013　Alan Kay：物件、訊息與對物件導向的歷史誤讀
- PLDST-014　Niklaus Wirth：簡潔、教育與可實作性的設計倫理
- PLDST-015　Dennis Ritchie：可攜式系統語言、機器透明度與克制的抽象
- PLDST-016　Bjarne Stroustrup：零額外成本、多範式與相容性政治

- PLDST-017　Guido van Rossum：可讀性、實用主義與 BDFL 裁決
- PLDST-018　Yukihiro Matsumoto：程式設計者幸福、語言自然性與社群信任
- PLDST-019　Larry Wall：語言多義性、後現代實用主義與社群文化
- PLDST-020　Anders Hejlsberg：工具驅動設計、漸進型別與平台折衷
- PLDST-021　Rich Hickey：價值、身分與簡單性的分離
- PLDST-022　Graydon Hoare 與 Rust 共同體：安全系統語言如何從個人原型轉為制度工程

SECTION

第四部　跨設計師比較

- PLDST-023　Wirth、Ritchie 與 Stroustrup：簡潔、機器控制與相容性之間的三種系統語言倫理
- PLDST-024　Guido、Matz 與 Larry Wall：可讀性、幸福與多義性之間的三種人本語言設計
- PLDST-025　Backus、McCarthy 與 Hickey：函數、符號與簡單性的不同道路
- PLDST-026　個人設計者、仁慈獨裁者與 RFC 制度：語言治理風格比較

SECTION

第五部　方法落地與封頂

- PLDST-027　PLDST 評估矩陣與設計決策語料庫規格
- PLDST-028　PLDST SKILL 技術規格：資料搜尋、決策抽取與風格判定
- PLDST-029　AI 模擬程式語言設計師風格：混合、轉譯與失真問題
- PLDST-030　程式語言設計師風格譜系總論：設計自由、複雜度與代價

第一部：5 篇
第二部：5 篇
第三部：12 篇
第四部：4 篇
第五部：4 篇

合計：30 篇

> 5+5+12+4+4=30

SECTION

卷一　為何設計者值得被研究

PLDST-001 至 005。

SECTION

卷二　語言設計的五場根本衝突

PLDST-006 至 010。

SECTION

卷三　設計語言的人

PLDST-011 至 022。

SECTION

卷四　不同道路的正面比較

PLDST-023 至 026。

SECTION

卷五　從歷史研究到 AI 設計工具

PLDST-027 至 030。

設計風格

> DesignStyle
> =
> RepeatedTradeoffRules
> +
> BurdenAllocation
> +
> ResponsibilityAllocation
> +
> EvidenceStandard
> +
> Governance
> +
> TemporalRevision

完整決策

> DDR
> =
> Actor
> +
> Time
> +
> Problem
> +
> Constraints
> +
> Options
> +
> Decision
> +
> Rationale
> +
> Implementation
> +
> Consequences

語言狀態

> L□
> =
> Design□
> +
> Implementation□
> +
> Compatibility_(<t)
> +
> Ecosystem□
> +
> Governance□

良好配置

> GoodAllocation
> =
> CostAtCapableLayer
> +
> ResponsibilityWithControl
> +
> VisibleTradeoffs
> +
> BoundedFailure
> +
> RevisableGovernance

最終總論

> ProgrammingLanguageDesign
> =
> TheConstitutionalAllocation
> (

[R1] John Backus, “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,” Communications of the ACM, 1978.

— 函數級程式設計、馮紐曼風格、程式代數及設計自我批判。

[R2] Dennis M. Ritchie, “The Development of the C Language,” HOPL II／ACM.

— C、B、BCPL、Unix、PDP-11、實作與共同設計歷史。

[R3] Bjarne Stroustrup, The Design and Evolution of C++, official historical description and ACM／HOPL materials.

— 真實限制、被拒方案、相容性、抽象與語言演化。

[R4] ACM History of Programming Languages proceedings.

— 以設計者、歷史限制、實作及演化研究程式語言。

[R5] Python Enhancement Proposals, PEP 13: Python Language Governance.

— 五人 Steering Council、廣泛但克制的權力、委派、選舉與接班。

[R6] Rust official Governance and Leadership Council documentation.

— RFC、公開審議、領域團隊、Purview 與跨團隊協調。

[R7] Swift.org, Language Steering Group and Swift Evolution documentation.

— 公開語言演化、Steering Group、專案權力與社群參與。

[R8] ISO C++／WG21, Practices and Procedures and committee participation materials.

— 書面提案、工作小組、多實作與共識標準化。

[R9] PLDST-001 至 PLDST-029。

— 本文的直接理論、個案、比較及工程基礎。

資料查核日期： 2026-07-30。

SECTION

F.1 「複雜度配置」不是物理守恆律

本文沒有主張所有複雜度永遠等量保存。

更好的抽象、演算法、工具與介面可以真正消除偶發負擔。

PLDST 要求追蹤的是：

- 未消除的本質負擔；
- 被轉移的負擔；
- 新加入的負擔；
- 延後至未來的負擔。

SECTION

F.2 「自由衝突」不是永恆零和

工具、分析、資料結構與制度創新可以同時改善多種自由。

本文只否定「所有自由與保證可以無條件無成本最大化」。

SECTION

F.3 設計者風格不是心理診斷

PLDST 所稱「設計決策人格」是歷史決策模式，不推論私人心理本質、政治立場或道德人格。

SECTION

F.4 成功不能反證所有代價

語言廣泛採用可能來自：

- 時機；
- 公司；
- 平台；
- 教育；

- Library；
- 既有生態；
- 市場。

成功是結果資料，不是所有設計哲學的自動證明。

SECTION

F.5 當代治理已與創始期分離

截至 2026-07-30：

- Python 正式治理為 Steering Council；
- Rust 由多個領域團隊與 Leadership Council 構成；
- Swift 的公開演化由 Language Steering Group 引導；
- C++ 正式標準由 WG21 程序形成。

本文沒有把現代決策全部歸給創始者。

SECTION

F.6 第一批不是終極全集

三十篇完成的是第一個可運作閉環：

理論
→ 個案
→ 比較
→ 語料
→ SKILL
→ 模擬
→ 總論

它不是對所有語言與設計者的完整覆蓋。

PLDST 第一批 30 篇已完成以下閉環：

> 【Question
> →
> Theory
> →
> CaseStudy
> →
> Comparison
> →
> Corpus
> →
> Skill
> →
> Simulation
> →
> GeneralTheory】

系列第一批至此正式封頂。