程式不等於程式碼：可執行問題模型的基本定義

A Program Is Not Source Code: A Fundamental Definition of Executable Problem Models

v0.1 / 2026-07-27 / Neo.K with Aletheia / 初版完成

https://thisoneisneok.com/html/papers/pu-1-01.html

SECTION

A Program Is Not Source Code: A Fundamental Definition of Executable Problem Models

論文編號： PU-1-01

系列：「程式宇宙書系」第 1 冊《系統性程式設計》

作者： Neo.K with Aletheia

機構： EveMissLab／一言諾科技有限公司

版本： v0.1

日期： 2026 年 7 月 27 日

SECTION

摘要

程式設計教育長期將學習順序安排為語法、變數、條件、迴圈、函式、類別、框架與專案。然而，這種順序往往讓學習者誤以為「程式」就是一組文字指令，並把能寫出可執行程式碼視為已經完成程式設計。實際上，原始碼只是程式的一種表示；演算法只是程式的一種轉換結構；軟體是程式、資料、資源、介面與執行環境的集合；應用程式則是面向特定使用情境的軟體實例；系統更包含外部主體、制度、物理環境與持續運作關係。

本文提出：

> 【Program
> ≠
> Source Code】

並將程式初步定義為：

> 對某一問題世界之實體、狀態、事件、行動、規則、邊界與成功條件所形成的可執行結構化描述。
> 形式上：

> 【mathcal P_r
> =
> <
> W,
> E,
> S,
> V,
> A,
> R,
> B,
> I,
> O,
> F
> >】

其中：

- W：問題世界；
- E：實體與關係；
- S：狀態；
- V：事件；

- A：行動；
- R：規則與不變量；
- B：系統邊界；
- I：輸入與外部條件；
- O：輸出與世界效果；
- F：失敗、恢復與完成判準。

  原始碼只是在特定語言下對此結構的投影：

> Source_L
> =
> Π_L
> (
> mathcal P_r
>)

因此，同一程式可以具有不同來源文字、不同程式語言、不同 AST、不同 IR、不同部署形式，甚至不同物理載體。反之，兩份外觀近似的程式碼，也可能因世界模型、狀態邊界、權限、錯誤語意與 Runtime 不同，而不是同一程式。

本文區分六個常被混用的概念：

- Algorithm：對一類輸入進行有限或明確轉換的程序結構；
- Source Code：某程式語言中的文字或結構化表示；
- Program：可執行問題模型；
- Software：程式、資料、資源、設定、相依與工具鏈的整體產品；
- Application：面向特定角色與使用情境運作的軟體；
- System：包含軟體、硬體、主體、制度、外部服務與環境的持續運作整體。

  本文提出「問題世界先於來源文字」原則：

> 【Problem World
> →
> Executable Structure
> →
> Language Projection】

若省略前兩層而直接寫碼，設計者便容易把檔案結構當成系統結構，把函式名稱當成責任邊界，把資料表當成真實世界，把成功回傳值當成目的完成。

本文也提出「程式完整性」概念。一個只有成功路徑、沒有非法狀態、權限、例外、恢復、外部效果與驗證證據的程式，只是局部執行片段。程式完整性至少包含：

- 世界覆蓋；
- 狀態合法性；
- 行動前後條件；
- 邊界與相依；
- 失敗與恢復；
- 可觀測性；
- 完成判準；
- 語意可追蹤性。

本文使用排序演算法、待辦事項、訂單系統、網站部署、AI Agent 與長時程研究系統等案例，說明演算法、程式碼與完整程式之間的差異。本文亦提出可證偽研究綱領，包括程式碼量與問題覆蓋的相關性、跨語言程式同一性、世界模型缺失率、成功旗標錯配、程式完整性與維護成本、初學者教學順序及可視化外部記憶效應。

本文為後續〈問題世界建模〉奠定方法論起點：在寫任何來源文字之前，設計者必須先回答系統認為世界裡有哪些東西、它們如何變化、哪些狀態不合法，以及系統真正負責到哪裡。

關鍵詞： 程式本體論、程式碼、可執行問題模型、軟體、應用程式、系統、程式完整性、系統性程式設計

SECTION

Abstract

Programming education often begins with syntax and treats executable source code as the primary object of programming. This paper argues that a program is not identical to source code.

A program is defined as an executable structured description of a problem world, including entities, states, events, actions, rules, boundaries, inputs, outputs, failures, recovery, and completion criteria:

> mathcal P_r
> =
> <
> W,
> E,
> S,
> V,
> A,
> R,
> B,
> I,
> O,
> F
> >

Source code is only a language-specific projection of that structure:

> Source_L
> =
> Π_L
> (
> mathcal P_r
>)

The paper distinguishes algorithms, source code, programs, software, applications, and systems. It introduces the principle that the problem world precedes source text and develops a concept of program completeness based on world coverage, legal state, boundaries, failure, recovery,

The paper concludes that programming should begin with problem-world modeling rather than syntax.

Keywords: program ontology, source code, executable problem model, software, application, system, systematic programming

一段文字能被編譯、解譯或執行，不代表它已經形成完整程式。

例如：

```
print("Hello, world!")
```

它確實可執行。

但它沒有處理：

- 誰需要這個輸出；
- 輸出後世界如何改變；
- 失敗時怎麼辦；
- 誰有權執行；
- 完成如何驗證；
- 是否存在持續狀態；
- 是否與外部系統互動。

  這並不表示它不是程式片段，而是說「可執行」與「完整程式」是不同層級。

  程式碼是表達。

  程式是結構。

  軟體是被交付與維持的整體。

  系統是持續運作的世界關係。

SECTION

2.1 Algorithm

演算法是針對某類輸入，依明確規則產生輸出或狀態轉換的程序結構。

形式上：

$$
A
:
X
\to
Y
$$

或：

$$
S_{t+1}
=
A
(
S_t,
x_t
)
$$

演算法不必包含使用者、權限、UI、部署、持久化、法律責任與外部恢復。

SECTION

2.2 Source Code

原始碼是某語言中的表示：

$$
C_L
\in
\mathcal{L}
$$

它可能是文字、AST、視覺節點、結構化表單、DSL 或符號圖。

SECTION

2.3 Program

程式是能夠在某 Runtime 中，依據問題世界模型，對合法狀態執行受規則約束轉換的結構。

SECTION

2.4 Software

軟體通常包括一或多個程式、資料、設定、靜態資源、相依套件、部署結構、工具鏈與文件。

SECTION

2.5 Application

應用程式是面向特定角色、目的與使用情境的軟體運作實例。

SECTION

2.6 System

系統包含軟體、硬體、使用者、組織、制度、外部服務、資料來源、實體環境、持續時間、維護與治理。

因此：

> Algorithm
> subsetneq
> Program
> subsetneq
> Software
> ⊆
> System

此包含關係是概念性的，不代表每個實例都能以簡單集合關係完整描述。

本文定義：

> 程式是對某一問題世界之實體、狀態、事件、行動、規則、邊界、輸入、輸出、失敗與完成條件所形成的可執行結構化描述。

形式上：

> 【mathcal P_r
> =
> <
> W,
> E,
> S,
> V,
> A,
> R,
> B,
> I,
> O,
> F
> >】

SECTION

3.1 問題世界 W

系統試圖理解與改變的領域。

SECTION

3.2 實體 E

世界中被承認的對象及其關係。

SECTION

3.3 狀態 S

世界在特定時間的合法情況。

SECTION

3.4 事件 V

觸發狀態重新評估的發生。

SECTION

3.5 行動 A

系統或主體對世界施加的轉換。

SECTION

3.6 規則 R

前置條件、後置條件、不變量與合法性。

SECTION

3.7 邊界 B

系統負責與不負責的範圍。

SECTION

3.8 輸入 I

來自外部世界的資料、命令、感測與事件。

SECTION

3.9 輸出 O

對數位或物理世界產生的效果與證據。

SECTION

3.10 失敗與完成 F

錯誤、回復、補償、驗證與完成判準。

對程式 mathcal P_r，使用語言 L 表示：

> Source_L
> =
> Π_L
> (
> mathcal P_r
>)

不同語言會強調函數、物件、型別、資料流、規則、並行、效果或視覺節點等不同結構。

同一問題模型可以被表示為 Python、Rust、SQL、狀態圖、AST、IR、工作流、規則表或自然語言規格。

同一份程式碼若執行於不同 Runtime、設定、權限、資料、外部服務與世界假設，可能形成不同實際程式行為。

所以：

> SameSource
> not⇒
> SameProgramBehavior

本文提出：

> 【Problem World
> →
> Executable Structure
> →
> Language Projection】

SECTION

5.1 問題世界不是需求句子

「做一個訂單系統」不是問題世界。

它只是指向一個尚未展開的領域。

真正的問題世界至少要回答：

- 誰購買；
- 誰販售；
- 商品與庫存如何存在；
- 付款何時成立；
- 配送由誰完成；
- 爭議如何進入；
- 哪些狀態不可同時成立；
- 哪些外部系統不受本系統控制。

SECTION

5.2 世界假設

每個程式都包含隱含假設。

例如：

使用者具有唯一身份。
時間順序可被信任。
付款通知不會重複。
網路最終會恢復。
資料庫是權威狀態。

若這些假設沒有被明示，它們仍會存在，只是變成隱藏風險。

SECTION

5.3 問題與功能不同

功能描述：

使用者可以新增任務。
問題描述則要進一步問：

- 為何要新增；
- 任務屬於誰；
- 是否可以共享；
- 何時失效；
- 是否可撤回；
- 完成狀態由誰判定；
- 通知是否等於任務完成。

功能是世界行動的一部分，不是世界本身。

SECTION

6.1 可執行不只指 CPU 執行

本文中的「可執行」包含：

- 可由機器執行；
- 可由 Runtime 解釋；
- 可由工作流推進；
- 可由人機協作程序履行；
- 可驗證狀態是否發生變化。

SECTION

6.2 結構化描述

一個可執行結構應能回答：

- 當前世界狀態是什麼；
- 哪個事件發生；
- 哪個主體要求何種行動；
- 是否具有權限；
- 前置條件是否成立；
- 應產生何種狀態差分；
- 失敗時保留什麼；
- 如何驗證結果。

SECTION

6.3 轉換形式

$$
\Delta W\boxtimes = F(W\boxtimes, e\boxtimes, a\boxtimes, p\boxtimes, r\boxtimes)
$$

其中：

- W⊠：當前問題世界狀態；
- e⊠：事件；
- a⊠：行動；
- p⊠：權限與主體；
- r⊠：規則。

SECTION

6.4 沒有世界差分的程式

某些程式看似只計算數值，例如排序、壓縮與解析。

但它們仍在計算世界中改變：

- 記憶；
- 順序；
- 表示；
- 可見性；
- 後續決策條件。

所以世界差分不必是物理行動，也可以是數位狀態差分。

SECTION

7.1 同語言不等於同程式

兩個專案都使用 Python，不代表它們相同。

SECTION

7.2 同演算法不等於同程式

相同排序演算法可被用於：

- 商品價格；
- 醫療優先級；
- 搜尋結果；
- 社群推薦。

  演算法相同，但問題世界、價值與後果完全不同。

SECTION

7.3 同來源碼不等於同實際程式

若環境變數、資料、權限或相依版本不同，實際行為也不同。

SECTION

7.4 同問題模型的跨語言實現

若兩個實現保存：

- 同一實體語意；
- 同一合法狀態；
- 同一轉換規則；
- 同一錯誤與完成條件；
- 可接受的外部效果；

  則可以視為同一程式的不同語言投影。

SECTION

7.5 程式同一性條件

暫定：

> SameProgram(P□,P□)
> ⇔
> C_W
> ∧
> C_S
> ∧
> C_R
> ∧
> C_F
> ∧
> C_E

其中：

- C_W：問題世界相容；
- C_S：狀態語意相容；
- C_R：規則相容；
- C_F：失敗與完成相容；
- C_E：外部效果相容。

  不要求位元完全相同。

SECTION

8.1 排序演算法

排序演算法可表示為：

> Sort
> :
> X□
> →
> X□

它回答如何重新排列元素。

但完整排序程式還需要知道：

- 依什麼欄位排序；
- 缺失值如何處理；
- 是否穩定排序；
- 權限是否允許看到全部元素；
- 排序結果如何被使用；
- 錯誤是否會改變原資料。

SECTION

8.2 推薦演算法

即使演算法只做排名，完整程式仍需處理：

- 候選內容如何進入；
- 哪些內容被排除；
- 使用者能否關閉；
- 推薦是否可解釋；
- 受影響創作者能否申訴；
- 排名是否改變現實分配。

SECTION

8.3 演算法的世界嵌入

> Program
> =
> Algorithm

但此式仍不完整，還需要狀態、權限、失敗與驗證。

SECTION

9.1 可見性偏差

程式碼最容易被：

- 編輯；
- 計數；
- 儲存；
- 比較；
- 審查；
- 執行。

問題世界、責任與隱含假設則較不容易看見。

SECTION

9.2 工具偏差

IDE、版本控制與編譯器主要操作程式碼，因此開發者容易把工具可見物當成全部程式。

SECTION

9.3 教學偏差

初學者先學語法，便自然把「會寫語法」當成「會設計程式」。

SECTION

9.4 交付偏差

專案常以：

- 功能已完成；
- 程式碼已提交；
- 測試已通過；

作為完成證據，卻沒有檢查世界結果。

SECTION

9.5 文字本體偏差

文字程式語言長期成為主流，使人們誤認為程式必須先以文字存在。

但文字只是其中一種高效投影。

SECTION

10.1 資料夾是物理或工具組織

資料夾可依：

- 檔案類型；
- 技術層；
- 功能；
- 團隊；
- 部署單元；

排列。

SECTION

10.2 系統結構由責任與狀態決定

真正的系統邊界應回答：

- 哪個模組擁有哪些狀態；
- 誰可以修改；
- 哪些規則由誰維持；
- 哪些失敗由誰恢復；
- 外部依賴如何隔離。

SECTION

10.3 同資料夾多責任

一個 utils 資料夾可能混合多種無關責任。

SECTION

10.4 多資料夾單一責任

同一訂單責任可能分散於：

- API；
- domain；
- database；
- frontend；
- events；
- tests。

因此：

> FolderBoundary
> not⇒
> ResponsibilityBoundary

SECTION

11.1 儲存投影

資料表是為查詢、交易與效能建立的儲存結構。

SECTION

11.2 實體可能跨多表

一個使用者身份可能分散於：

- 帳號；
- 個人資料；
- 權限；
- 訂閱；
- 關係；
- 稽核歷史。

SECTION

11.3 一表可能不是一實體

事件表、關聯表、快取表與投影表，不必是獨立世界實體。

SECTION

11.4 資料庫權威問題

系統必須知道哪些資料是：

- 權威狀態；
- 快取；

- 投影；
- 歷史；
- 衍生結果。

  把資料表直接視為世界，會讓儲存需求僭位成存在論。

  本文將程式完整性定義為：

> 【mathcal C_P
> =
> <
> C_W,
> C_S,
> C_A,
> C_B,
> C_F,
> C_O,
> C_T
> >】

其中：

- C_W：世界覆蓋；
- C_S：狀態合法性；
- C_A：行動與規則完整；
- C_B：邊界明確；
- C_F：失敗與恢復；
- C_O：可觀測與驗證；
- C_T：語意可追蹤。

SECTION

12.1 世界覆蓋

系統是否表示真正相關的實體、主體、事件與例外。

SECTION

12.2 狀態合法性

是否定義：

- 合法狀態；
- 非法狀態；
- 未知狀態；
- 爭議狀態；
- 終止狀態。

SECTION

12.3 行動完整

每個重大行動是否有：

- 執行者；
- 權限；
- 前置條件；
- 狀態差分；
- 外部效果；
- 證據。

SECTION

12.4 邊界完整

是否知道：

- 哪些外部服務不受控制；
- 哪些資料不可信；
- 哪些物理效果需由人類完成；
- 哪些制度判定不應由程式替代。

SECTION

12.5 失敗完整

是否處理：

- 部分完成；
- 重複事件；
- 超時；
- 網路中斷；
- 權限失敗；
- 外部服務成功但本地失敗；
- 無法回滾效果。

SECTION

12.6 觀測完整

是否能證明：

- 發生了什麼；
- 誰執行；
- 何時執行；
- 使用何種規則；
- 世界是否真的改變。

SECTION

12.7 語意追蹤

是否能從使用者目的追蹤到：

需求
→ 世界實體
→ 狀態
→ 規則
→ 原始碼
→ 執行證據
→ 世界結果

SECTION

13.1 計算成功

函式回傳：

{"success": true}
只表示某計算步驟依內部條件成功。

SECTION

13.2 系統成功

系統狀態已達到預定合法狀態。

SECTION

13.3 世界成功

外部數位或物理效果已經發生。

SECTION

13.4 意圖成功

原始目的被受影響主體接受。

因此：

> Success_C
> not⇒
> Success_S
> not⇒
> Success_W
> not⇒
> Satisfied_I

SECTION

13.5 完成分層

成熟程式可使用：

```
completion:
 computation: "complete"
 system_state: "complete"
 external_effect: "pending-verification"
 intent_acceptance: "awaiting-human-review"
```

SECTION

14.1 邊界不是部署圖的外框

邊界定義：

- 系統負責什麼；
- 系統只能觀測什麼；

- 系統不能保證什麼；
- 誰必須接手；
- 何種效果需要外部證據。

SECTION

14.2 外部依賴

付款、物流、身分驗證、郵件、雲端服務與人類批准，都可能位於邊界外。

SECTION

14.3 邊界協議

程式需要明示：

> Contract
> =
> <
> Input,
> Assumption,
> Output,
> Failure,
> Evidence
> >

SECTION

14.4 邊界錯誤

最危險的錯誤之一，是系統宣稱完成超出自己可觀測範圍的事情。

例如寄送 API 回傳成功，只能證明請求被接受，不一定證明收件人已閱讀。

SECTION

15.1 原始碼不自己執行

程式需要：

- 編譯器或解譯器；
- 作業系統；
- 記憶模型；
- 程式庫；
- 權限；
- 裝置；
- 排程；
- 外部服務。

SECTION

15.2 Runtime 語意

相同來源碼在不同 Runtime 可能具有不同：

- 數值精度；
- 並行順序；
- 錯誤處理；
- 時間；
- I/O；
- 資源限制。

SECTION

15.3 Runtime 合約

完整程式應明示：

- 版本；
- 相依；
- 配置；
- 資源；
- 權限；
- 外部接口；
- 失效條件。

SECTION

15.4 環境漂移

即使來源碼未變，Runtime 或依賴更新也可能改變實際程式。

所以版本控制只保存原始碼，未必保存完整可重現程式。

SECTION

16.1 多投影

完整程式可能同時具有：

- 問題世界圖；
- 狀態圖；
- 資料模型；

- 原始碼；
- AST；
- IR；
- 測試；
- 執行日誌；
- UI；
- 文件。

SECTION

16.2 權威來源問題

若不同投影衝突，必須知道：

- 哪一層是權威；
- 如何同步；
- 誰能修改；
- 如何驗證一致。

SECTION

16.3 原始碼不是唯一權威

在某些系統中：

- schema 是權威；
- 狀態機是權威；
- 規則表是權威；
- 工作流定義是權威；

- Intent IR 是權威。

  原始碼可能只是編譯產物。

SECTION

16.4 程式本體與投影

本文暫定：

> Program Ontology
> ≠
> Any Single Projection

程式本體更接近各投影共同指向的可執行問題結構。

SECTION

17.1 演算法

快速排序、合併排序或其他排序程序。

SECTION

17.2 程式碼

用某語言寫出的 sort()。

SECTION

17.3 程式

需定義：

- 排序對象；
- 比較規則；

- 缺失值；
- 穩定性；
- 權限；
- 錯誤；
- 是否修改原集合；
- 結果用途。

SECTION

17.4 系統

若排序用於醫療資源分配，還需處理：

- 公平；
- 申訴；
- 緊急例外；
- 規則版本；
- 人類覆核。

相同演算法進入不同世界，便形成不同程式責任。

SECTION

18.1 初級程式碼

建立、修改、刪除任務。

SECTION

18.2 問題世界

需要：

- 使用者；
- 任務；
- 專案；
- 期限；
- 擁有者；
- 分享；
- 提醒；
- 完成證據；
- 取消與不適用。

SECTION

18.3 狀態

proposed
active
blocked
completed
cancelled
disputed

SECTION

18.4 完整性問題

completed=true 是否由使用者自行判定？

共享任務由誰完成？

過期任務是否自動失效？

刪除是否保留歷史？

這些都不是 CRUD 語法能自動回答。

SECTION

19.1 程式碼片段

```
order.status = "paid"
```

SECTION

19.2 真正程式問題

- 款項是否已入帳；
- 是否重複通知；
- 庫存是否保留；
- 付款後能否取消；
- 退款如何補償；
- 多方資料不一致時誰是權威。

SECTION

19.3 部分失敗

可能發生：

```
付款成功
→ 本地更新失敗
→ 庫存未扣除
→ 使用者再次付款
```

完整程式必須處理冪等、補償與證據。

SECTION

20.1 原始碼完成

頁面與功能已寫完。

SECTION

20.2 軟體完成

還需要：

- 資源；
- 設定；
- 相依；
- 建置；
- 測試；
- 安全掃描。

SECTION

20.3 應用完成

還需要：

- 網域；
- 身分；
- 使用流程；
- 內容；
- 無障礙；
- 裝置相容。

SECTION

20.4 系統完成

還需要：

- 部署；
- 監控；
- 備份；
- 回滾；
- 營運責任；
- 真實使用者驗證。

  所以「程式碼完成」只是整體鏈中的一個節點。

SECTION

21.1 模型輸出不等於 Agent 程式

一個模型可以產生文字或工具調用建議，但持續 Agent 還需要：

- 身分；
- 目標；
- 記憶；
- 計畫；
- 工具；
- 權限；
- 預算；
- 檢查點；
- 停止條件；
- 人類接管。

SECTION

21.2 任務完成不等於目的完成

Agent 可能成功修改檔案，但：

- 修改是否正確；
- 是否破壞其他功能；
- 是否符合使用者意圖；
- 是否有權發布；
- 是否能回復；

  仍需獨立判定。

SECTION

21.3 Agent 程式模型

> P_(agent)
> =
> <
> Goal,
> Memory,
> Plan,
> Capability,
> Permission,
> WorldState,
> Evidence,
> Recovery
> >

提示詞只是其中一種來源表示。

SECTION

21.4 Agent 的系統邊界

Agent 不應因能呼叫工具，就被視為能保證外部世界結果。

工具成功、系統狀態與人類接受仍需分層。

SECTION

22.1 單篇輸出

一篇論文文字只是研究程式的輸出投影。

SECTION

22.2 持續研究程式

完整研究程式還包含：

- 問題；
- 已知文獻；
- 定義；
- 命題；
- 證據；
- 反例；
- 計算實驗；
- 失敗紀錄；
- 版本；
- 後續節點。

SECTION

22.3 研究同一性

若只保留最終文件而遺失推導、失敗與依賴，便難以重建研究程式。

SECTION

22.4 研究完成

文件存在，不代表命題已驗證。

因此：

> DocumentComplete
> not⇒
> ResearchComplete

SECTION

23.1 傳統順序

語法
→ 小題目
→ 資料結構
→ 物件導向
→ 框架
→ 專案
學生常到專案階段才被迫自行發明系統思維。

SECTION

23.2 建議順序

> 【問題
> →
> 世界
> →

SECTION

23.3 語法仍然重要

本文不是否定語法。

語法是精確表達與執行的必要工具。

但語法應服務結構，而不是取代結構。

SECTION

23.4 小型練習也能建立世界觀

即使教條件判斷，也可先問：

- 世界有哪些狀態；
- 哪些狀態合法；
- 條件判斷代表哪一條規則；
- 另一條分支是否真的是完整例外。

SECTION

24.1 程式結構不能只存在於作者腦中

當專案變大，單靠來源文字難以同時看見：

- 實體關係；
- 狀態流；
- 事件時間；
- 模組依賴；
- 權限；

- 外部效果。

SECTION

24.2 可視化不等於裝飾

狀態圖、世界圖、資料流與責任圖，是程式不同結構的外部投影。

SECTION

24.3 圖也不是本體全部

圖同樣具有表示落差。

所以需要多投影互相驗證，而不是用一張「全系統圖」冒充完整世界。

SECTION

24.4 團隊共享

若團隊無法用共同表示回答「系統到底認為世界是什麼」，程式知識便高度集中於少數人。

- 可執行即完整： 能運行就宣稱程式完成。
- 來源碼本體化： 把某語言表示當成程式全部。
- 演算法僭位： 有演算法就忽略世界嵌入與制度後果。
- 功能清單僭位： 功能列表被當成問題世界。
- 檔案夾架構化： 以資料夾排列取代責任與狀態邊界。
- 資料表存在論： 儲存投影被當成世界實體本身。
- 成功旗標僭位： 內部成功被當成外部目的完成。
- Runtime 遺忘： 忽略環境、權限、版本與相依。
- 邊界隱藏： 系統對超出觀測範圍的結果做保證。

- 只有成功路徑： 未處理部分完成、重複與補償。
- 錯誤皆例外： 把可預期世界狀態都當成技術例外。
- 測試即證明世界成功： 內部測試通過被當成真實效果。
- 同碼即同行為： 忽略設定、資料與外部系統差異。
- 跨語言即重寫： 語言改變時沒有保存問題語意。
- 模型輸出即 Agent： 忽略持續身份、記憶、權限與恢復。
- 文件完成即研究完成： 輸出投影取代研究證據。
- 語法先行永久化： 初學者長期停留於文字技巧。
- 單一投影權威： 一張圖、一份碼或一份規格被視為全部程式。
- 維護者猜測世界： 關鍵假設未被外部化。
- 程式員英雄化： 系統依賴個人腦內結構，而非共享方法。

SECTION

26.1 程式碼量與問題覆蓋

測量程式碼量增加是否真的提高：

- 世界狀態覆蓋；
- 例外覆蓋；
- 邊界清晰度；
- 完成驗證。

預期二者不具有穩定線性關係。

SECTION

26.2 世界模型缺失率

從事故、需求變更與申訴中統計，多少問題源於：

- 實體遺漏；
- 狀態遺漏；
- 主體遺漏；
- 邊界誤判。

SECTION

26.3 跨語言同一性

將同一問題模型以不同語言實現，測量狀態、規則、失敗與外部效果的語意保持。

SECTION

26.4 同來源環境差異

在不同 Runtime、資料與設定中執行同一來源碼，測量行為差異。

SECTION

26.5 成功旗標錯配率

統計內部 success 與外部效果、使用者接受不一致的事件。

SECTION

26.6 程式完整性與維護成本

研究完整性指標較高的專案，是否具有較低：

- 修復時間；
- 交接時間；
- 變更影響；

- 回復成本。

SECTION

26.7 教學順序實驗

比較：

- 語法先行；
- 問題世界先行；

兩組學習者在陌生專案中的設計、除錯與解釋能力。

SECTION

26.8 可視化外部記憶效應

測量狀態圖、世界圖與責任圖對：

- 團隊理解；
- 新成員上手；
- 錯誤發現；
- 修改信心；

的影響。

SECTION

26.9 權威投影一致性

測量規格、圖、程式碼、測試與 Runtime 行為之間的差異率。

SECTION

26.10 邊界聲明準確率

檢查系統文件所宣稱責任範圍，是否與實際可觀測及可控制範圍一致。

SECTION

26.11 研究程式可重建性

只憑最終文件與包含失敗、版本、證據的完整研究包，分別測量第三方重建研究路徑的能力。

SECTION

26.12 初學者英雄依賴

研究缺少外部結構表示的團隊，是否更依賴少數核心作者。

- 程式不等於原始碼。
- 原始碼是程式在特定語言中的投影。
- 演算法是轉換結構，不自動包含完整問題世界。
- 軟體包含程式、資料、資源、設定、相依與交付結構。
- 應用程式是面向角色與情境的軟體運作實例。
- 系統包含軟體以外的硬體、主體、制度與環境。
- 問題世界應先於來源文字。
- 功能清單不等於問題世界。
- 程式的核心是受規則約束的世界狀態轉換。
- 同一程式可以具有多種語言與結構投影。
- 同一來源碼在不同 Runtime 中不必產生相同行為。
- 檔案結構不等於責任邊界。
- 資料表不等於世界實體。
- 程式完整性包含世界、狀態、行動、邊界、失敗、觀測與追蹤。

- 計算成功不等於系統成功、世界成功或意圖滿足。
- Runtime、權限與外部相依是實際程式定義的一部分。
- 程式本體不等於任何單一投影。
- 可視化是程式結構的外部記憶，但同樣需要邊界與驗證。
- 程式教育應從問題世界與狀態開始，而非永久停留於語法。

20.

> 【真正的程式設計，】

> 【不是把想法翻譯成程式碼，】

> 【而是把問題世界建造成可理解、可執行、】

> 【可驗證且可維護的結構。】

SECTION

28.1 承接第 0 冊

第 0 冊已建立：

- 意圖、計算與物理三宇宙；
- 耦合與表示落差；
- 數位人工存在；
- 生成與限制；
- 計算目的與價值邊界。

本文把這些地基帶入程式方法論，指出程式必須保存問題世界、合法狀態、邊界與目的，而非只保存來源文字。

SECTION

28.2 銜接 PU-1-02

下一篇將正式建立問題世界建模法，處理：

- 實體；
- 屬性；
- 關係；
- 規則；
- 角色；
- 外部世界；
- 系統邊界；
- 未知與假設。

也就是回答：

> 寫碼以前，究竟要把哪一個世界建出來？

程式碼很重要。

沒有精確語言，程式無法穩定被機器執行。

但程式碼不是起點，也不是全部。

在來源文字以前，已經存在：

- 問題世界；
- 主體目的；
- 實體與關係；
- 合法與非法狀態；

- 事件與行動；
- 責任與邊界；
- 失敗與恢復；
- 世界效果與完成證據。

  來源碼只是把這些結構投影到特定語言。

  若投影之前沒有結構，程式碼就會成為隱含假設的堆積。

  若投影之後沒有驗證，程式碼就會把內部成功冒充世界成功。

  因此，本文將程式重新定義為：

> 【Program
> =
> Executable Problem Model】

更完整地說：

> 【Program
> =
> World Structure
> +
> Legal State Transitions
> +
> Boundary Contracts
> +
> Failure and Recovery
> +
> Observable Completion】

程式設計的第一個問題，不應是：

我要使用哪一個語言？

而應是：

系統認為世界裡有哪些東西？

它們如何變化？
哪些狀態永遠不應出現？
誰能做什麼？
系統負責到哪裡？
失敗後如何保留世界？
完成究竟由什麼證據成立？

本文的最終命題是：

> **【當程式被理解為可執行問題模型，】**

> **【程式設計才會從語法技巧，】**

> **【轉化為可以教學、協作、驗證與治理的系統方法。】**

```
program_model:
 program_id: "order-payment-v1"
 problem_world:
  entities:
   - "order"
   - "payment"
   - "customer"
   - "merchant"
  external_entities:
   - "payment-provider"
 states:
  order:
   - "created"
   - "payment-pending"
   - "paid"
   - "cancelled"
   - "disputed"
 actions:
  confirm_payment:
   actor: "payment-provider"
   preconditions:
    - "order.status == payment-pending"
   effects:
    - "order.status = paid"
   idempotent: true
 boundaries:
  guarantees:
   - "local order state update"
  does_not_guarantee:
```

```
    - "bank settlement finality"
  failure:
   retryable:
    - "network-timeout"
   compensating:
    - "duplicate-charge"
  completion:
   computation: "event-processed"
   system: "order-paid"
   external: "settlement-unverified"
   intent: "customer-confirmation-pending"
```

概念	核心內容	不必包含
Algorithm	輸入到輸出的轉換程序	使用者、部署、制度
Source Code	特定語言表示	完整世界與外部效果
Program	可執行問題模型	完整組織與所有硬體
Software	程式、資料、資源與交付物	全部外部制度
Application	面向特定情境的軟體	整個社會技術網路
System	軟硬體、主體、制度與環境	—

```
program_completeness:
  world_coverage:
   entities_defined: true
   affected_parties_visible: partial
  state:
   legal_states_defined: true
   unknown_state_supported: true
  action:
   permissions_defined: true
   preconditions_defined: true
   external_effects_declared: true
  boundary:
   guarantees_declared: true
   non_guarantees_declared: true
  failure:
   retry_defined: true
   compensation_defined: partial
   irreversible_effects_visible: true
  observability:
   evidence_defined: true
   actor_attribution: true
```

traceability:
intent_to_code: true
code_to_world_evidence: partial

- PU-1-01　程式不等於程式碼：可執行問題模型的基本定義
- PU-1-02　問題世界建模：實體、關係、規則與系統邊界
- PU-1-03　資料—狀態—事件—行動：程式系統的四元動力結構
- PU-1-04　責任、模組與契約：從檔案分類到系統邊界
- PU-1-05　可視化作為外部結構記憶：多角色、多尺度的程式理解
- PU-1-06　失敗也是程式：驗證、可觀測、恢復與長期維護

SECTION

Neo.K／EveMissLab 相關理論

- Neo.K with Aletheia，《三重宇宙論：意圖、計算與物理存在的基本區分》，2026。
- Neo.K with Aletheia，《三宇宙耦合動力學：從想要、表示到世界改變》，2026。
- Neo.K with Aletheia，《表示落差：意圖、計算模型與物理現實之間的不可完備映射》，2026。
- Neo.K with Aletheia，《數位宇宙作為人工存在域：實體、狀態、規則與歷史》，2026。
- Neo.K with Aletheia，《生成與限制：計算機宇宙如何創造並封閉可能性》，2026。
- Neo.K with Aletheia，《我們到底想要什麼：計算目的、價值邊界與世界選擇》，2026。
- Neo.K with Aletheia，《結構先於文字：Nova 與後文本程式語言本體論》，2026。

SECTION

一般理論背景

- Turing, A. M., “On Computable Numbers, with an Application to the Entscheidungsproblem,” 1936.

- Dijkstra, E. W., “Notes on Structured Programming,” 1970.
- Parnas, D. L., “On the Criteria To Be Used in Decomposing Systems into Modules,” 1972.
- Brooks, F. P., The Mythical Man-Month, 1975.
- Jackson, M., Software Requirements & Specifications, 1995.
- Gamma, E. et al., Design Patterns, 1994.
- Evans, E., Domain-Driven Design, 2003.
- Fowler, M., Patterns of Enterprise Application Architecture, 2002.
- Harel, D., “Statecharts: A Visual Formalism for Complex Systems,” 1987.
- Lamport, L., Specifying Systems, 2002.
- Meyer, B., Object-Oriented Software Construction, 1997.

SECTION

v0.1 — 2026-07-27

- 正式進入第 1 冊《系統性程式設計》。
- 完成十八篇地基論文第七篇。
- 區分 Algorithm、Source Code、Program、Software、Application 與 System。
- 將程式定義為可執行問題模型。
- 建立程式十元基本結構。
- 提出問題世界先於來源文字原則。
- 建立跨語言程式同一性條件。
- 分析演算法、檔案結構與資料表的僭位問題。
- 建立程式完整性七元模型。

- 區分計算、系統、世界與意圖四層成功。
- 將 Runtime、邊界與外部相依納入程式定義。
- 建立程式多投影與權威結構問題。
- 加入排序、待辦、訂單、網站部署、AI Agent 與研究系統案例。
- 提出二十類失敗模式與十二項可證偽研究方向。
- 完成與 PU-1-02《問題世界建模》的銜接。