本專案的目標是開發一個可擴充的五子棋對戰系統，並實作具備不同策略等級的 AI 代理人 (Agents)。透過這個專案深入理解 OOP 的四大核心原則，同時探索 啟發式演算法 (Heuristic Algorithm) 在賽局決策中的應用。其中完成了一個圖形化介面 (GUI) 的遊戲環境，更設計了從隨機 (Random) 到貪婪 (Greedy)，再到具備攻守評估能力的智慧 (Smart) AI，成功模擬了人類的下棋思維

在架構設計上的 OOP 四大核心原則 ：

抽象 (Abstraction) 的力量： 定義了 BaseAgent 作為抽象基底類別，強制規範所有子類別必須實作 choose_action 方法。這讓我在開發初期就確立了明確的介面合約，避免了後續開發時介面不一致的問題

繼承 (Inheritance) 減少代碼冗餘： 讓 SmartAgent 繼承自 GreedyAgent。這意味著 Smart Agent 天生就具備了「一步致勝」和「一步防守」的基礎能力，不需要重寫代碼。我只需在子類別中專注於撰寫複雜的分數評估邏輯 (_evaluate_board)

多型 (Polymorphism) 與解耦： 在 GomokuArena (競技場) 中，程式碼只需呼叫 agent.choose_action()，完全不需要知道當下的對手是隨機的還是智慧的。這種低耦合的設計，讓我在測試階段可以隨意抽換對手進行對戰測試。

封裝 (Encapsulation) 保護邏輯： 將複雜的數學運算（如 _check_win_simulation 和 _get_position_score）封裝為私有方法。這讓主邏輯保持乾淨，外部使用者無法誤用這些內部運算，體現了封裝的安全性。

再來在 AI 邏輯的實作上，經歷了從「規則式」到「評分式」的思維轉變，解決了幾個問題：

1.如何平衡進攻與防守？

最初 AI 容易陷入「貪攻」的陷阱，為了造一個活三而忽略了對手的活四。 解決方案：我引入了加權公式 Score = MyScore - (OpponentScore * 0.9)。這讓我學會了賽局理論中的基本概念——最佳的決策不僅是讓自己得分，更要最大化地減少對手的得分。係數 0.9 的設定，讓 AI 在保持侵略性的同時，不會對防守視而不見。

2.分數權重的設定 (The Weighting Problem)

我曾困惑於為何 AI 在對手快贏時仍不防守。 反思與修正：我意識到必須建立「級距 (Order of Magnitude)」。我將「連五」設定為 10,000,000 分，而「活四」僅為 100,000 分。這 100 倍的差距是為了在數學上模擬「絕對勝利」與「不確定性」的區別。這讓我學到：在演算法中，數值的大小不只是數字，更代表了決策的優先級與邏輯意義。

圖形介面整合：在使用 Pygame 整合 Gym 環境時，遇到了渲染更新不同步的問題。我通過分離 render() 邏輯並建立獨立的視窗初始化檢查 (if self.window is None)，解決了視窗閃爍與卡頓的問題

NumPy 的維度處理：在處理棋盤座標時，曾混淆 (x, y) 與 (row, col) 的關係。透過統一使用 NumPy 矩陣操作與 flatten() 方法，確保了座標轉換的準確性。

這個專案是我第一次從「寫程式 (Coding)」跨越到「軟體設計 (Software Design)」

中間學到了不少東西

ex.

架構先行：良好的 UML 規劃能讓 coding 過程事半功倍。

數據驅動決策：AI 的聰明程度取決於評分函數 (Evaluation Function) 的設計細膩度。

除錯思維：透過觀察 AI 的「笨」行為（如自殺式下法），反推程式碼邏輯漏洞，是極佳的學習方式。