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

A Genealogy of Programming Language Designer Styles: From Language-Feature Classification to Decision-Style Signatures

v1.0 / 2026-07-30 / Neo.K / 公開版／方法論奠基論文

https://thisoneisneok.com/html/papers/pldst-001.html

英文名稱： A Genealogy of Programming Language Designer Styles: From Language-Feature Classification to Decision-Style Signatures

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-001

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／方法論奠基論文

SECTION

摘要

現有程式語言分類通常以範式、型別系統、執行模型、語法形式、抽象層級或應用領域為核心。這些分類能回答「一種語言具有哪些特徵」，卻不一定能回答另一個問題：

> 為何不同程式語言設計者，在面對相似問題時，會反覆選擇不同的取捨方式？

Niklaus Wirth 將設計簡潔視為 Modula-2 與 Oberon 最重要的指導原則；Guido van Rossum 在回顧 Python 時，同時強調可讀性、降低不必要的表達變異、對 Unix／C 生態的開放性，以及針對常見情境的工程調校；Go 的設計文件把大型組織中的建置速度、依賴管理、閱讀、維護與團隊生產力置於語言研究新穎性之前；Ruby 則公開接受「外表自然、內部複雜」的複雜度配置；C++ 的演化長期受相容性、一般化機制、硬體映射

本文提出 程式語言設計師風格譜系（Programming Language Designer Style Taxonomy, PLDST）。

PLDST 不將設計者進行心理人格診斷，也不把語言成敗簡化為個人天才或缺陷；它把「設計決策人格」定義為：

> 在特定歷史、技術、使用者與治理條件下，設計者或設計共同體對問題 framing、複雜度配置、責任分配、相容性、抽象、可讀性、安全與演化方式所呈現的可重複決策模式。

本文建立六項方法論核心：

- 區分語言特徵、設計哲學、設計者風格、制度風格與生態結果；
- 以「設計決策語料」而非單一名言作為分析單位；
- 建立原始來源、決策文件、語言構造、治理記錄與回顧結果的證據階層；
- 以問題定位、價值優先序、複雜度配置、責任邊界、演化模式與治理方式構成風格描述；
- 將風格視為有時間相位、情境依賴與信心區間的向量，而非固定類型；
- 提供可轉譯為個案研究、比較研究、AI 分析 SKILL 與書籍章節的標準輸出格式。

本文的核心命題是：

> 【程式語言設計風格
> ≠
> 語言特徵總和】

更精確地說：

> 【設計風格
> =
> 反覆出現的取捨規則
> +
> 複雜度配置
> +
> 責任配置
> +

關鍵詞： 程式語言設計、設計師風格、PLDST、設計決策、複雜度配置、語言史、HOPL、認知維度、語言治理

SECTION

一、為何「語言分類」不等於「設計師分類」

程式語言可以依多種方式分類：

- 命令式／函數式／邏輯式；
- 靜態型別／動態型別；
- 編譯式／直譯式；
- 系統語言／腳本語言／領域語言；
- 物件導向／資料導向／事件導向；
- 純函數式／多範式；
- 記憶體安全／非記憶體安全；
- 垃圾回收／手動管理／所有權管理。

這些分類描述的是語言構造或使用模式。它們通常回答：

> What does the language provide?

PLDST 要回答的是：

> Why did the designer repeatedly choose this kind of solution?

兩種語言可能都採靜態型別，但背後理由完全不同：

- 為了形式化推理；
- 為了執行效率；
- 為了工具與重構；

- 為了大型團隊溝通；
- 為了排除特定錯誤；
- 為了更明確的介面契約。

  反之，同一位設計者也可能在不同語言或不同時期採用不同機制，卻維持相似的深層取捨規則。

  因此：

> Feature Similarity
> not⇒
> Style Similarity

以及：

> Feature Difference
> not⇒
> Style Difference

SECTION

二、從「語言作品」回到「決策者」

語言是設計結果，但設計風格隱藏在下列問題中：

- 設計者認為最主要的問題是什麼？
- 他把誰視為主要使用者？
- 他把複雜度移到哪裡？
- 他願意讓使用者承擔多少責任？
- 他偏好禁止錯誤、揭露錯誤，還是容許恢復？
- 他是否接受多種做法共存？
- 他如何理解效能、抽象與硬體？

- 他如何處理向後相容？
- 他如何處理既有生態？
- 他把語言演化視為個人創作、委員會工作，還是社群治理？

這些問題共同構成「設計決策人格」。

SECTION

三、「人格」不是心理診斷

本文使用「設計決策人格」時，指的是穩定的決策模式，不是：

- 臨床心理人格；
- 私生活性格；
- 道德評價；
- 智力高低；
- 對設計者動機的讀心。

其英文可更保守地表述為：

> Decision-Style Signature

亦即：

> Σ_d
> =
> 設計者d在可觀察決策中的風格簽名

分析對象是公開決策，而不是設計者不可觀察的內心。

SECTION

四、範式與語言特徵分類

範式分類能揭示計算模型、抽象與程式結構，但它更接近「語言如何表達計算」，不直接描述「設計者如何選擇」。

同一種函數式語言可以：

- 極度純粹；
- 工程實用；
- 強型別；
- 動態型別；
- 研究導向；
- 商業整合導向。

因此，範式是 PLDST 的輸入，不是最終分類。

SECTION

五、語言史與 HOPL

ACM SIGPLAN 的 HOPL 傳統重視：

- 語言早期歷史；
- 後續演化；
- 語言特徵與概念；
- 語言家族；
- 設計、實作、標準化與社會影響。

HOPL-IV 的徵稿與寫作指引尤其強調技術準確、歷史完整、設計動機、演化與跨語言脈絡。[R9]

PLDST 接受 HOPL 的歷史方法，但分析焦點不同：

- HOPL 主要重建一種語言如何形成；
- PLDST 進一步抽取跨決策、跨語言、跨時期的風格模式。

SECTION

六、認知維度

Green 與 Petre 的 Cognitive Dimensions of Notations 提供一套討論記號系統與使用者關係的詞彙，例如：

- viscosity；
- visibility；
- juxtaposability；
- hidden dependencies；
- premature commitment；
- secondary notation；
- closeness of mapping。

該框架明確將這些維度視為討論與權衡工具，而非單一最佳設計清單。[R8]

PLDST 吸收其精神：

> 設計維度之間通常存在權衡，不能把所有維度同時最大化。

但 Cognitive Dimensions 主要評估記號與使用活動；PLDST 則評估設計者如何反覆配置這些權衡。

SECTION

七、人本與跨學科語言設計

Coblenz、Aldrich、Myers 與 Sunshine 主張，語言設計應依使用者背景、目標應用與領域需要選擇品質屬性，並結合形式、觀察性與人本方法，而不是假設一組屬性適用所有語言。[R10]

這對 PLDST 有兩個重要限制：

- 設計風格不能脫離目標使用者與目標場景；
- 某個取捨在一個情境中合理，不代表在所有情境中都合理。

SECTION

八、目前的研究缺口

目前已有：

- 語言範式分類；
- 語法與特徵分類；
- 記號可用性框架；
- 語言歷史；
- 設計原則；
- 治理流程；
- 個別設計者訪談。

但仍缺少一套統一方法，能同時回答：

- 如何從原始資料抽取設計風格？
- 如何避免把語言結果全部歸因於創始人？
- 如何處理設計者思想隨時間變化？
- 如何比較個人設計、共同設計與委員會設計？
- 如何把優點與代價放在相同框架中？
- 如何將分析轉為可重複執行的 SKILL？

PLDST 即針對此缺口。

SECTION

九、語言特徵

語言特徵是可直接觀察的構造，例如：

- 類別；
- 模式匹配；
- 所有權；
- 巨集；
- 泛型；
- 垃圾回收；
- 例外；
- 協程；
- 型別推導。

特徵回答：

> 語言具有什麼機制？

SECTION

十、設計原則

設計原則是對取捨的明確聲明，例如：

- 可讀性重要；
- 簡潔優先；
- 零額外成本；
- 實用勝於純粹；
- 避免猜測；
- 相容性優先；
- 自然性優先。

原則回答：

> 設計者公開宣稱什麼重要？
>
> 但原則不一定等於實際決策，因為原則可能：

- 彼此衝突；
- 只在特定時期有效；
- 是社群回顧後的總結；
- 是口號而非可操作規則。

SECTION

十一、設計決策

設計決策是：

> q
> =
> (
> p,
> c,
> A,
> x,
> r,
> b,
> o
>)

其中：

- p：問題；
- c：歷史與工程情境；

- A：可選方案集合；
- x：實際選擇；
- r：公開理由；
- b：複雜度與責任配置；
- o：後續結果。

設計風格必須從多個 q 中歸納，不能只從單一功能或名言判斷。

SECTION

十二、個人風格

個人風格是可合理歸因於特定設計者的重複決策模式。

例如：

- 在多次決策中反覆縮小核心；
- 反覆選擇顯式規則；
- 反覆保護相容性；
- 反覆把複雜度移給編譯器；
- 反覆偏好使用者自然表達。

SECTION

十三、制度風格

語言進入社群、基金會、委員會或標準組織後，設計風格可能由制度產生。

Rust 的 RFC 流程要求重大改變進入公開設計與共識程序；此時某項決策未必能簡化為某一位創始人的個人偏好。[R7]

制度風格包括：

- 決策透明度；
- 共識要求；
- 穩定性門檻；
- 提案格式；
- 實驗機制；
- 棄用政策；
- 利害關係人代表方式。

SECTION

十四、生態結果

一種語言最後呈現的樣貌，也會受下列因素影響：

- 實作者；
- 標準委員會；
- 程式庫；
- 公司策略；
- IDE；
- 套件管理；
- 教育採用；
- 使用者慣例；
- 向後相容壓力。

因此：

> Observed Language
> =
> Initial Design

不可將今日整個語言的所有特徵都歸於創始設計者。

SECTION

十五、Designer Decision Corpus

PLDST 為每位設計者或設計共同體建立：

> Designer Decision Corpus（DDC）
>
> DDC 包含：

- 原始語言規格；
- 設計論文；
- HOPL 回顧；
- 設計者文章；
- 訪談與演講；
- 提案與拒絕記錄；
- 郵件列表；
- Issue／RFC／PEP；
- 實作限制；
- 語言演化結果。

SECTION

十六、最小決策記錄

每一筆決策至少記錄：

```
decision_id
designer_or_body
language
period
```

problem
target_users
target_domain
constraints
alternatives
selected_choice
stated_rationale
inferred_rationale
complexity_allocation
responsibility_allocation
compatibility_cost
outcome
sources
attribution_confidence

SECTION

十七、不能只分析成功功能

只看成功功能會產生倖存者偏差。

DDC 還應包含：

- 被拒絕的提案；
- 被撤回的功能；
- 設計者後悔的決策；
- 因實作成本而放棄的方案；
- 因相容性保留的缺陷；
- 社群繞過原設計的做法。

「不加入什麼」往往比「加入什麼」更能揭示風格。

SECTION

十八、E1：設計者直接聲明

例如：

- 設計論文；
- 官方演講；
- 原始訪談；
- 書籍；
- HOPL 回顧。

優點：

- 能直接知道設計者如何解釋決策。

限制：

- 回顧可能重構過去；
- 設計者可能簡化歷史；
- 原則聲明未必與全部實作一致。

SECTION

十九、E2：正式決策文件

例如：

- PEP；
- RFC；
- 標準提案；
- 設計備忘錄；
- Issue 決議。

E2 能顯示：

- 問題；

- 替代方案；
- 反對理由；
- 相容性；
- 教學與遷移成本；
- 最終決策。

SECTION

二十、E3：語言與工具構造

實際語言設計可以驗證公開原則是否反覆落實。

例如：

- 語法是否強制統一；
- 型別錯誤在哪個階段發現；
- 抽象是否具有執行成本；
- 是否有多種等價構造；
- 工具是否承擔複雜度。

SECTION

二十一、E4：治理與演化紀錄

用於判斷：

- 個人風格是否已轉為制度風格；
- 誰具有最後決定權；
- 是否強調共識；
- 是否保護相容性；

- 是否容許實驗；
- 是否有穩定化程序。

SECTION

二十二、E5：次級研究與社群解讀

包括：

- 學術分析；
- 技術史；
- 可靠評論；
- 使用者研究。

E5 可以補充結果與影響，但不能取代 E1–E4 來斷言設計者本人動機。

SECTION

二十三、證據權重

對每項風格推論 h，可表示為：

> Conf(h)
> =
> f(
> Source,
> Attribution,
> Recurrence,
> Consistency,
> Context
>)

其中：

- Source：來源品質；
- Attribution：能否歸因於該設計者；
- Recurrence：是否多次出現；
- Consistency：言論與決策是否一致；
- Context：是否充分考慮時代與限制。

輸出不應只給分數，還要標示：

高信心
中信心
低信心
證據不足

SECTION

二十四、第一層：設計情境

任何風格分析先記錄：

> C_d
> =
> (
> Era,
> Hardware,
> Audience,
> Domain,
> Host,
> Economy,
> Governance
>)

包括：

- 設計年代；
- 硬體與作業系統；

- 目標使用者；
- 目標應用；
- 既有宿主生態；
- 經濟與組織條件；
- 決策制度。

脫離情境的評價容易形成後見之明偏誤。

SECTION

二十五、第二層：問題定位

設計者主要把問題定位在哪裡？

P1　機器問題

- 硬體控制；
- 記憶體；
- 效能；
- 可預測性；
- 編譯。

P2　程式問題

- 抽象；
- 組合；
- 模組；
- 正確性；
- 重用。

P3　使用者問題

- 可讀性；
- 可學性；
- 表達自然；
- 錯誤理解；
- 認知負擔。

P4　組織問題

- 建置速度；
- 大型團隊；
- 維護；
- 依賴；
- 標準化。

P5　領域問題

- 科學計算；
- 商業資料；
- 分散式；
- 教育；
- Web；
- 嵌入式。

設計者通常同時關注多個問題，但具有主要重心。

SECTION

二十六、第三層：價值優先序

PLDST 不採單一總分，而使用多軸優先序。

V1　語義經濟

小型核心 ←——————→ 特徵多元

V2　顯式性

顯式宣告 ←——————→ 推導與慣例

V3　安全約束

信任使用者 ←——————→ 系統預防

V4　機器透明度

直接硬體映射 ←——————→ Runtime 中介

V5　可讀一致性

單一／規範方式 ←——————→ 多種表達方式

V6　擴展開放性

封閉正交核心 ←——————→ 巨集／元編程開放

V7　一般性

領域專用 ←——————→ 通用機制

V8　相容性

概念修正優先 ←——————→ 向後相容優先

V9　工具中心性

語言自行承擔 ←——————→ 編譯器／IDE／工具承擔

V10　生態整合

自足系統 ←——————→ 宿主與既有生態整合

SECTION

二十七、第四層：複雜度配置

語言設計經常不是消除全部複雜度，而是重新配置複雜度。

本文提出啟發式複雜度預算：

> B_C
> =
> (
> C_(surface),
> C_(semantics),
> C_(compiler),
> C_(runtime),
> C_(library),
> C_(tooling),
> C_(user),
> C_(governance)
>)

分別表示：

- 表面語法；
- 語義規則；
- 編譯器；
- 運行時；
- 程式庫；
- 工具；
- 使用者；
- 治理。

這不是嚴格守恆定律。好的設計確實可以透過統一概念降低總複雜度；但在多數具體取捨中，降低某一層的成本會增加另一層的負擔。

例如：

- 型別推導降低表面冗長，但增加編譯器與錯誤解釋負擔；
- 垃圾回收降低使用者記憶體管理負擔，但增加 Runtime 成本；
- 強制格式降低風格差異，但限制使用者自由；
- 元編程增加表達力，但提高工具與理解成本；
- 向後相容降低遷移成本，但提高語言與規格累積複雜度。

SECTION

二十八、第五層：責任配置

設計者如何分配責任？

> B_R
> =
> (
> R_(programmer),
> R_(compiler),
> R_(runtime),
> R_(library),
> R_(tool),
> R_(organization)
>)

關鍵問題：

- 錯誤應由誰預防？
- 資源應由誰管理？
- 程式風格應由誰統一？

- 相容性成本由誰承擔？
- 效能調校由誰承擔？
- 安全政策由誰承擔？

設計風格常常就是責任配置哲學。

SECTION

二十九、第六層：演化與治理

E1　演化節奏

重新設計 ←——————→ 漸進演化

E2　決策中心

單一設計者 ←——————→ 委員會／社群

E3　實驗方式

先設計後發布 ←——————→ 實驗、回饋、穩定化

E4　相容性政策

允許斷裂 ←——————→ 長期相容

E5　標準化強度

實作主導 ←——————→ 規格／標準主導

SECTION

三十、形式化表示

對設計者 d 在時期 t 的設計決策集合：

$$
Q_{d,t} = \{q_{\square}, q_{\square}, \ldots, q_{\square}\}
$$

每個決策經特徵映射：

$$
\phi(q_i)
=
(
P_i,
V_i,
B_{C_i},
B_{R_i},
E_i
)
$$

風格簽名為：

$$
\Sigma_{d,t}
=
\mathrm{Aggregate}
(
\{
w_i \phi(q_i)
\}_{i=1}^{n}
)
$$

其中 w_i 依證據品質、歸因、重複性與情境完整度調整。

SECTION

三十一、風格不是單一平均值

如果設計者在不同時期發生明顯轉變，應表示為：

$$
\Sigma_d
=
\{
\Sigma_{d,t_1},
\Sigma_{d,t_2},
\dots
$$

例如：

- 創始期；
- 生態擴張期；
- 標準化期；
- 社群治理期；
- 退休後回顧期。

不可將數十年演化壓成單一靜態分數。

SECTION

三十二、矛盾不是錯誤

設計原則本來可能存在張力：

- 簡潔與表達力；
- 相容性與修正；
- 安全與靈活；
- 效率與抽象；
- 自然性與規則一致；
- 個人願景與社群共識。

PLDST 應記錄：

> 設計者如何排序衝突原則。

真正的風格往往出現在兩個好目標不能同時滿足時。

SECTION

三十三、原型不是互斥類別

以下原型是高維空間中的參考點，而不是八個箱子。設計者可以同時接近多個原型。

SECTION

三十四、語義極簡建築師

主要特徵：

- 小型核心；
- 概念經濟；
- 正交性；
- 少量規則；
- 實作可理解。

優勢：

- 易於推理；
- 規格清楚；
- 教學與實作成本較低。

風險：

- 實務便利性移至程式庫或使用者；
- 過度排斥必要特化；
- 生態功能成長較慢。

SECTION

三十五、實用整合者

主要特徵：

- 接受不完美現實；

- 重視既有系統；
- 漸進採用；
- 常見情境最佳化；
- 生態互操作。

優勢：

- 採用阻力低；
- 能進入真實工作流；
- 工程價值快。

風險：

- 例外累積；
- 歷史包袱；
- 語義純度下降。

SECTION

三十六、認知工效設計者

主要特徵：

- 可讀性；
- 易學性；
- 降低選擇；
- 錯誤可理解；
- 語法自然。

優勢：

- 降低入門與維護成本；
- 團隊風格較一致。

風險：

- 內部實作複雜；
- 隱式規則增加；
- 對專家限制過多。

SECTION

三十七、機器現實主義者

主要特徵：

- 硬體映射；
- 可預測成本；
- 記憶體布局；
- 零額外成本；
- 避免隱藏 Runtime。

優勢：

- 適合系統與資源受限環境；
- 效能模型清楚。

風險：

- 使用者責任高；
- 語言與硬體細節耦合；
- 安全與易用性成本上升。

SECTION

三十八、安全邊界建築師

主要特徵：

- 型別與效果；
- 所有權；
- 能力；
- 編譯期排除；
- 明確不變量。

優勢：

- 降低高代價錯誤；
- 大型系統契約清楚。

風險：

- 表達摩擦；
- 學習曲線；
- 合法程式可能難以表達；
- 錯誤訊息壓力大。

SECTION

三十九、表達力擴張者

主要特徵：

- 多範式；
- 巨集；

- 元編程；
- DSL；
- 高階抽象。

優勢：

- 能建立領域語言；
- 壓縮重複；
- 讓語言成為語言建構工具。

風險：

- 團隊風格碎片化；
- 工具難以理解；
- 程式局部語義不透明。

SECTION

四十、組織工程設計者

主要特徵：

- 大型團隊；
- 建置速度；
- 依賴管理；
- 維護與閱讀；
- 工具鏈一致性。

優勢：

- 適合大規模生產；

- 降低協作成本。

風險：

- 個人表達力受限；
- 對小型或研究問題未必最佳；
- 設計可能被組織情境綁定。

SECTION

四十一、制度演化設計者

主要特徵：

- 提案程序；
- 相容性；
- 多方利害關係人；
- 穩定化；
- 長期治理。

優勢：

- 語言可持續；
- 決策可追溯；
- 降低個人任意性。

風險：

- 決策緩慢；
- 折衷累積；
- 願景稀釋；

- 舊設計難以修正。

SECTION

四十二、Wirth：簡潔作為生成原則

Wirth 在 Modula-2 與 Oberon 的歷史回顧中，明確將設計簡潔稱為最重要的指導原則，並把概念清晰、特徵經濟、實作效率與可靠性視為其結果。[R1]

PLDST 不應只把他標記為「極簡派」，而應進一步記錄：

- 簡潔是原因還是結果？
- 他願意犧牲哪些便利？
- 教育、實作與語義清晰如何互相支撐？
- 不同年代的硬體環境如何影響其小型系統觀？

初步風格推論：

高語義經濟
高概念一致性
低特徵冗餘
高實作可理解性
教育與工程雙重導向

SECTION

四十三、Python：可讀性與開放實用的混合

Guido van Rossum 回顧 Python 時指出：

- Python 是 ABC 的後裔，但面向 Unix／C 使用者；
- ABC 的封閉系統與難以擴充是他不願重複的錯誤；
- Python 重視常見情境的效率；
- 縮排與降低格式自由度有助於可讀性與風格一致。[R2]

PEP 20 由 Tim Peters 撰寫，是對 Python 設計原則的濃縮性表達；它是 Informational PEP，不是完整、規範性的語言規格。[R3]

這顯示 Python 風格不是單純「簡單」：

可讀一致性
實用性
生態開放
降低不必要變異
接受工程折衷
它同時避免 ABC 的封閉純粹，又保留 ABC 對清晰表達的重視。

SECTION

四十四、Go：語言設計服務於組織工程

Go 的官方設計回顧明確表示，Go 主要是為了解決 Google 大型軟體工程的生產力、建置、依賴、閱讀、除錯與維護問題，而不是追求程式語言研究上的突破。[R4]

該文也公開說明垃圾回收的複雜度配置：

- 使用者獲得更簡單的記憶體管理；
- 成本主要移至語言實作者與 Runtime；
- 同時保留資料布局控制以降低部分成本。[R4]

PLDST 初步推論：

高組織工程導向
高閱讀與維護優先
偏好有限特徵集合
願將複雜度移給實作者
保留部分機器模型透明度

SECTION

四十五、Ruby：自然性優先於內部簡單

Ruby 官方介紹將其描述為多種語言思想的平衡，並引用 Matsumoto 對「自然而非簡單」的追求，以及「外表簡單、內部複雜」的設計態度。[R5]

這是一個明確的複雜度配置案例：

> C_(surface)downarrow
> ⇒
> C_(runtime)+C_(semantics)uparrow

初步風格推論：

高使用者自然性
高表達彈性
接受多種寫法
接受內部複雜
偏好程式設計者愉悅與表達

SECTION

四十六、C++：相容性、一般性與機器模型

Stroustrup 在 C++0x 設計文章中同時強調：

- 語言演化受既有程式與社群相容性約束；
- 應偏好一般化機制，而非無限加入專用特徵；
- 零額外成本與直接硬體映射是基礎；
- 語言演化本身也是設計，而不是設計的反面。[R6]

  因此 C++ 不能只被分類成「性能派」：

  高機器透明度
  高一般機制偏好
  高向後相容
  高社群多樣性壓力
  高演化約束

  今日 C++ 的風格也不能完全歸於 Stroustrup個人，因為標準委員會與大型既有生態已成為制度性設計者。

SECTION

四十七、Rust：從創始風格到制度風格

Rust 的 RFC 流程把重大語言變更放入公開設計、討論與共識程序，目標是為成熟平台建立一致且受控制的演化路徑。[R7]

因此研究 Rust 時至少要分開：

早期創始設計
Pre-1.0 社群重構
RFC 制度
Edition 演化

若只給 Rust 一個「Graydon Hoare 風格分數」，會錯誤忽略制度演化。

SECTION

四十八、避免抽象讚美

「簡潔」「安全」「自然」「實用」本身都不是無條件優點。

每個評價必須寫成：

在什麼場景
對什麼使用者
降低什麼成本
增加什麼成本
在什麼條件下失效

SECTION

四十九、情境化優勢

例如「高安全約束」可能適合：

- 長期維護；
- 高風險系統；
- 大型團隊；
- 邊界清楚的領域。

但可能不適合：

- 快速探索；
- 高度動態資料；
- 原型；
- 尚未穩定的問題模型。

SECTION

五十、優點與代價成對輸出

PLDST 的標準格式：

風格選擇　主要收益　主要代價

小型核心　規則少、可推理　程式庫／使用者負擔

強推導　表面簡潔　錯誤解釋與編譯器複雜

高相容性　生態穩定　歷史複雜度累積

多種寫法　表達自由　閱讀與工具一致性下降

強靜態約束　提前排錯　表達與學習摩擦

Runtime 自動化　使用便利　隱藏成本與延遲

元編程　領域表達力　工具與理解困難

SECTION

五十一、創始人偏誤

著名語言容易被敘述為單一天才作品，但真實設計可能涉及：

- 共同設計者；
- 實作者；
- 標準委員會；
- 公司需求；
- 使用者反饋；
- 學術傳統；
- 前代語言。

PLDST 必須列出共同貢獻者與制度。

SECTION

五十二、成功回寫偏誤

語言成功後，社群可能把後來形成的優點回寫成最初明確目標。

分析時要分開：

original intention
early implementation
later rationalization
ecosystem outcome

SECTION

五十三、作品等於作者偏誤

一種語言的缺陷可能不是設計者偏好，而是：

- 當時硬體限制；
- 發布期限；
- 相容壓力；
- 公司政策；
- 編譯器能力；
- 標準妥協。

不能把所有結果人格化。

SECTION

五十四、名言偏誤

設計者的名言適合建立假說，不適合單獨完成分類。

最低要求：

> 名言
> +
> 至少兩項決策證據
> +
> 至少一項反例檢查

SECTION

五十五、標準個案報告

每位設計者的報告包含：

- 歷史情境；
- 目標使用者；
- 主要問題定位；
- 核心設計原則；
- 關鍵決策；
- 被拒絕的替代方案；
- 複雜度配置；
- 責任配置；
- 相容性與演化；
- 治理方式；
- 優勢場景；
- 失效場景；
- 反例與矛盾；
- 時期差異；

- 信心標記；
- 來源分級。

SECTION

五十六、風格摘要卡

設計者：
主要語言：
分析時期：
主要問題：
核心原則：
風格原型：
複雜度移入：
複雜度移出：
使用者責任：
系統責任：
相容性態度：
治理方式：
主要優勢：
主要代價：
歸因信心：

SECTION

五十七、數值不應假裝客觀

可以使用 1–5 或低／中／高協助比較，但每個分數必須附：

- 證據；
- 時期；
- 解釋；
- 信心；
- 反例。

分數是索引，不是真理。

SECTION

五十八、輸入

SKILL 可接受：

- 設計者姓名；
- 語言名稱；
- 設計文件；
- 訪談；
- 提案；
- 原始碼與語法；
- 使用者自有語言設計。

SECTION

五十九、處理管線

Entity Resolution
→ Web Research
→ Primary Source Collection
→ Decision Extraction
→ Attribution Classification
→ Context Segmentation
→ Style Mapping
→ Trade-off Analysis
→ Contradiction Check
→ Fact-check
→ Report

SECTION

六十、來源要求

每次分析都必須重新搜尋：

- 官方語言網站；
- 設計者原始文章；
- HOPL 或同級歷史論文；
- 正式提案系統；
- 可靠學術研究；
- 最新治理與語言狀態。

不得只沿用上一位設計師的搜尋結果。

SECTION

六十一、輸出物

A. 史實層

只陳述可確認資料。

B. 原話層

列出設計者明確主張，避免斷章取義。

C. 決策層

列出實際功能、拒絕與取捨。

D. 推論層

給出 PLDST 風格判定。

E. 反證層

列出不符合初步風格的決策。

F. 應用層

分析該風格適合與不適合的情境。

SECTION

六十二、SKILL 不應做的事

不得：

- 用網路流行印象取代原始資料；
- 將設計師標記為「好／壞」；
- 只依一種語言判斷全部思想；
- 把社群演化全歸於創始人；
- 把心理人格推論成事實；
- 將分數當作絕對排名；
- 忽略時代與硬體限制；
- 把優點與代價分開書寫。

SECTION

六十三、書籍定位

暫定書名：

它不是語言教科書，也不是人物傳記，而是：

> 以設計決策為單位，研究程式語言設計者如何分配複雜度、自由、安全、效能、可讀性、相容性與治理權。

SECTION

六十四、章節模板

每位設計者章節可使用：

- 他看見了什麼問題；
- 他不滿意當時的什麼；
- 他最相信什麼；
- 他把複雜度移到哪裡；
- 他把責任交給誰；
- 他拒絕了什麼；
- 他的設計在哪裡成功；
- 他的風格在哪裡產生代價；
- 後來的社群如何改變其作品；
- AI 能否模擬這種風格。

SECTION

六十五、公開資料限制

有些設計決策：

- 沒有公開記錄；
- 只存在私人郵件；
- 由多人共同形成；
- 被後來敘事重構。

因此部分判定只能是中低信心。

SECTION

六十六、語言與人難以完全分離

設計者會被：

- 技術條件；
- 組織；
- 經濟；
- 社群；
- 既有語言；
- 實作者；

限制。PLDST 不能把設計風格視為全因。

SECTION

六十七、評估者偏見

分析者本身也有語言偏好。

因此應：

- 公開維度；
- 保存來源；
- 列出反例；
- 允許多位分析者比較；
- 將事實與推論分層。

SECTION

六十八、分數簡化

高維設計思想不可能被完全壓縮成雷達圖。圖表與分數只適合導覽，完整判斷仍需閱讀決策證據。

程式語言設計史經常被描述為：

- 一連串新語法；
- 一連串新範式；
- 一連串更安全或更方便的功能；
- 一連串著名設計者的傳記。

但更深一層看，程式語言設計史也是：

> 不同設計者對複雜度應由誰承擔、錯誤應在哪裡阻止、使用者應有多少自由、機器成本應多透明、舊程式應被保護到什麼程度，以及語言應由誰治理的長期爭論。

因此，PLDST 的研究對象不是單一功能，而是反覆出現的決策規則。

本文提出：

> 【Σ_(d,t)
> =
> (
> Context,
> Problem,
> Values,
> Complexity,
> Responsibility,
> Evolution
>)】

並主張：

- 設計者風格不能由範式直接推出；
- 風格必須從多筆決策語料中歸納；
- 設計者原話、實際決策與生態結果必須分離；
- 個人風格與制度風格必須分離；
- 風格具有時間相位；

- 每個優點都必須同時描述代價與適用條件；
- 任何數值分類都必須附上證據與信心；
- 「設計決策人格」是一個研究工具，不是心理診斷。

PLDST 最終要建立的，不是一張誰優誰劣的排行榜，而是一套能夠回答下列問題的分析語言：

> 當一位設計者面對程式語言不可避免的衝突時，他通常保護什麼、犧牲什麼、把成本交給誰，又如何讓這些選擇在多年之後形成一種可辨識的設計風格？

SECTION

A.1 決策記錄

```
q
=
(
problem,
context,
alternatives,
choice,
rationale,
allocation,
outcome
)
```

SECTION

A.2 風格簽名

```
Σ_(d,t)
=
Aggregate
(
{
```

SECTION

A.3 複雑度配置

> B_C
> =
> (
> C_(surface),
> C_(semantics),
> C_(compiler),
> C_(runtime),
> C_(library),
> C_(tooling),
> C_(user),
> C_(governance)
>)

SECTION

A.4 責任配置

> B_R
> =
> (
> R_(programmer),
> R_(compiler),
> R_(runtime),
> R_(library),
> R_(tool),
> R_(organization)
>)

[R1] Niklaus Wirth, “Modula-2 and Oberon,” manuscript revised 2006; published in the HOPL-III proceedings, 2007.

— Wirth 對 Modula-2 與 Oberon 的歷史回顧，明確說明設計簡潔、概念清晰、特徵經濟、實作效率與可靠性的關係。

[R2] Guido van Rossum, “Foreword for Programming Python, First Edition,” 1996, Python.org.

— Python 起源、ABC 的影響、Unix／C 使用者、開放擴充、常見情境效率、縮排與可讀性。

[R3] Tim Peters, “PEP 20 – The Zen of Python,” Python Enhancement Proposals.

— Informational PEP，濃縮 Python 的部分設計原則；作者為 Tim Peters。

[R4] Rob Pike, “Go at Google: Language Design in the Service of Software Engineering,” 2012, Go project.

— Go 的大型軟體工程目標、組織情境、垃圾回收與複雜度配置。

[R5] Ruby official website, “About Ruby: The Ideals of Ruby’s Creator.”

— Ruby 的混合來源、自然性、外表與內部複雜度、表達彈性。

[R6] Bjarne Stroustrup, “The Design of C++0x,” C/C++ Users Journal, 2005.

— 相容性、一般機制、語言演化、零額外成本與硬體模型。

[R7] Rust RFC 0002, “RFC Process,” 2014.

— Rust 重大變更的公開設計、共識與成熟平台治理流程。

[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, pp. 131–174, DOI: 10.1006/jvlc.1996.0009.

[R9] ACM SIGPLAN, HOPL-IV Papers and Content Guidelines, 2021.

— 語言歷史、演化、設計主題、技術準確性與歷史完整性的研究要求。

[R10] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine, “Interdisciplinary Programming Language Design,” PL’18, 2018.

— 目標使用者、應用情境、品質屬性、形式方法、人本方法與跨學科語言設計。

[R11] Michael Kölling, “Principles of Educational Programming Language Design,” Informatics in Education 23(4), 2024, pp. 823–836, DOI: 10.15388/infedu.2024.29.

— 教育語言中的簡潔、模組性、正交性、可讀性及其現代解釋。

後續個案論文統一使用：

[F] Fact：可確認史實
[Q] Quote：設計者或正式文件原意
[D] Decision：可辨識設計決策
[I] Interpretation：本文分析推論
[C] Counterevidence：反例或矛盾
[U] Uncertain：證據不足

本篇完成初稿後，已再次以官方或原始資料核對下列項目：

- Wirth 的來源與年代

  《Modula-2 and Oberon》手稿於 2006 年修訂，後收入 2007 年 HOPL-III 論文集；「設計簡潔是最重要指導原則」來自其摘要，不是本文自行替他創造的口號。

- Python 與 PEP 20 的歸因

  Python 的起源、ABC 關係、Unix／C 目標讀者、縮排與可讀性主要依 Guido van Rossum 1996 年文章；PEP 20 的作者是 Tim Peters，文件類型是 Informational，因此本文沒有把它寫成 Guido 親筆或規範性語言標準。

- Go 的共同設計與文件作者

  Go at Google 是 Rob Pike 於 2012 年發表的官方設計回顧；Go 的核心設計者包括 Robert Griesemer、Rob Pike 與 Ken Thompson。本文將該文作為 Go 設計共同體的原始材料，而不是把 Go 全部歸因於 Pike 個人。

- Ruby 的自然性與內部複雜度

  「natural, not simple」以及外表簡單、內部複雜的說法，可在 Ruby 官方介紹頁找到；本文將其用於複雜度配置分析，而沒有推導為 Matsumoto 的一般心理人格。

- C++ 的相容性與零額外成本

  相容性、一般機制、語言演化與 zero-overhead 的表述，均可在 Stroustrup 2005 年 C++0x 設計文章中找到；本文同時標示現代 C++ 已具有委員會與社群制度風格，避免全部歸因於個人。

- Rust 的制度歸因

  Rust RFC 0002 的起始日期為 2014 年 3 月 11 日，並將重大變更置於公開設計與共識流程。本文只用它說明制度風格，不將現代 Rust 的全部設計歸於單一創始者。

- Cognitive Dimensions 的用途

  Green 與 Petre 將 Cognitive Dimensions 定位為討論與權衡工具，而非一組保證最佳設計的規則；本文沿用此限制。

- 複雜度預算的理論地位

  本文的 B_C 是分析用啟發式模型，不是嚴格的複雜度守恆定律。成稿已明確寫出，良好統一設計有可能真正降低總複雜度。

- 史實、原話與推論分層

  代表性人物段落中的「初步風格推論」均屬本文分析，不等同設計者自我描述。後續個案論文將使用 [F]／[Q]／[D]／[I]／[C]／[U] 標記進一步細分。