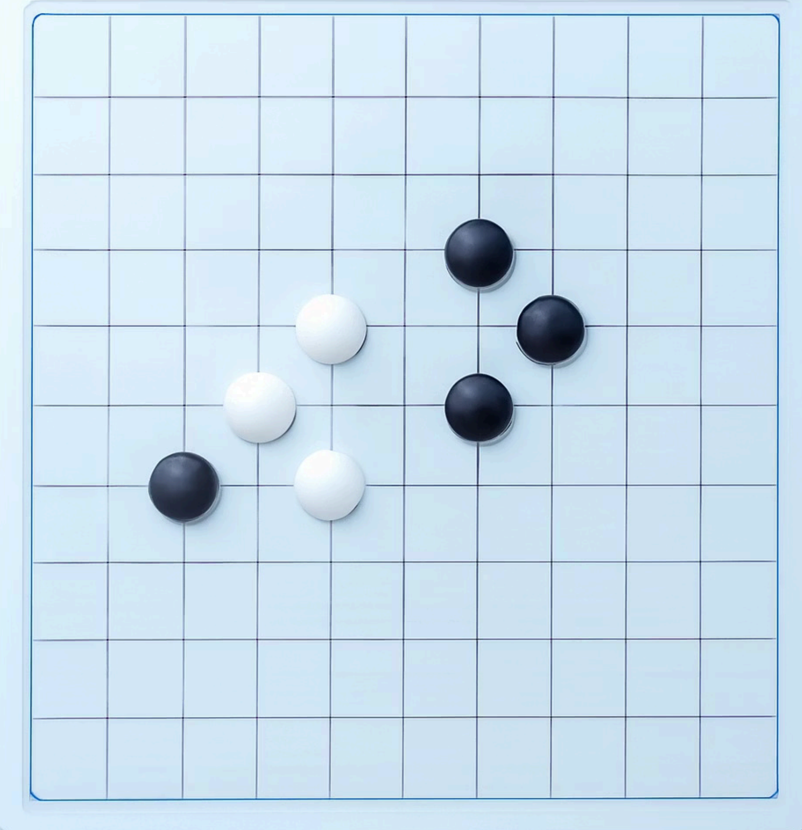
Gomoku

Fantastic 4

Bala Varad Samip Sunny

Made with GAMMA



Introduction & Game Rules

Gomoku (Five in a Row) is a two-player turn-based game

Played on a 15×15 grid

Players take turns placing stones

The first player to form an unbroken line of five stones horizontally, vertically, or diagonally wins

This system uses a networked architecture supporting real-time matches, ranking system, and AI opponents

Made with GAMMA

DEMO TIME

Made with GAMMA

Functional & Non-Functional Requirements

Functional Requirements (FR)

- FR1-FR3: Account Management & Lobby Features
- FR4-FR5: Matchmaking & Game Start
- FR6-FR9: Game Play & Game End
- FR10-FR11: AI Opponent & Admin Functions

Non-Functional Requirements (NFR)

- Stability: Non-blocking I/O, Exception Handling
- Responsiveness: Event-driven Architecture
- Modularity: Layered Design
- Maintainability: Open/Closed Principle
- Cross-platform: Qt 6 + CMake

Made with GAMMA

System Architecture Design

Key Features:

- Layers communicate via signals/slots (Observer pattern)
- The server shares the Domain and Protocol layers
- The server is authoritative and validates all moves

Presentation Layer (Presentation)

- GameWindow
- BoardWidget
- Login/Hub/Game Screens

Domain Layer (Domain)

- Player
- Board
- Move (Pure C++)

Controller Layer (Controller)

- GameClient
- ClientState
- Command

Network Layer (Network)

- NetworkClient
- Protocol

Made with GAMMA

UML

Made with GAMMA

Design Patterns - State Pattern & Strategy Pattern

STATE PATTERN

Problem:

MOVE messages have different meanings in different phases (LobbyState ,MatchingState, InGameState, GameOverState)

Solution:

- Each phase is a **ClientState** implementation
- The controller holds a ClientState* pointer
- Adding a new phase only requires adding one file
- Avoids switch(m_phase) code bloat

STRATEGY PATTERN

Problem:

The matchmaking algorithm may change

Solution:

- Matchmaker holds a **MatchStrategy***
- Default implementation: ClosestLevelStrategy
- Replacing the strategy requires no recompilation of Matchmaker
- Complies with the Open/Closed Principle

Made with GAMMA

Design Patterns - Observer Pattern & Factory Pattern

OBSERVER PATTERN

Problem:

Many parts of the UI need to react to network events, but we cannot have the network layer import or depend on the UI layer

Solution:

- When `BoardWidget` emits `cellClicked(x, y)`, it has no compile-time knowledge of who is listenings
- The network layer doesn't need to #include the UI layer when `GameClient` emits `lobbyUpdated(names)`, the HubScreen observes and re-renders

FACTORY PATTERN

Problem:

The server creates players in two contexts(human & AI). Hard-coding `new` calls throughout the server would couple every handler to every concrete player type.

Solution:

- **PlayerFactory**::create(kind, name, xp) returns a unique_ptr<Player>
- Game session code treats all players the same
- Adding a new player type requires one new subclass and one new branch in the factory

Made with GAMMA

Design Patterns - Singleton Patterns

SINGLETON PATTERN

Problem:

There must be exactly one lobby registry per server process, and every connection handler must see the same instance.

Solution:

- Ensures a globally unique lobby manager (`LobbyManager::instance()` uses a function-local static)
- Implemented using function-local static variables
- Thread-safe and the destructor runs automatically at process exit up

Made with GAMMA

Design Principles (Part 1)

Single Responsibility Principle (SRP)

- Board changes only when game rules change
- NetworkClient changes only when transport method changes
- ClosestLevelStrategy changes only if the matchmaking algorithm changes

Loose Coupling

- The contracts between layers are minimal
- UI ↔️ Controller: Only via Qt signals
- Controller ↔️ Network: NetworkClient::send and messageReceived
- No cross-layer implementation detail dependencies

Encapsulate What Varies

- Matchmaking algorithm → MatchStrategy
- Player types → PlayerFactory
- Phase behavior → ClientState
- User commands → Command

Made with GAMMA

Design Principles (Part 2)

Delegation Over Inheritance

- Inheritance only for is-a relationships: `LobbyState` is a `ClientState`, `AIPlayer` is a `Player`
- GameClient holds ClientState* (not inheritance)、Matchmaker holds MatchStrategy* (not inheritance)
- Swaps the pointer to change behavior and not to create subclass

Law of Demeter

- The renderer calls `m_board->cellAt(x, y)` — not `m_board->grid[x][y]`
- The difference matters: if we rename or restructure the internal grid array, the renderer is unaffected.
- The UI calls `m_client->executeCommand(...)` — not `socket->write(...)`
- The UI has no idea a socket even exists. It talks to the controller, and the controller handles everything below it.

Made with GAMMA