Reversi (Othello)

Agenda

Introduction & Origin

Game Rules

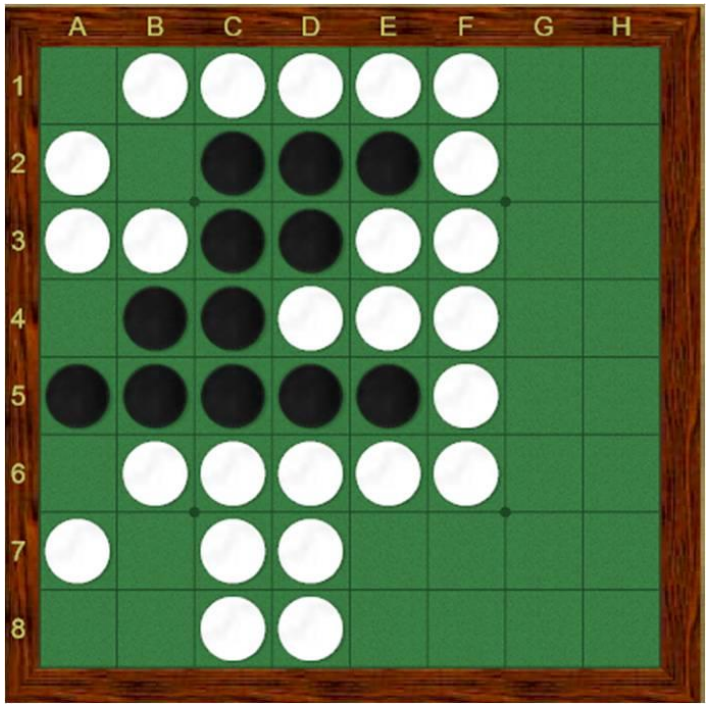
Project Architecture

Software Development Life Cycle

Conclusion

Introduction & History

Reversi, also known as Othello, dates back to the late 19th century. The game is believed to have been created around 1883 in England. Credit is often attributed to Lewis Waterman or Jhon W. Mollett, though the true origin remains debated. The name “Othello” was inspired by Shakespeare’s play due to the contrasting black and white pieces.



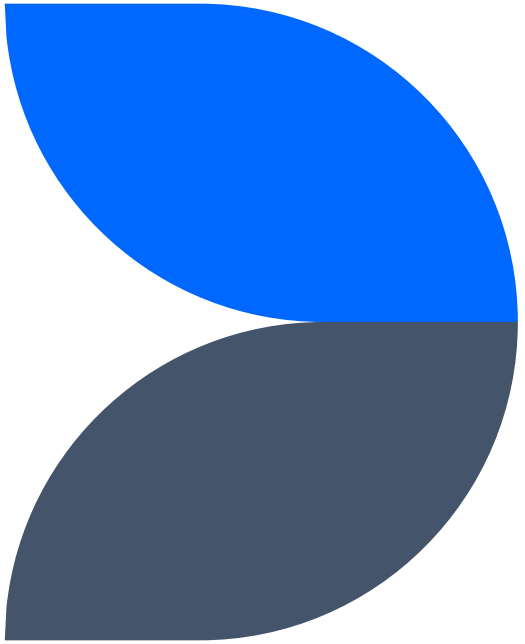
Game Rules

- **Game is played on an 8×8 board** and starts with four coins placed at the center: two black and two white in diagonal arrangement.
- **Players take turns placing one coin** of their color on an empty square.
- **A placed coin must flank opponent's coins** (horizontally, vertically, or diagonally) between the placed coin and another coin of the same color.
- **All opponent's coins that fall between two matching coins are flipped**, becoming the color of the player who made the move.
- **A move is only legal if it flips at least one opponent piece,** otherwise the player can't place their coin.
- **If a player has no legal move**, they must pass, and the opponent continues.
- **If neither player can make a move**, typically when the board is full or no legal move exists, the game ends.
- **The player with the majority of colored coins at the end wins**, counted by total flips made throughout the game.
- **Corner positions are the most advantageous**, because coins placed there cannot be flipped later, influencing strategic play.

Project Architecture

Domain	Pure domain logic: entities, rules, board, types, exceptions
Applications	Use cases. Orchestrates domain and calls ports (interfaces).
Ports	Define interfaces (abstract base classes) the application depends on — UI, AI, persistence.
Adapters	Concrete implementations of ports. Examples: ConsoleUI, PygameUI, RandomAI, MinimaxAI.
Infrastructure	Logging, configuration, DI (Dependency Injection) wiring.
Launchers	Compose dependencies and start the app.

Software Development Life Cycle



1. Requirement Analysis

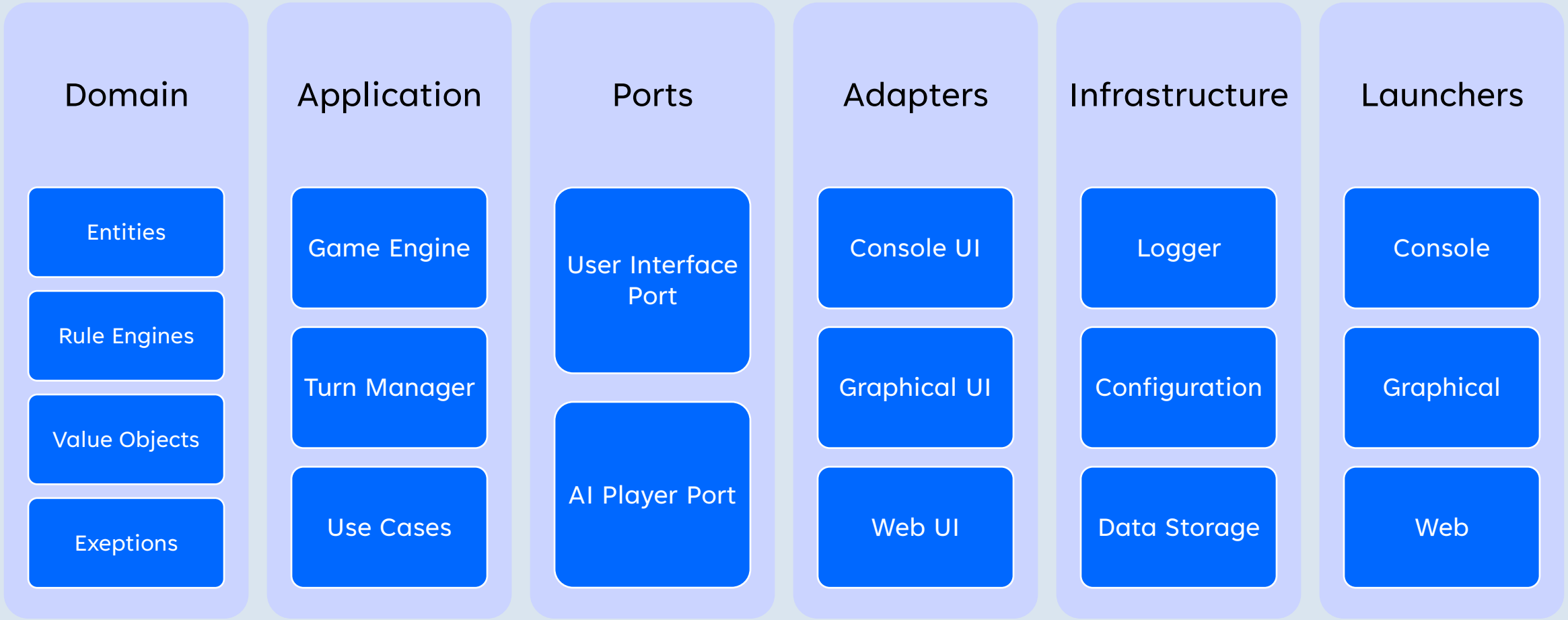
Functional Requirements:

- System must allow two players to play Reversi with valid move enforcement.
- Game must calculate valid moves, flips, and detect end-of-game state.
- Provide visual interaction (Console first, GUI later using Pygame).
- Allow enabling Player vs Player initially; later support AI vs Player.
- Game must expose separate interfaces for UI to maintain portability.

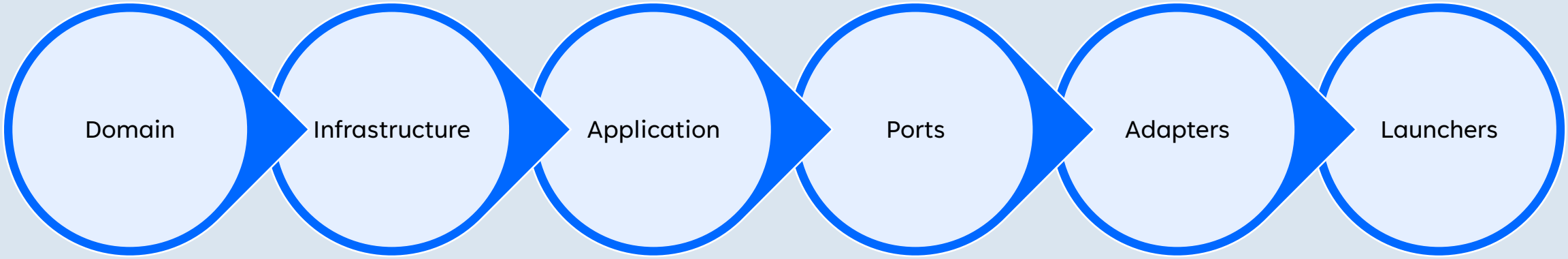
Non-Functional Requirements:

- Clean and extensible architecture.
- Versioning through Git.
- Structured modules for future scalability:
  - AI strategy injection
  - Logging and persistence
  - GUI implementation with minimal disruption
- Code must be testable at domain and application layers.

2. System Design & Architecture



3. Implementation Phase



- Create Entities
- Implement Core Rules

- Add Configuration
- Implement Data Storage Strategies

- Implement Game Engine
- Add use-cases

- Add different ports

- Add different adapters for console/gui/web

- Add different launchers for console/gui/web

4. Testing Strategies

Domain	• Rules & Validation • Flipping & Game-end logic
Application	• Turn Flow & Game Initialization • Error Propagation
Adapters	• Rendering Utilities • User Input Parsing
Infrastructure	• Logging • Configuration
Launchers	• Rendering • Control/Data Flow

5. Maintenance and Scalability

- Add difficulty-based AI engines into the AI Ports:
  - Easy – Random Choice
  - Medium – Greedy Choice
  - Hard – Minimax Algorithm
- Add persistence (save/load feature)
- Add logging-based analytics
- Release web-based UI (adapter for FastAPI) [Optional]
- Integration with leaderboard service

Conclusion

Revisiting the Game Rules

- Reversi is a game of territory control through strategic flips.
- Every legal move must capture at least one opponent’s coin.
- Board awareness—especially edges and corners—directly influences winning potential.
- Simplicity at the rule level opens the door for deeper strategic computation.

Recap of Project Architecture

- We established a modular structure that isolates the core game logic from UI and infrastructure.
- Domain rules remain platform-agnostic, ensuring minimal rework as features evolve.
- Ports and Adapters give the project long-term flexibility:
  - Adding a GUI without modifying logic.
  - Extending AI strategies.
  - Enabling persistence in later phases.
- The architecture reflects real-world enterprise structure where interfaces contractually bind systems while allowing independent evolution.

SDLC Summary

- Requirements were modeled for both initial release and scalable expansion.
- System design aligned with separation of responsibilities, safety of changes, and testability.
- The implementation plan follows staged evolution:
  - First console interaction for correctness,
  - Second GUI for richer UX, and optionally web.
- Testing focuses on correctness of rules before presentation refinements are made.
- Maintenance phase ensures the team can iterate confidently and extend features over time.

Thank you

Talha Ahmad & Areeb