Gomoku

1. Introduction

Gomoku ("five in a row") is a turn-based two-player game played on a 15×15 grid. A move is legal whenever the chosen cell is empty; a player wins by placing five of their stones in an unbroken line — horizontal, vertical, or diagonal. The rules are tiny, but the networked, lobby-based version of the game has enough moving parts (real-time message exchange, level-based matchmaking, three categories of player) to make it a useful canvas for studying object-oriented design.

2. Requirements

2.1 Functional

ID	Requirement
FR1	A player can create a persistent account and log in with a password.
FR2	The server maintains a registry of currently connected, authenticated players (the lobby).
FR3	Authenticated clients can view the current lobby list and leaderboard.
FR4	A player can request a match. The server matches them with the *closest-level* available player.
FR5	When two players are matched, both clients transition into a game and receive match/game notifications.
FR6	Matched players play 15×15 Gomoku, alternating turns; only the player whose turn it is may move.
FR7	The server detects 5-in-a-row (horizontal, vertical, both diagonals) and ends the game with a winner.
FR8	All accepted moves are mirrored to both clients in real time over the network.
FR9	When a game ends, both clients return to the lobby; the winner gains XP (+10), stats, and history persist.
FR10	A client can request "Play vs AI"; the server hosts the AI in the same `GameSession` flow as PvP.
FR11	An admin can connect on a separate port to inspect players/matches, subscribe to events, kick, and force matches.

2.2 Non-functional

- **Stability** — the system must not crash on disconnect, malformed input, or unexpected message ordering. No blocking I/O calls; no UI updates from non-UI threads.

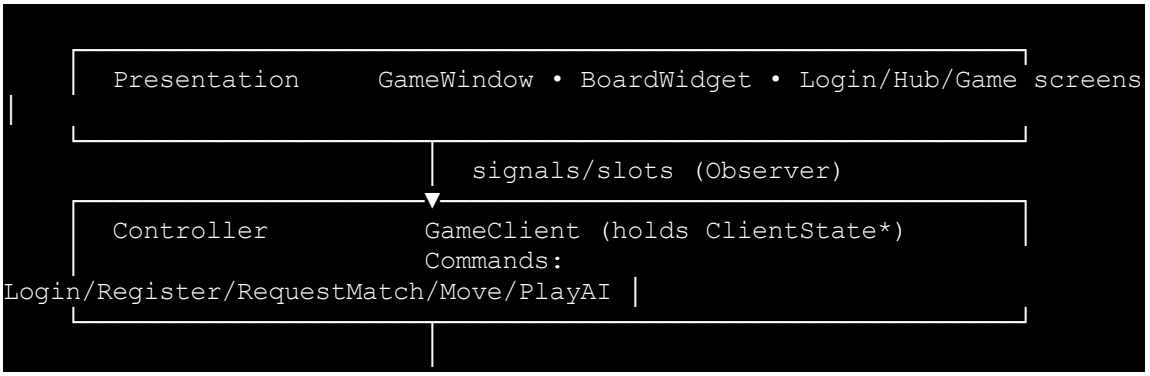
- **Responsiveness** — every network operation is event-driven via Qt signals. The UI thread is never blocked on `waitForReadyRead()`, `waitForConnected()`, or any other blocking primitive.
- **Modularity** — UI, controller, game logic, and networking live in separate classes/layers. A class never reaches across two layers.
- **Maintainability** — adding a new matchmaking rule, a new client-side phase, or a new player kind must not require modifying any existing class (Open/Closed via Strategy, State, Factory).
- **Demo-readiness** — startup in under ten seconds; the demo path is one server + two clients (or one client in AI mode).
- **Cross-platform** — Qt 6 + CMake; the same code builds on macOS, Linux, and Windows.

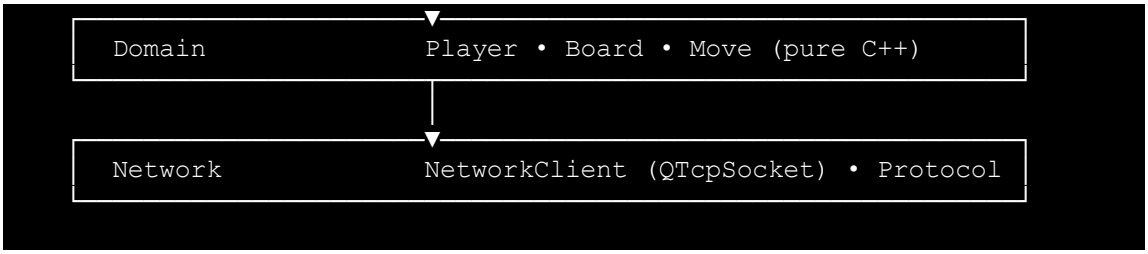
2.3 Use Cases

- **UC1 Login / Create Account.** The client connects to the server. The user logs in or creates an account. The lobby list updates with their entry and any other online players.
- **UC2 Request Match.** A lobbied user clicks "Find Match". The server pairs them with the player whose level is closest to theirs.
- **UC3 Play Game.** Matched players take turns clicking cells. Each click sends `MOVE:x,y`; the server validates and broadcasts the mirror.
- **UC4 Win/Loss.** A 5-in-a-row triggers `GAME_OVER:winner`; XP updates; both clients return to the lobby.
- **UC5 Disconnect Mid-Game.** If one player's socket closes, the other is awarded a forfeit win and returned to the lobby.
- **UC6 Play vs AI.** A user clicks "Play vs AI"; the server creates a `GameSession` with a server-hosted `AIPlayer`.
- **UC7 Admin Monitoring.** An admin connects to the admin port and can list players/matches, subscribe to events, inspect leaderboard history, kick players, and force matches.

3. Design Overview

3.1 Layered Architecture





The server has its own stack that *shares* the Domain and Protocol layers verbatim:

```
LobbyServer (QTcpServer)
    ├── ClientHandler (one per connection)
    ├── LobbyManager  (singleton: PlayerRegistry)
    ├── Matchmaker    (Strategy)
    ├── GameSession   (one per active match)
    ├── UserStore + GameStore (SQLite persistence)
    └── AdminServer + AdminHandler (separate admin port)
```

The same `Board` class is shared, but the server is authoritative: server-side sessions validate moves and detect wins, while the client keeps a render mirror. The same `Protocol::encode/parse` functions speak the player wire on both sides. Sharing the domain and protocol layer prevents the protocol drift that bites real client-server projects.

3.2 Class Catalog

Class	Responsibility
`Player`	Hold name/XP/wins/losses and expose the `level = xp/100` formula.
`Board`	15×15 grid: `placeMove`, `checkWin`, `cellAt`. Pure C++; unit-testable.
`Move`	Plain value type carrying `(x, y, playerId)` across layers.
`Protocol`	Encode/decode every wire message.
`NetworkClient`	Wrap a `QTcpSocket`; emit framed `messageReceived(QString)`.
`GameClient`	Client-side controller; current state + dispatcher.
`ClientState` (+ four)	Phase-specific behaviour: Lobby/Matching/InGame/GameOver.
`Command` (+ five)	One user intent encapsulates "send this on the wire".
`PlayerFactory`	Build a `Player` of the right concrete kind (Local/Remote/AI).
`AIPlayer`	Compute the AI's next move from a heuristic.

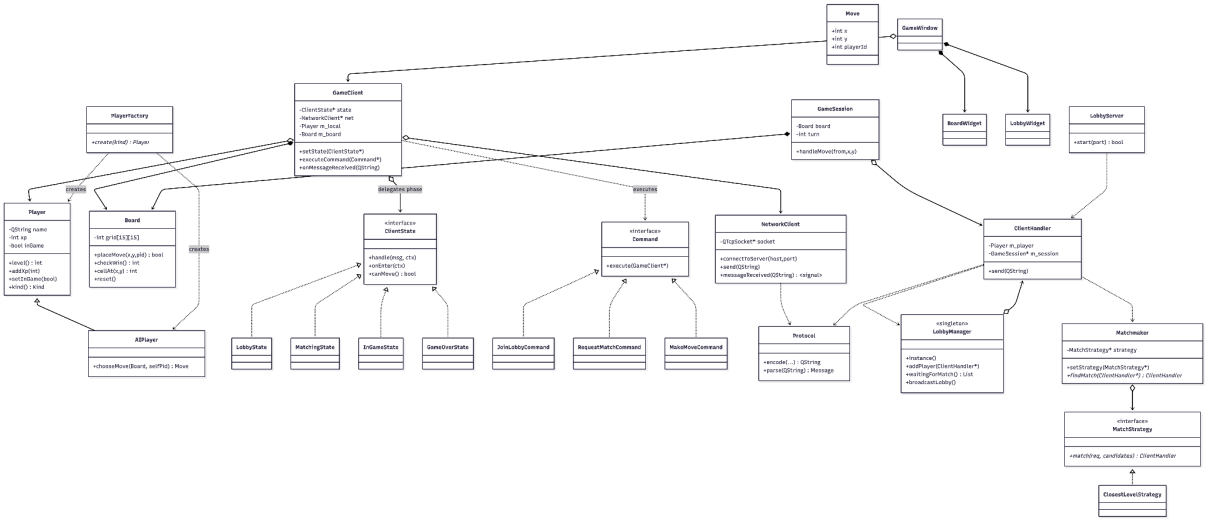
Class	Responsibility
`GameWindow`	Top-level Qt widget; wires UI signals to GameClient commands.
`BoardWidget`	Render the board read-only; emit `cellClicked(x, y)`.
`LoginScreen`	Render login/create-account controls.
`HubScreen`	Render profile, online players, leaderboard, match/AI/history controls.
`GameScreen`	Render the in-game board and match status.
`LobbyServer`	Accept TCP connections; spawn `ClientHandler` per socket.
`ClientHandler`	Per-connection: parse messages, dispatch to lobby/matchmaker/session.
`LobbyManager`	Singleton registry of online players + lobby broadcast.
`Matchmaker`	Find a partner using the current `MatchStrategy`.
`MatchStrategy` + impl	One algorithm for choosing a partner (closest-level by default).
`GameSession`	Server-authoritative match: validate moves, detect win/forfeit, broadcast.
`UserStore`	Persist accounts, password hashes, XP, wins, losses, and leaderboard rows.
`GameStore`	Persist game IDs and completed match history.
`AdminServer`/`AdminHandler`	Separate admin protocol for monitoring and control.

Every entry has *one* reason to change. The matchmaking algorithm is isolated from the matchmaker; the wire format is isolated from the socket; the rules are isolated from the renderer.

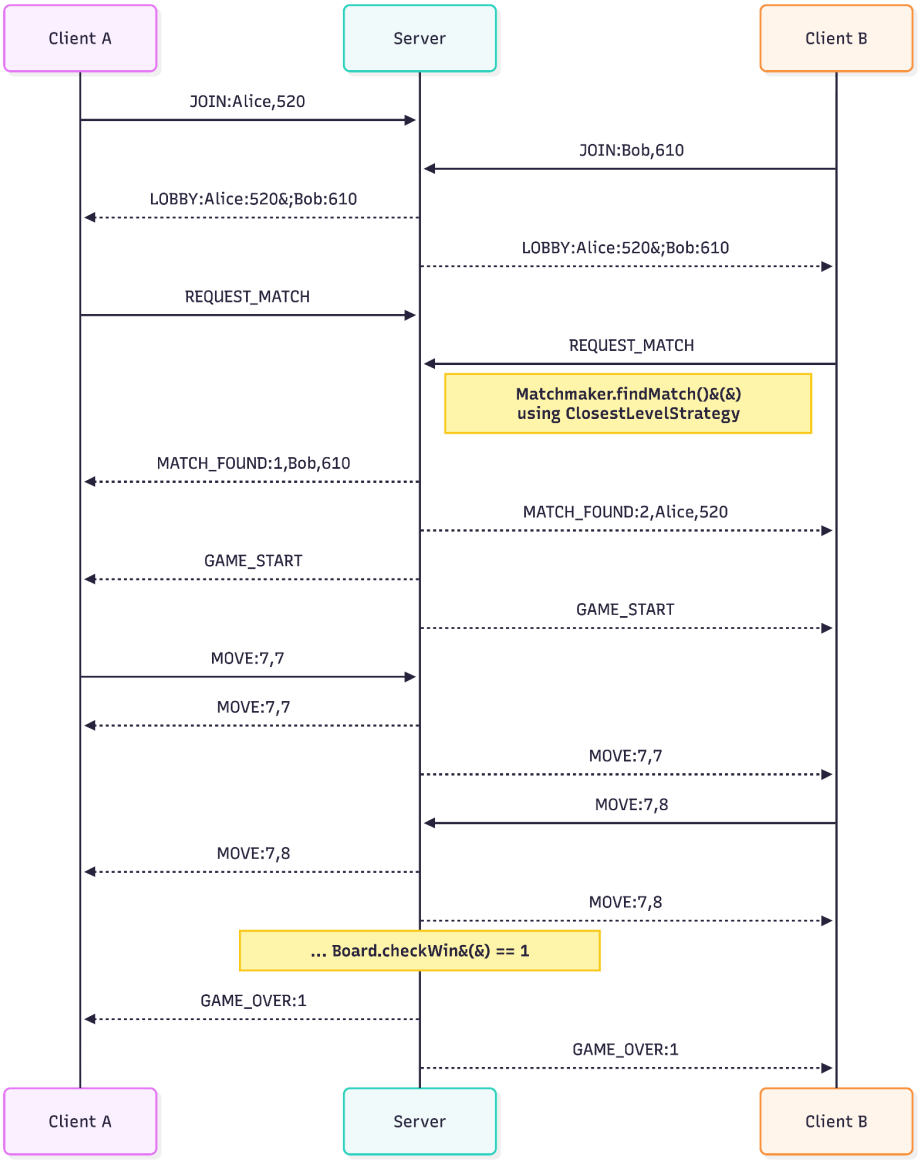
4. UML Diagrams

The class and sequence diagrams are provided as separate attachments:

- `docs/UML.png` — high-level class diagram.

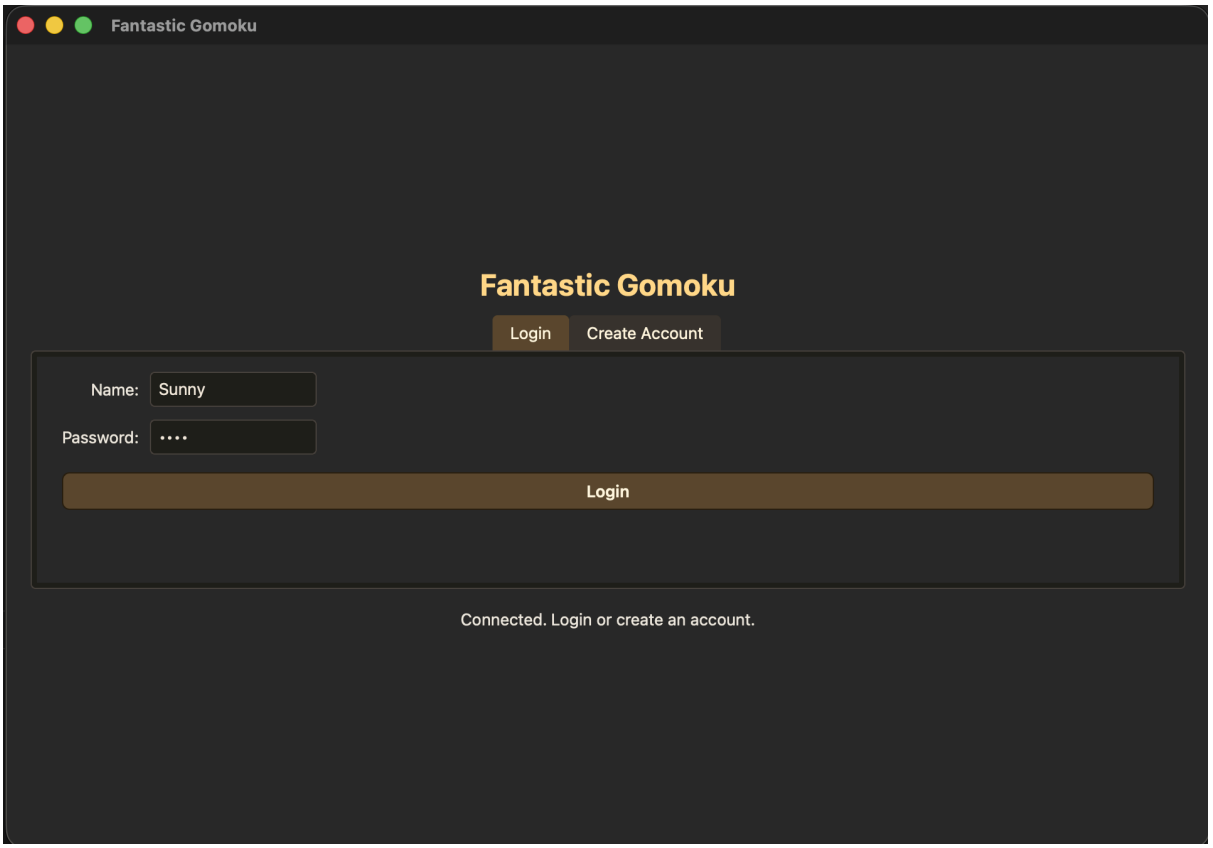


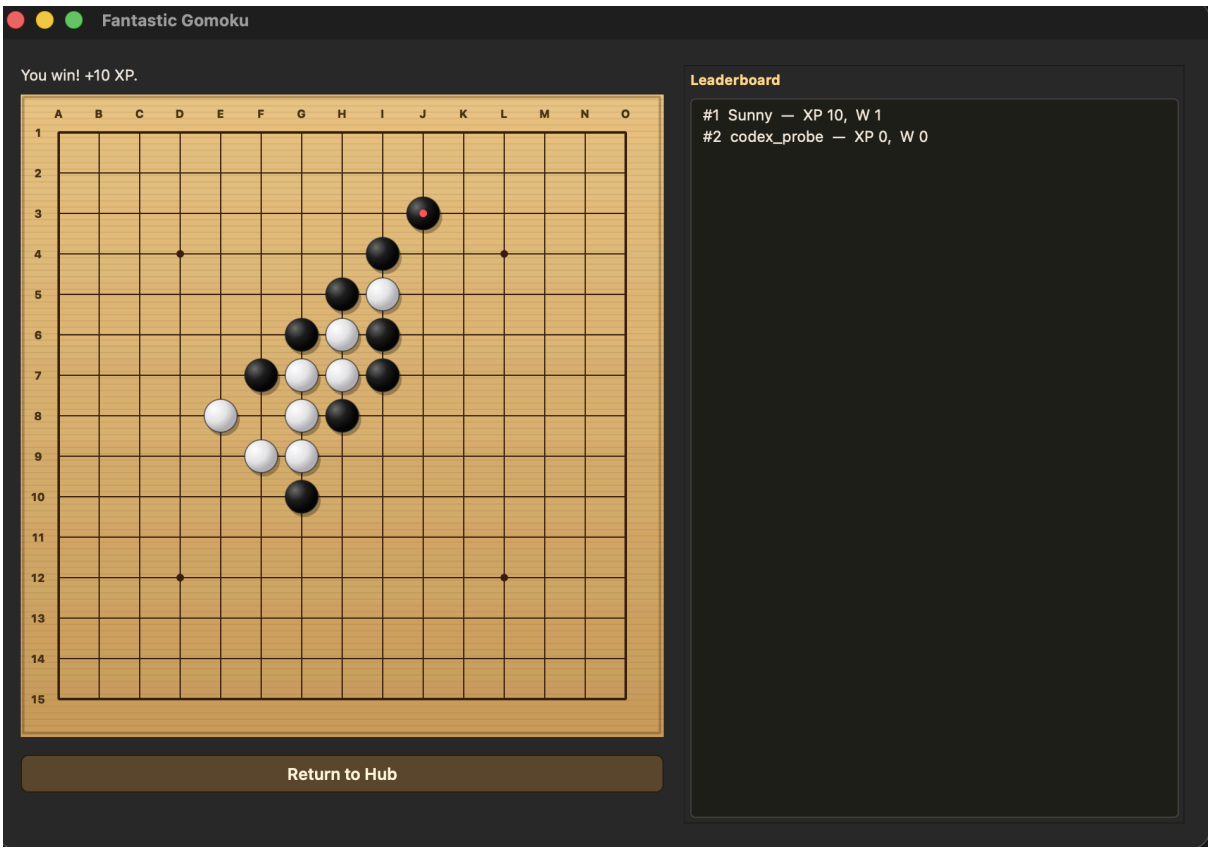
- `docs/Sequence Diagram.png` — match/demo flow sequence diagram.

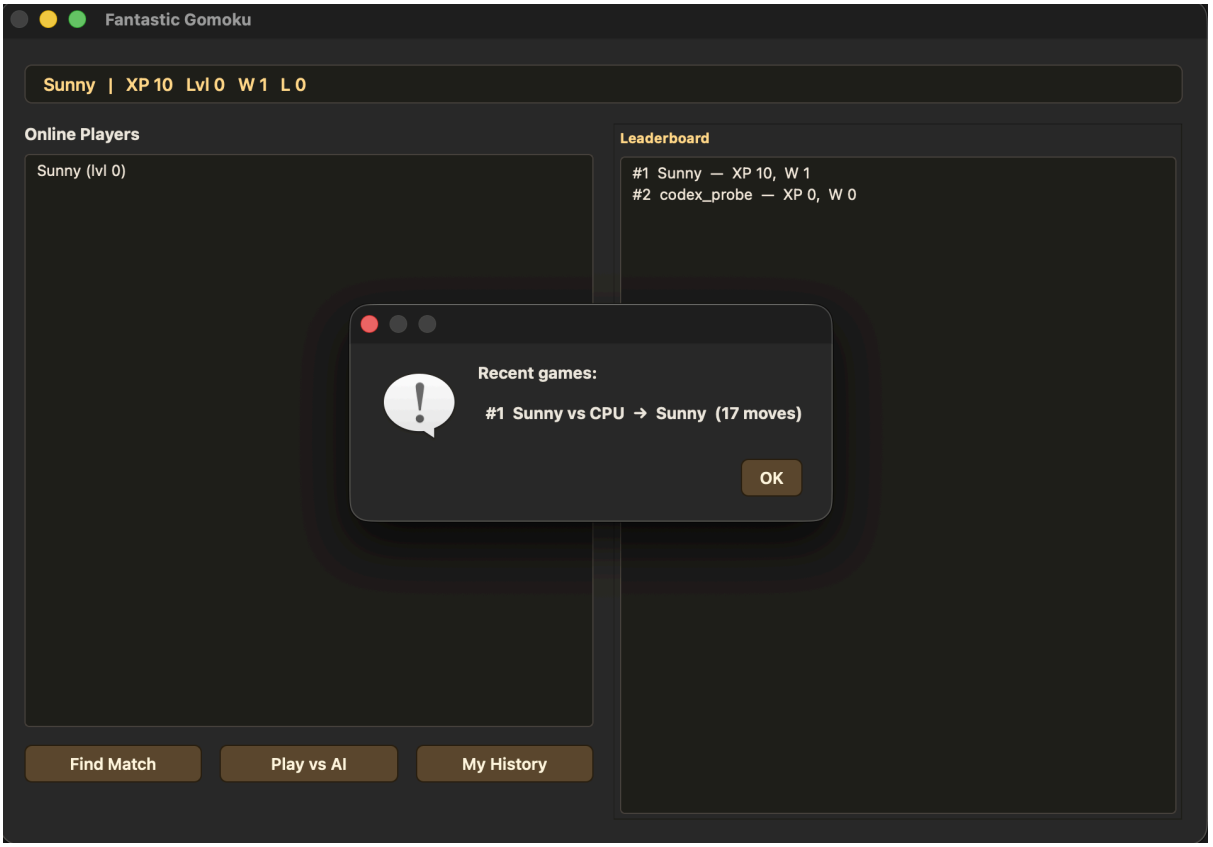


4.1 Supplementary Screenshots

The following screenshots show the application running through the main demo path: login/account creation, gameplay, and leaderboard updates.







5. Design Principles Applied

5.1 Single Responsibility (high cohesion)

Every class above has *one* reason to change. The cleanest examples:

- `Board` changes only if Gomoku's rules change. It does not know about sockets, UI, or players' XP.
- `NetworkClient` changes only if the framing or transport changes. It does not know what the messages mean.
- `ClosestLevelStrategy` changes only if the matchmaking heuristic changes. It does not know how candidates were collected or how the result is reported.

The contrast is the naïve design's `GomokuApp` (§7): one class with five reasons to change.

5.2 Loose Coupling

The contracts between layers are deliberately small:

- UI ↔ controller: Qt signals only (`cellClicked(x,y)`, `boardChanged()`).
- Controller ↔ network: `NetworkClient::send` and the `messageReceived` signal.
- Server ↔ matchmaking: a single virtual call — `MatchStrategy::match`.

No class transitively imports the *implementation* details of two layers above or below it. That is what lets us, for example, replace the entire `BoardWidget` (perhaps with a QML version later) without recompiling the controller.

5.3 Encapsulate What Varies

We identified four things in this domain that are likely to change:

1. **The matchmaking algorithm** → behind `MatchStrategy`.
2. **The set of player kinds** → behind `PlayerFactory`.
3. **Phase-specific behaviour** → behind `ClientState`.
4. **One-shot user actions** → behind `Command`.

Each variable thing is wrapped in an interface. Adding a new matchmaking rule is one new class; nothing else changes.

5.4 Delegation Over Inheritance

Inheritance is used only for *is-a* relationships:

`LobbyState` *is a* `ClientState` (it describes one phase).

- `ClosestLevelStrategy` *is a* `MatchStrategy`.
- `AIPlayer` *is a* `Player`.

For everything else, we delegate:

`GameClient` *has a* `ClientState*` (the active phase) — the client doesn't subclass to change behaviour, it swaps the state pointer.

- `Matchmaker` *has a* `MatchStrategy*` — likewise.

This keeps inheritance hierarchies shallow (one level) and avoids the fragile-base-class problem.

5.5 Law of Demeter

The renderer does not reach into the model:

```
// BoardWidget::paintEvent
int v = m_board->cellAt(x, y);   // not m_board->grid[x][y]
```

The UI does not reach into the network:

```
// onCellClicked - no socket->write here
m_client->executeCommand(std::make_unique<MakeMoveCommand>(x, y));
```

The state classes do not poke at `GameClient`'s internals beyond a stable, public surface — `ctx->board()`, `ctx->setSelfPid()`, `ctx->emit lobbyUpdated()`. Each call talks to one peer; no chain of `ctx->getX()->getY()->doZ()` exists.

5.6 Polymorphism Over Type Switching

For the main extension points, the code avoids phase/type switches and uses polymorphic dispatch:

An incoming network message is dispatched by `m_state->handle(msg, this)`. Each state implements `handle` differently. There is no `switch (m_phase)`.

- A user action is dispatched by `cmd->execute(this)`. Each command implements `execution` differently.

- A matchmaking pick is `m_strategy->match(req, candidates)`. Each strategy implements a `match` differently.

This is the payoff for the State/Command/Strategy patterns: a code shape that grows by adding files, not by editing existing ones.

6. Design Patterns Used

We required six patterns. Each is in the codebase because it solves a specific problem; none is decorative.

6.1 State Pattern — `ClientState`

Problem. A `MOVE` message means very different things depending on what the client is doing. In `LobbyState` it should be ignored; in `MatchingState` it indicates a protocol violation; in `InGameState` it is an opponent's stone; in `GameOverState` it is stale. A `switch (m_phase)` block would grow with every new phase.

Solution. Each phase is a class implementing `ClientState`. The controller holds a `ClientState*` and forwards all incoming messages to it. Adding a `ReconnectingState` later means adding one file.

Code:

```
void GameClient::deliver(const Protocol::ParsedMessage& msg) {
    if (m_state) m_state->handle(msg, this);
}
```

6.2 Strategy Pattern — `MatchStrategy`

Problem. The spec calls for closest-level matching today. Tomorrow, we might want a random pairing for testing, or a rating-based pairing once ELO is added. We don't want `Matchmaker` to recompile when that happens.

Solution. `Matchmaker` owns a `MatchStrategy*`. The default implementation is `ClosestLevelStrategy`; replacing it is a single `Matchmaker::setStrategy(...)` call. `Matchmaker` itself is closed against new algorithms.

Code:

```
    ClientHandler* Matchmaker::findMatch(ClientHandler* requester) {
        auto candidates = LobbyManager::instance().waitingForMatch();
        candidates.removeAll(requester);
        return m_strategy->match(requester, candidates);
    }
```

6.3 Observer Pattern — Qt Signals/Slots

Problem. Many things (the lobby list, the board, the connection status) need to react to events without those events being aware of their listeners.

Solution. Qt's signal/slot system is a typed implementation of the Observer pattern. We use it pervasively:

- `BoardWidget::cellClicked(x, y)` — the widget *announces*; whoever is connected reacts.
- `GameClient::lobbyUpdated(names)` — the controller *announces*; the hub screen observes.

Because the slot side of a connection is just `QObject::connect(...)`, the emitter has no compile-time knowledge of who listens. This is what makes it possible for a network message to update the UI without any `#include` from the network layer to the UI layer.

6.4 Command Pattern — `LoginCommand`, `RegisterCommand`, `RequestMatchCommand`, `MakeMoveCommand`, `PlayAICommand`

Problem. A button click should not call `socket->write(...)` directly — that is two layers of leakage. We also want a single choke-point where the controller can reject illegal actions ("you can't move; you're in the lobby").

Solution. Each user action is a `Command` object. The widget constructs one and hands it to `GameClient::executeCommand()`:

```
    void GameWindow::onCellClicked(int x, int y) {
        m_client->executeCommand(std::make_unique<MakeMoveCommand>(x,
y));
    }
```

`MakeMoveCommand::execute` calls `ctx->submitMove(x, y)`, which consults the current state's `canMove()` before doing anything. The widget layer is now ignorant of both the wire format and the state machine.

6.5 Factory Pattern — `PlayerFactory`

Problem. The server creates different player kinds in different contexts: authenticated humans are plain `Player` values restored from SQLite, while "Play vs AI" needs an `AIPlayer`. Hard-coding `new` calls in handlers and sessions would couple those call sites to every concrete type.

Solution. `PlayerFactory::create(kind, name, xp)` returns a `unique_ptr<Player>`. Adding a fourth kind (say a `ReplayPlayer` for replays) means adding one branch in the factory and one subclass — no other file changes.

This is also where Liskov substitution earns its keep: game-session code can treat the AI as a `Player`-compatible participant where identity/progression are concerned, while AI-specific move selection stays inside `AIPlayer`.

6.6 Singleton Pattern — `LobbyManager`

Problem. There must be exactly one lobby per server process, and every connection handler must see the same registry. Passing a shared pointer to every handler would work, but it adds plumbing for no real benefit; the lifetime is naturally one-per-process.

Solution. `LobbyManager::instance()` returns a reference to a function-local `static`:

```
LobbyManager& LobbyManager::instance() {
    static LobbyManager s;
    return s;
}
```

This idiom is thread-safe in C++11+, and the destructor runs at process exit. We deliberately don't allow heap-based singleton creation/destruction — it removes the most common Singleton bug class (use-after-free during shutdown).

Singleton is the riskiest of the six patterns, so it is limited to process-wide services whose lifetime naturally matches the server: `LobbyManager`, `Matchmaker`, `MatchRegistry`, `UserStore`, `GameStore`, and `AdminEventBus`.

7. Improvement

Naïve construct	Replaced with	Pattern/principle gained
One mega-class	Four-layer split (UI / Controller / Domain / Net)	SRP, separation of concerns
Direct `repaint()` from socket handler	`boardChanged` signal → UI slot	Observer, loose coupling
Hard-coded matchmaking	`Matchmaker` + `MatchStrategy`	Strategy
`switch (m_phase)`	`ClientState* m_state`	State
`socket->write(...)` from buttons	`Command::execute(ctx)`	Command
`new GomokuApp` constructs all player types	`PlayerFactory::create(kind, ...)`	Factory
Globally-readable `int board[15][15]`	`Board::cellAt(x, y)`	Encapsulation, Demeter
Many global lookups for the lobby list	`LobbyManager::instance()` + clear lifetime	Singleton (used judiciously, with rationale)

The result is more files but fewer reasons for any single file to change. A future contributor adding a feature touches one or two new files; in the naïve design, they would have edited `GomokuApp` and broken something unrelated.

8. Build & Run

```
# From the project root, with Qt 6 installed:
cmake -S . -B build-demo -DBUILD_TESTING=OFF \
      -DCMAKE_PREFIX_PATH=$HOME/Qt/6.10.2/macos
cmake --build build-demo --target gomoku-demo

# Demo path 1: networked
./build-demo/gomoku-server -p 5555 --slots 20 --admin-port 5600 &
./build-demo/gomoku-client      # window 1: create/login Alice
./build-demo/gomoku-client      # window 2: create/login Bob
# Both click "Find Match" → matched on closest level.

# Demo path 2: server-hosted AI
./build-demo/gomoku-client      # login, then click "Play vs AI."
```