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

Designers, Communities, and Institutions: Avoiding Founder-Attribution Bias in Programming Language History

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

https://thisoneisneok.com/html/papers/pldst-004.html

英文名稱： Designers, Communities, and Institutions: Avoiding Founder-Attribution Bias in Programming Language History

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-004

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

程式語言史經常以單一創始者為敘事中心：

- Python 是 Guido van Rossum 的語言；
- C++ 是 Bjarne Stroustrup 的語言；
- Ruby 是 Yukihiro Matsumoto 的語言；

- Rust 是 Graydon Hoare 的語言；
- Go 是 Rob Pike 的語言；
- JavaScript 是 Brendan Eich 的語言。

這些說法作為入門識別並非全錯，但一旦被用來解釋語言的全部特徵、長期演化、制度選擇與生態結果，就會形成 創始人歸因偏誤。它把多階段、多主體、多制度的語言演化壓縮為單一人物的穩定意志，進而混淆：

- 誰最早提出問題；
- 誰選擇核心設計；
- 誰共同設計機制；
- 誰完成編譯器與運行時；
- 誰批准規格；
- 誰維護相容性；
- 誰透過程式庫、工具與使用習慣實際改變語言；
- 誰在事故、標準化或公司策略中承擔最終責任。

Go 的官方歷史明確指出，Robert Griesemer、Rob Pike 與 Ken Thompson 於 2007 年共同開始設計；Rust 的重大技術變更進入 RFC 與團隊治理；Python 在 Guido van Rossum 於 2018 年卸下 BDFL 身分後，透過社群投票建立 Steering Council；C++ 自 1990–1991 年起由 ISO WG21 進行標準化與演化；Java 透過 JCP、JSR、Reference Implementation 與 Technology Compatibility Kit 建立制度化規格流程；ECMAScript 則由 TC39 的提案階段、Champion、實作回饋與共識機制持續演進。[R1][R2][R3][R4][R5][R6]

本文提出 多主體語言歸因模型（Multi-Actor Language Attribution Model, MALAM），將語言演化表示為：

L⊠
=
cal-E
(
F,
D,
I,
G,
O,

其中：

- F：Founder／Initiator，創始與啟動者；
- D：Co-designers，共同設計者；
- I：Implementers，實作者；
- G：Governance bodies，治理機構；
- O：Organizations，企業、研究機構與贊助者；
- S：Standards and specifications，標準與規格制度；
- U：Users and ecosystem，使用者與生態；
- X：External constraints，硬體、平台、相容性與時代限制。

本文進一步區分四種不可混同的歸因：

> 【功勞歸因
> ≠
> 因果歸因
> ≠
> 權力歸因
> ≠
> 責任歸因】

一位創始者可能具有高度歷史功勞，卻不再擁有當前決策權；一個委員會可能具有正式批准權，卻不是最初概念的發明者；一家公司可能提供絕大部分實作資源，卻不應自動取得對整個社群價值的唯一解釋權；生態共同體可能沒有正式投票權，卻能透過採用、拒絕、慣例與工具形成事實上的設計選擇。

本文建立：

- 語言生命週期的七個歸因階段；
- 十類設計參與者；
- 功勞、因果、權力、責任與維護的五維歸因矩陣；
- 個人風格、共同設計風格、組織風格、制度風格與生態風格的分層方法；

- Python、Go、Rust、C++、Java／JCP 與 ECMAScript／TC39 的制度轉換案例；
- 適用於 PLDST 個案研究、比較研究與 AI SKILL 的來源標記與歸因規則。

本文的核心命題是：

> 【語言的名字可以指向一位創始者，
>
> 但語言的歷史不能只歸因於一個人。】

關鍵詞： 程式語言史、創始人偏誤、共同設計、語言治理、標準委員會、RFC、PEP、WG21、TC39、JCP、PLDST

SECTION

一、什麼是創始人歸因偏誤

本文所稱的創始人歸因偏誤，是指：

> 因為某位人物最早建立、命名、公開或代表一種語言，研究者便將該語言後續的全部設計決策、制度變遷、技術成果與生態特徵，過度歸因於該人物的個人風格。

它常出現在下列句型：

X 設計了這個語言，所以今日所有特徵都反映 X 的思想。
X 喜歡簡潔，所以社群後來拒絕某功能也是 X 的風格。
這個語言存在某缺陷，因此是 X 的人格造成。
X 離開後語言仍如此演化，證明一切本來都由 X 決定。

這些敘述可能包含部分真實因素，但缺少：

- 時間；
- 決策主體；
- 組織條件；
- 實作限制；
- 正式治理；

- 生態選擇；
- 反例。

SECTION

二、創始者為何仍然重要

反對過度歸因，不等於否認創始者。

創始者常常決定：

- 最初問題如何被 framing；
- 第一組核心概念；
- 哪些前代語言被繼承；
- 第一個實作；
- 命名與公共敘事；
- 早期審美；
- 第一批參與者；
- 誰有權接受與拒絕變更。

因此，創始者可能對語言早期相位具有高權重：

> $w\_F(t\square) \gg w\_F(t\square)$

但其權重是否隨時間下降，取決於：

- 是否仍具有否決權；
- 是否仍主導實作；
- 是否建立制度；
- 語言是否標準化；

- 生態是否擴張；
- 公司是否接管資源；
- 社群是否重構核心。

SECTION

三、「誰創造了語言」其實包含多個問題

3.1 誰提出了原始問題？

例如：

- 現有語言過於複雜；
- 大型建置速度太慢；
- 記憶體安全與效能難兼得；
- Web 需要可嵌入腳本；
- 教學語言需要更清楚的抽象。

3.2 誰選擇核心機制？

例如：

- 所有權；
- 垃圾回收；
- Prototype；
- 顯式錯誤值；
- 縮排語法；
- Traits；

- 模組。

3.3 誰讓它真正可用？

包括：

- 編譯器實作者；
- Runtime 工程師；
- 標準程式庫作者；
- 套件管理者；
- 文件作者；
- 工具與 IDE 開發者。

3.4 誰決定它成為標準？

可能是：

- 創始者；
- Core team；
- Steering Council；
- RFC team；
- ISO 委員會；
- Ecma technical committee；
- JCP Executive Committee；
- 公司產品決策。

3.5 誰決定它實際如何被使用？

可能是：

- 框架；
- 程式庫；
- 教育；
- 雲端平台；
- 公司規範；
- 社群慣例；
- 套件管理；
- Linter 與 Formatter。

因此，「創造者」不是單一資料欄位。

SECTION

四、功勞歸因 Credit Attribution

功勞歸因回答：

> 誰應因某項創新、工作或歷史貢獻獲得承認？

它包括：

- 原始構想；
- 共同設計；
- 第一個實作；
- 關鍵改寫；
- 社群組織；
- 文件；
- 標準化；

- 長期維護。

  功勞可以共享，且不一定與正式權力相同。

SECTION

五、因果歸因 Causal Attribution

因果歸因回答：

> 哪些主體與條件實際造成某項結果？

例如某個語言功能出現，可能同時由：

- 設計者理念；
- 硬體限制；
- 公司需求；
- 使用者壓力；
- 前代語言；
- 編譯器技術；
- 相容性；

  共同造成。

因果模型應寫成：

```
Outcome
=
f(
Actors,
Constraints,
Alternatives,
Institutions,
Timing
)
```

而不是：

> Outcome
> =
> Personality(Founder)

SECTION

六、權力歸因 Authority Attribution

權力歸因回答：

> 當時誰能批准、否決、延後或撤回設計？

權力可能來自：

- 專案所有權；
- Commit 權限；
- BDFL；
- 團隊 Charter；
- 委員會投票；
- 公司資金；
- 商標；
- 發行控制；
- 實作控制；
- 標準批准。

具有觀念影響力，不等於具有正式決定權。

SECTION

七、責任歸因 Accountability Attribution

責任歸因回答：

> 若設計造成安全、相容、治理或生態問題，誰應處理？

可能包括：

- 原提案作者；
- 批准機構；
- 實作團隊；
- 發行管理者；
- 公司；
- 標準機構；
- 使用者組織。

不能只把責任放回早已離開專案的創始者。

SECTION

八、維護歸因 Maintenance Attribution

維護歸因回答：

> 誰持續支付語言存在與演化的成本？

包括：

- 修復；
- Regression test；

- 移植；
- 發行；
- 安全回報；
- 標準更新；
- 套件基礎設施；
- 文件；
- 相容性；
- 社群治理。

  維護者可能沒有創始光環，卻對今日語言樣貌具有更直接影響。

SECTION

九、五維歸因向量

對某項決策 q 與主體 a，定義：

> A(a,q)
> =
> (
> Cr,
> Ca,
> Au,
> Ac,
> Ma
>)

其中：

- Cr：Credit；
- Ca：Causal contribution；

- Au：Authority；
- Ac：Accountability；
- Ma：Maintenance。

同一主體在五維上的權重可以完全不同。

SECTION

十、創始者 Founder／Initiator

典型貢獻：

- 提出初始問題；
- 建立核心願景；
- 第一版設計；
- 第一個實作或原型；
- 命名；
- 招募共同設計者。

風險：

- 後來的公共敘事將所有共同工作吸收到創始者名下。

SECTION

十一、共同設計者 Co-designer

共同設計者可能：

- 與創始者同步提出概念；
- 負責不同子系統；
- 反駁並修正原始設計；

- 建立語法、型別、Runtime 或程式庫。

  Go 是明確案例：官方 FAQ 與歷史資料將 Robert Griesemer、Rob Pike、Ken Thompson 同列為最初設計者。[R1]

  若只稱 Go 為「Rob Pike 設計」，會抹去共同創始結構。

SECTION

十二、實作者 Implementer

實作者不是單純把規格翻譯成程式碼。

實作會反向決定：

- 哪些功能可行；
- 錯誤訊息；
- 效能；
- 未定義邊界；
- 與平台的互操作；
- 是否能穩定化；
- 功能是否被撤回。

  第一個可用編譯器、第二個獨立實作與優化實作，對語言形成的因果作用可能不同。

SECTION

十三、程式庫與工具作者

標準程式庫、Formatter、Linter、Package manager、Build system 與 Language server 會決定：

- 什麼被視為慣用寫法；
- 哪些選項被隱藏；

- 如何組織專案；
- 如何處理錯誤；
- 如何發布與更新。

  一種語言的有效風格為：

> L_(effective)
> =
> Core
> +
> Library
> +
> Tooling
> +
> Conventions

SECTION

十四、技術領導與 Core Team

此類主體可能具有：

- Triage；
- Roadmap；
- RFC 指派；
- Release decision；
- 部門協調；
- 最終技術裁決。

  它介於個人與制度之間，常形成共同體風格。

SECTION

十五、正式治理機構

例如：

- Python Steering Council；
- Rust Leadership Council／Teams；
- ISO WG21；
- TC39；
- JCP Executive Committee。

治理機構定義：

- 誰可以提出；
- 如何審查；
- 何謂共識；
- 何時成熟；
- 誰能否決；
- 如何處理衝突。

SECTION

十六、公司與研究機構

公司或機構可能提供：

- 員工時間；
- 基礎設施；
- 商標；

- 發行；
- 產品採用；
- 法務；
- 安全；
- 標準代表。

但組織資助不等於所有參與者皆代表組織意志，也不代表社群對語言沒有實質塑造。

SECTION

十七、標準機構與國家代表

ISO、Ecma 等標準制度涉及：

- 國家會員；
- 委員會；
- 正式文件；
- 會議程序；
- 投票；
- 編輯；
- 發布週期。

C++ 的現代語言設計不能只被描述為 Stroustrup 個人的延伸，因為 WG21 自 1990–1991 年成立後，已有正式、多國、分組的標準化結構。[R4]

SECTION

十八、使用者與生態共同體

使用者透過：

- 採用；
- 拒用；
- Workaround；
- Framework；
- Style guide；
- Experience report；
- Issue；
- Fork；
- 套件；
- 教育；

  對語言形成選擇壓力。

  使用者不必擁有正式投票權，也能造成生態因果。

SECTION

十九、外部約束

外部約束不是人，但具有因果作用：

- 硬體；
- 作業系統；
- 瀏覽器；
- ABI；
- 網路；
- 既有程式；

- 法規；
- 安全事件；
- 市場時機；
- 相容性。

PLDST 不應把被環境迫使的妥協解釋成純個人偏好。

SECTION

二十、相位一：問題形成

主要資料：

- 設計者回憶；
- 原始備忘錄；
- 前代語言批判；
- 組織需求；
- 第一批實驗。

主要主體：

- 創始者；
- 小型共同設計群；
- 贊助機構。

SECTION

二十一、相位二：核心定型

主要工作：

- 語法；

- 語義；
- 編譯器；
- Runtime；
- 標準程式庫；
- 第一批使用案例。

  此時個人風格權重通常最高，但實作者的反饋已開始改變設計。

SECTION

二十二、相位三：公開採用

語言開始接收：

- 外部使用者；
- Bug report；
- 套件；
- 平台移植；
- 教學；
- 公司採用。

  部分設計原則可能被實務反駁或重新解釋。

SECTION

二十三、相位四：治理制度化

出現：

- PEP；
- RFC；

- Proposal process；
- Core team；
- Council；
- 標準委員會；
- Release policy。

  語言從「作品」轉向「制度」。

SECTION

二十四、相位五：相容性鎖定

當大量程式依賴既有行為後：

> Cost_(change)
> =
> Migration
> +
> Ecosystem
> +
> Education
> +
> Implementation
> +
> Trust

設計自由被歷史使用量限制。

SECTION

二十五、相位六：多實作與標準化

若存在：

- 多個編譯器；
- 多個 Runtime；
- 多個瀏覽器；
- 國際標準；
- Compatibility test；

規格與一致性制度的權重上升。

SECTION

二十六、相位七：後創始者時代

創始者可能：

- 離開；
- 退休；
- 失去正式權力；
- 成為象徵人物；
- 繼續影響但不再裁決。

此時語言仍可能保留其早期風格，但不能把新決策自動歸因於創始者。

SECTION

二十七、語言狀態

令語言在時間 t 的狀態為：

$$
\begin{aligned}
L_t &= (\\
&\quad \mathrm{Spec}_t,\\
&\quad \mathrm{Impl}_t,
\end{aligned}
$$

每次變更：

> L_(t+1)
> =
> L⊠+Δ q⊠

其中 Δ q⊠ 由多個主體與約束共同作用。

SECTION

二十八、參與圖

建立有向圖：

> G_q=(A,E)

節點包括參與者，邊表示：

- 提出；
- 共同設計；
- 實作；
- 審查；
- 批准；
- 否決；
- 測試；
- 採用；
- 維護；
- 撤回。

  一項提案可以具有：

  author

champion
reviewer
implementer
approver
editor
test author
maintainer
affected community

SECTION

二十九、權重不是固定人格分數

對主體 a、決策 q 與時期 t：

> w(a,q,t)
> =
> f(
> documented contribution,
> decision authority,
> implementation control,
> maintenance,
> counterfactual dependence
>)

所謂 counterfactual dependence 問：

> 若沒有這個主體，該決策是否仍會以相同形式發生？

這仍是推論，不是可完全計算的物理量。

SECTION

三十、歸因信心

每項歸因標記：

直接文件
多人一致回顧
正式決議

版本控制證據
實作證據
次級歷史
本文推論
爭議
信心分為：

- 高；
- 中；
- 低；
- 不可判定。

SECTION

三十一、Python：從 BDFL 到 Steering Council

PEP 13 記錄：

- Python 由 Guido van Rossum 啟動；
- Guido 自專案創始至 2018 年 7 月擔任 BDFL；
- 目前治理以五人 Steering Council 為核心；
- Council 具有廣泛權力，但尋求盡量少直接使用，而建立標準程序與委任機制。[R2]

2018 年後的治理轉換也經歷：

- 多個 PEP 801x 模型；
- PEP 8001 投票；
- 最終將制度寫入 PEP 13。[R2][R8]

因此應分期：

Python 早期

創始者權重：高
共同體權重：逐步上升

最終裁決：BDFL

Python 現代

創始功勞：Guido
正式權力：Steering Council 與委任流程
設計材料：PEP 作者、Delegate、Core developers
實作與維護：多團隊共同體
現代 Python 某項 PEP 被接受，不應自動寫成「Guido 的設計風格」。

SECTION

三十二、Go：共同創始與受管理提案

Go 官方 FAQ 與歷史頁明確指出：

- Robert Griesemer、Rob Pike、Ken Thompson 於 2007 年 9 月開始討論與設計；
- 語言源自他們及 Google 同事面對的大型軟體工程問題。[R1]

  後續 Go 的語言、程式庫與工具重大變更，需進入 Change Proposal Process；官方 Contribution Guide 明確要求重大語言、API、工具與命令列改變先通過提案程序。[R9]

  因此 Go 至少有三層歸因：

- 三位共同創始者的早期風格；
- Go Team 與實作者的工程風格；
- Proposal review、相容性政策與使用者經驗報告形成的制度風格。

  將 Go 全部歸因於 Rob Pike，既不符合官方創始歷史，也會忽略後續制度。

SECTION

三十三、Rust：創始者、重構共同體與 RFC 制度

Rust 的早期創始與 Graydon Hoare 密切相關，但現代 Rust 的設計已不能只以創始者解釋。

RFC 0002 定義重大功能進入語言與標準程式庫的受控路徑；RFC 1068 進一步描述團隊與 RFC 治理；當代專案又建立 Leadership Council 作為整體治理結構。[R3][R10]

Rust 個案至少應分為：

個人原型期
Mozilla／共同設計期
Pre-1.0 大規模重構
RFC 制度期
Edition 與穩定演化期
現代 Leadership Council／Team 治理
Ownership、Borrow checker、Traits、Cargo、Edition 並非都能以單一人物解釋。

SECTION

三十四、C++：創始設計與 ISO WG21

Stroustrup 對 C++ 的早期架構、抽象理念與硬體模型具有不可替代的創始貢獻。

但 WG21：

- 成立於 1990–1991 年；
- 由 ISO／IEC JTC1／SC22 的國家會員專家參與；
- 依正式 ISO 要求與 WG21 自身程序運作；
- 使用提案文件、分組、會議與投票推進標準。[R4][R11]

  因此現代 C++ 功能具有多重主體：

- 提案作者；
- Evolution／Library Working Group；
- Core／Library wording；
- 實作者；
- 國家代表；
- 委員會共識；
- 使用者與既有程式相容壓力。

「C++ 喜歡增加功能，因為 Stroustrup 個人喜歡複雜」是低品質歸因。它忽略：

- 委員會多方需求；
- 已有生態；
- 相容性；
- 不同應用領域；
- 提案流程；
- 多編譯器實作。

SECTION

三十五、Java：創始語言、平台公司與 JCP

Java 的早期設計與 James Gosling、Sun 團隊密切相關，但 Java 技術規格後來進入 Java Community Process。

JCP 官方資料將其描述為：

- 國際 Java 社群標準化與批准 Java 技術規格的流程；
- 結合社群審查、專家輸入與技術領導；
- 通過的規格配有 Reference Implementation；
- 另有 Technology Compatibility Kit 測試實作是否符合規格。[R5]

因此 Java 的歸因需要區分：

語言創始
平台與 JVM 實作
公司產品策略
JSR Expert Group
JCP Executive Committee
Reference Implementation
TCK 與相容性制度
廣大企業生態

JCP 也並非完全等同開放社群自治；公司、規格領導與正式會員結構仍具有不同權重。PLDST 應描述制度本身，而不是用「社群治理」四字抹平差異。

SECTION

三十六、ECMAScript／TC39：實作者共識與分階段成熟

JavaScript 最初由 Brendan Eich 建立，但今日 ECMAScript 語言規格由 TC39 持續維護與演進。

TC39 官方將自身描述為由 JavaScript 開發者、實作者、學者等共同維護語言定義的團體；提案經多階段成熟，Stage 3 表示設計解決方案已完成主要問題處理，Stage 4 完成後進入下一年度規格快照。[R6]

TC39 的關鍵角色包括：

- Proposal author；
- Champion；
- Committee participants；
- Browser／engine implementers；
- Spec editors；
- Test262；
- 生態回饋；
- 共識程序。

因此今日某項 JavaScript 功能不應寫成「Brendan Eich 的設計」。更精確的說法是：

> 它是 ECMAScript 制度下，由特定 Champion 推動、經 TC39 共識、實作與測試成熟的功能。

SECTION

三十七、個人風格

個人風格來自：

- 一位設計者在多項直接決策中的重複取捨；
- 本人原始文章；

- 可歸因的設計文件；
- 明確拒絕；
- 親自實作；
- 具有實質裁決權的時期。

SECTION

三十八、共同設計風格

共同設計風格不是各成員風格的簡單平均。

它可能來自：

- 互相制衡；
- 角色分工；
- 共同問題；
- 經常性折衷；
- 固定討論程序；
- 共同實作經驗。

Go 的早期風格應先研究三位創始者如何互動，而不是只將三個個人分數相加。

SECTION

三十九、組織風格

組織風格可能包括：

- 工程生產力；
- 發行節奏；
- 商業相容；

- 平台控制；
- 安全政策；
- 人員與基礎設施。

公司需求可以深刻影響語言，但不應自動被寫成每位設計者的個人信念。

SECTION

四十、制度風格

制度風格由規則反覆產生：

- PEP；
- RFC；
- Stage；
- WG；
- JSR；
- 選舉；
- 共識；
- Compatibility test；
- Edition；
- Release train。

它的特徵包括：

- 決策速度；
- 透明度；
- 參與門檻；

- 代表性；
- 相容性偏好；
- 實驗偏好；
- 否決方式；
- 責任分配。

SECTION

四十一、生態風格

生態風格可能表現在：

- 慣用程式碼；
- 套件結構；
- Framework；
- 格式；
- 相容慣例；
- 教育；
- 社群價值。

它不一定由規格強制，卻能比語法更直接影響一般使用者。

SECTION

四十二、五層風格表示

$$
\begin{aligned}
&\mathrm{Style}(L,t)\\
&=\\
&(\\
&S_{(\mathrm{personal})},\\
&S_{(\mathrm{collective})},
\end{aligned}
$$

任何個案研究都應標示分析的是哪一層。

SECTION

四十三、單一作者神話

將所有共同設計者、實作者與社群工作吸收為創始人的作品。

修正：

列出共同設計者
列出第一實作者
列出關鍵後續團隊
列出正式治理

SECTION

四十四、後期特徵回寫

將後來制度接受的功能寫成創始者原始願景。

修正：

記錄功能首次提出時間
當時決定權
提案作者
批准機構
創始者是否參與

SECTION

四十五、創始者人格化缺陷

語言的每個歷史包袱都被解釋為創始者性格缺陷。

修正：

- 查核相容性；
- 查核硬體；
- 查核公司；
- 查核委員會；

- 查核實作；
- 查核被否決替代方案。

SECTION

四十六、制度去人格化

另一個極端是把所有決策歸於「社群」，使真正有權力的人消失。

「社群決定」可能實際表示：

- 五人 Council 決定；
- Champion 說服委員會；
- 公司實作先行；
- 少量 Core member 批准；
- 國家代表投票；
- 生態以採用形成壓力。

修正：

> 不只寫制度名稱，也寫誰在制度中具有何種權力。

SECTION

四十七、提交者等於設計者

Version control 顯示誰提交程式碼，不一定顯示誰：

- 提出概念；
- 寫規格；
- 做審查；
- 提供測試；

- 批准；
- 維護。

Commit attribution 只是實作證據的一部分。

SECTION

四十八、公司等於所有貢獻者

公司僱用多數核心成員，不代表：

- 每項決策是公司命令；
- 外部成員沒有影響；
- 員工沒有專業自主；
- 生態沒有否決能力。

同樣地，將公司影響完全忽略也不準確。

SECTION

四十九、今日權力回寫過去

當代治理機構不能被回寫成早期創始階段的決策者；早期 BDFL 也不能被回寫成今日每項決策的權力來源。

SECTION

五十、步驟一：固定決策與時期

不要分析：

誰設計了 Rust？
應分析：

Rust 1.0 前的 ownership model 由哪些主體形成？
Rust RFC 制度如何改變語言特徵的批准方式？
Rust 2024 Edition 的正式責任鏈是什麼？

SECTION

五十一、步驟二：建立角色表

Initiator
Co-designer
Spec author
Proposal author
Champion
Reviewer
Implementer
Approver
Maintainer
Affected community

SECTION

五十二、步驟三：建立時間線

至少包含：

- 問題提出；
- 第一版；
- 主要重構；
- 公開發行；
- 治理制度；
- 標準化；
- 創始者退出；
- 當前狀態。

SECTION

五十三、步驟四：建立權力圖

回答：

- 誰可以正式接受？
- 誰可以阻止？
- 誰控制主要實作？
- 誰控制 Release？
- 誰控制商標與基礎設施？
- 誰承擔相容性？
- 誰能使提案因不實作而失效？

SECTION

五十四、步驟五：建立證據層級

A 級

- 正式規格；
- PEP／RFC／Proposal；
- 會議決議；
- 版本控制；
- 官方治理文件。

B 級

- 設計者原始文章；
- HOPL；
- 官方訪談；

- 發行回顧。

C 級

- 可靠學術史；
- 參與者回憶；
- 技術出版物。

D 級

- 社群印象；
- 二手文章；
- 無來源傳聞。

PLDST 核心歸因不能只依 D 級資料。

SECTION

五十五、步驟六：分離五種歸因

對每一主體分別記錄：

Credit
Causal contribution
Authority
Accountability
Maintenance

SECTION

五十六、步驟七：找反例

若初步判斷某創始者「反對複雜功能」，應查：

- 是否曾支持複雜功能；
- 是否因相容性而拒絕；

- 是否由後期制度接受；
- 是否只是時間與資源不足；
- 是否本人後來改變觀點。

SECTION

五十七、人物個案不能只寫人物

每篇設計師個案需同時包含：

- 人物直接決策；
- 共同設計者；
- 實作團隊；
- 贊助組織；
- 治理制度；
- 後期生態；
- 創始者退出後的變化。

SECTION

五十八、歸因標記

[F] 可確認史實
[Q] 當事人原始陳述
[D-I] 個人直接決策
[D-C] 共同設計決策
[D-O] 組織驅動決策
[D-G] 正式治理決策
[D-E] 生態形成結果
[I] 本文推論
[C] 反例或爭議
[U] 無法可靠歸因

SECTION

五十九、標準歸因卡

語言：
決策：
時間：
創始者：
共同設計者：
提案作者：
實作者：
批准機構：
主要贊助組織：
相容性約束：
生態影響：
功勞歸因：
因果歸因：
權力歸因：
責任歸因：
維護歸因：
反例：
來源：
信心：

SECTION

六十、人物風格結論格式

不得只寫：

X 是極簡派。

應寫：

在 t1 時期，X 對由其直接控制的 q1、q2、q3 決策，反覆偏好某種取捨；此判斷不延伸至 t2 後由 G 制度批准的全部語言功能。信心：中高。

SECTION

六十一、決策入口

- 任何人可提案？
- 需會員？

- 需 Champion？
- 需 Core member？
- 需國家代表？
- 需公司贊助？

SECTION

六十二、成熟階段

- 草案；
- 實驗；
- RFC；
- Stage；
- Candidate；
- Stable；
- 標準發布。

  成熟階段本身反映制度風格。

SECTION

六十三、決策方法

- 個人裁決；
- 多數決；
- 共識；
- 無反對；
- 國家投票；

- Council；
- Team delegation；
- 實作先行。

「共識」在不同制度中也可能具有不同實際含義。

SECTION

六十四、實作要求

- 是否需 Prototype？
- 是否需多個實作？
- 是否需測試？
- 是否需 Reference Implementation？
- 是否需 TCK？
- 是否需生態回饋？
- 是否需遷移工具？

TC39、JCP、Rust、Python 與 WG21 在這些方面具有不同制度指紋。

SECTION

六十五、相容性與撤回

- 何時可破壞相容性？
- 是否有 Edition？
- 是否允許 Feature gate？
- 如何棄用？
- 功能進入後能否撤回？

- 誰支付遷移成本？

SECTION

六十六、Python

早期：個人裁決權高
中期：PEP + BDFL／Delegate
現代：PEP + Steering Council + Delegation
演化偏好：程序化、相容、委任

SECTION

六十七、Go

早期：三人共同設計
現代：Go Team + Proposal review
強約束：Go 1 compatibility
演化偏好：受控、小步、工程證據

SECTION

六十八、Rust

早期：個人原型與公司共同體
現代：RFC + Teams + Leadership Council
特殊工具：Feature gate、Edition、stability
演化偏好：公開設計、實驗後穩定

SECTION

六十九、C++

早期：創始者主導
現代：ISO WG21 多國委員會
決策單位：Papers、Working Groups、Polls
演化偏好：廣泛領域、正式標準、高相容性

SECTION

七十、Java／JCP

早期：Sun 團隊與平台公司
制度：JSR + Expert Group + EC

證據：Specification + RI + TCK
演化偏好：平台一致性、標準批准、相容測試

SECTION

七十一、ECMAScript／TC39

早期：快速個人設計
現代：Champion + Staged Proposal + Consensus
證據：Spec text、implementation、tests
演化偏好：實作者參與、逐階成熟、年度快照

SECTION

七十二、輸入

SKILL 接受：

- 設計師；
- 語言；
- 特定功能；
- 特定時間；
- PEP／RFC／JSR／Proposal；
- Commit；
- 標準文件；
- 官方歷史。

SECTION

七十三、處理管線

重新網路搜尋
→ Entity resolution
→ Timeline segmentation
→ Actor extraction
→ Governance extraction
→ Decision graph

→ Five-attribution scoring
→ Counterevidence search
→ Institution/person separation
→ Fact-check
→ Report

SECTION

七十四、輸出 JSON 雛形

```
{
 "language": "Python",
 "decision": "acceptance of a hypothetical modern PEP",
 "period": "post-2018",
 "actors": [
  {
   "actor": "PEP author",
   "roles": ["proposal", "revision"],
   "credit": "high",
   "authority": "low-to-medium"
  },
  {
   "actor": "Steering Council or delegate",
   "roles": ["approval"],
   "authority": "high"
  },
  {
   "actor": "core developers",
   "roles": ["review", "implementation", "maintenance"]
  }
 ],
 "founder_attribution": {
  "applicable": false,
  "reason": "decision occurred after BDFL governance"
 },
 "confidence": "high"
}
```

SECTION

七十五、SKILL 禁止事項

不得：

- 將語言全部歸於最知名人物；

- 將公司名稱當成單一行動者；
- 將「社群」當成無結構主體；
- 將 Commit author 自動當提案作者；
- 將提案作者自動當批准者；
- 將創始功勞等同當前權力；
- 將制度批准等同原始發明；
- 將生態結果寫成設計者明確意圖；
- 忽略設計時期；
- 不查核共同設計者。

SECTION

七十六、文件不完整

私人討論、公司內部決策與未保存郵件可能使歸因不完整。

因此「沒有文件」不等於「沒有貢獻」。

SECTION

七十七、公開敘事可能重構歷史

創始者、公司與社群都可能：

- 強化自己的角色；
- 簡化共同設計；
- 以後來原則重述早期動機；
- 忽略失敗提案。

需要交叉查核。

SECTION

七十八、制度文件不等於實際權力

正式規則可能說所有人可參與，但實際影響仍取決於：

- 時間；
- 專業；
- 雇主資源；
- 英語；
- 會議參與；
- 實作能力；
- 社群地位。

PLDST 應區分 formal authority 與 effective influence。

SECTION

七十九、數值不能取代敘事

多主體歸因可用矩陣輔助，但不應假裝存在精確的「某人貢獻 37%」。

數值只用於：

- 比較；
- 缺口檢查；
- 信心；
- 可視化。

SECTION

八十、共同體也可能排除

去除創始者神話不代表社群或委員會天然公平。

制度可能存在：

- 參與門檻；
- 公司集中；
- 專業壁壘；
- 地域與語言偏差；
- 非正式權力；
- 決策疲勞。

  制度也需要被分析，而不是被理想化。

SECTION

八十一、Go 的創始歸因

已核對 Go 官方 FAQ、Go 官方案例與歷史材料：

- 2007 年 9 月由 Robert Griesemer、Rob Pike、Ken Thompson 共同開始設計；
- 不應只歸因於 Rob Pike；
- 後續重大語言、API、工具變更需經 Change Proposal Process。

SECTION

八十二、Python 的治理轉換

已核對 PEP 13、PEP 8000 與 PEP 8001：

- Guido van Rossum 至 2018 年 7 月擔任 BDFL；
- 新治理模型由社群程序選出；
- 現行核心為五人 Steering Council；

- Council 擁有廣泛但傾向少直接使用的權力。

  本文沒有將現代 Python 決策回寫為 Guido 個人裁決。

SECTION

八十三、Rust 的 RFC 與治理

已核對 RFC 0002、RFC 1068 與 Leadership Council 官方儲存庫：

- RFC Process 自 2014 年建立重大變更的受控路徑；
- 技術決策與團隊治理已有制度化文件；
- 現代 Rust 不宜僅以 Graydon Hoare 個人風格概括。

SECTION

八十四、C++ WG21

已核對 ISO C++ 官方委員會與 Standing Documents：

- WG21 成立於 1990–1991 年；
- 由 ISO／IEC JTC1／SC22 國家會員專家參與；
- 具有正式提案、分組、會議與投票程序。

  本文仍保留 Stroustrup 的創始功勞，但未將全部現代 C++ 決策歸於個人。

SECTION

八十五、Java JCP

已核對 JCP Procedures Overview：

- JCP 負責 Java 技術規格的標準化與批准；
- 採用社群審查、專家輸入與技術領導；

- 通過規格配有 Reference Implementation 與 TCK。

本文沒有把 JCP 誤寫成完全去中心化社群，也沒有把 Java 全部視為 Oracle 或 Gosling 的單一作品。

SECTION

八十六、TC39

已核對 TC39 Process 與 ECMAScript 官方頁面：

- 現代 ECMAScript 由 TC39 持續維護；
- 新功能採分階段 Proposal；
- Stage 3 與 Stage 4 有不同成熟與實作要求；
- Stage 4 完成功能進入後續年度規格。

本文沒有將現代 JavaScript 功能歸因於 Brendan Eich。

SECTION

八十七、HOPL 方法

已核對 HOPL-IV 的內容指引與研究定位：

- 語言史研究需處理設計、演化、實作、標準化與社會影響；
- 本文以多主體方法補充，而非取代 HOPL 的歷史重建。

程式語言的歷史需要人物，因為最初問題、審美、概念與勇氣往往確實來自具體的人。但人物不應吞噬共同體與制度。

完整語言歸因應同時回答：

- 誰提出？
- 誰共同設計？
- 誰實作？

- 誰批准？
- 誰維護？
- 誰出資？
- 誰承擔相容性？
- 誰透過採用或拒絕塑造結果？
- 誰在今日仍有權改變它？

本文提出：

> L⊠
> =
> cal-E
> (
> F,
> D,
> I,
> G,
> O,
> S,
> U,
> X
>)⊠

並將歸因分為：

> 【Credit
> +
> Causality
> +
> Authority
> +
> Accountability
> +

PLDST 之後研究任何設計師時，都不得把「某人是創始者」直接推導成：

此人造成所有特徵
此人代表全部社群
此人對今日決策仍有權力
此人應承擔全部缺陷
更精確的結論形式是：

> 某位設計者在某一時間相位，對某組可直接歸因的決策展現了穩定風格；該風格後來可能被共同設計、實作限制、組織需求、正式治理與生態選擇保留、修正、放大或抵消。
> 因此，程式語言設計師風格研究的成熟標準，不是能說出更多著名人物，而是能夠在承認個人創造力的同時，不抹去共同工作與制度因果。
>
> 最終原則為：

> 【承認創始者
> ∧
> 不把語言歷史簡化為創始者】

維度 問題

Credit 誰應獲得承認？

Causality 誰與哪些條件造成結果？

Authority 誰能批准或否決？

Accountability 誰應處理後果？

Maintenance 誰持續支付維護成本？

Founder／Initiator
Co-designer
Implementer
Library／Tool author
Technical leader
Governance body
Organization／Sponsor
Standards body
User／Ecosystem
External constraint

[R1] Go Project, “Frequently Asked Questions,” “Using Go at Google,” “Go, Open Source, Community,” and “How Go Was Made.”

— Go 由 Robert Griesemer、Rob Pike、Ken Thompson 共同啟動，以及其組織工程背景。

[R2] Python Enhancement Proposals, “PEP 13 – Python Language Governance,” “PEP 8000 – Python Language Governance Proposal Overview,” and “PEP 8001 – Python Governance Voting Process.”

— Python 從 BDFL 轉入 Steering Council 的正式治理歷史與選擇程序。

[R3] Rust Project, “RFC 0002 – RFC Process,” “RFC 1068 – Rust Governance,” and Rust Leadership Council repository.

— Rust 重大變更、團隊治理與現代專案權力結構。

[R4] ISO C++ Foundation, “The Committee: WG21,” “SD-4: WG21 Practices and Procedures,” and “SD-7: Mailing Procedures and How to Write Papers.”

— C++ 國際標準委員會、參與結構與提案程序。

[R5] Java Community Process, “JCP Procedures Overview,” “Program Overview,” and “JSR Overview.”

— Java 規格批准、Expert input、Reference Implementation 與 Technology Compatibility Kit。

[R6] Ecma TC39, “The TC39 Process,” TC39 official site, and ECMAScript specification repository.

— ECMAScript Proposal stages、Champion、Committee consensus、實作與年度規格。

[R7] ACM SIGPLAN, HOPL-IV Papers and Content Guidelines.

— 程式語言歷史對設計、實作、演化、標準化與社會影響的研究要求。

[R8] Python Enhancement Proposals, PEP 8010–8016 governance model proposals.

— Python 在 2018 年對多種治理制度的公開比較。

[R9] Go Project, “Contribution Guide,” “Go Wiki: Proposals,” and “Handling Issues.”

— 重大 Go 語言、API、工具與命令列變更的 Change Proposal Process。

[R10] Rust Project, RFC Book and governance repositories.

— RFC、Teams、Leadership Council 與技術決策分工。

[R11] ISO C++ Foundation, WG21 Standing Documents and Meetings and Participation.

— 多國專家、Working Groups、Papers、Meetings 與正式程序。

[F] Fact
[Q] Direct quotation or declared principle
[D-I] Individual decision
[D-C] Collective design decision
[D-O] Organization-driven decision
[D-G] Governance decision
[D-E] Ecosystem outcome
[I] Interpretation
[C] Counterevidence or controversy
[U] Uncertain attribution