ROYAL CHESS

A Feature-Rich Desktop Chess Application with AI, Player Profiles and Multiplayer

CMPE 202 — Software Systems Engineering · Spring 2026
San Jose State University · Engineering Extended Studies

LanguageC++17	**FrameworkQt 6.10**	**Source1,504 lines**
9 Design Patterns	**7 Design Principles**	**TCP Multiplayer**

https://github.com/anuradha1105/RoyalChessCMPE202/

Team Members: Anuradha Srivastav
Jay Hitesh Vithlani
Yash Rajeshbhai Kalathiya

1. What Is the Application?

Royal Chess is a fully-featured desktop chess application built in C++17 with the Qt6 framework. It implements a complete chess engine enforcing all standard rules, an AI opponent powered by Minimax with Alpha-Beta pruning, a persistent player profile system with match history and an ELO-inspired rating, dual chess clocks, and peer-to-peer TCP network multiplayer. The application is organized around nine design patterns and seven software design principles, demonstrating how good architecture keeps complex features manageable and independently changeable.

1.1 Core Features

Feature	Description
Chess Engine	Complete rule set: legal move generation for all six piece types, castling (kingside and queenside with full path-safety checking), en passant, pawn promotion, check, checkmate, stalemate, and draw by insufficient material.
AI Opponent	Minimax algorithm with Alpha-Beta pruning at three selectable depth levels — Squire (depth 1, ~30 nodes), Knight (depth 3, ~300 nodes), King (depth 4, ~900 nodes after pruning).
Player System	Player registration and persistent login. Points-based rating system: start at 1,000 ELO, +10 for a win, -10 for a loss. Match history and a ranked leaderboard across all registered players.
Chess Clocks	Dual countdown timers — one per player — with red warning colour below 30 seconds, plus a per-move timer that resets after each move.
Undo	Reverses the last player move and the AI response together, restoring the board to the position before the player's turn using the Command pattern.
TCP Multiplayer	Peer-to-peer network play on the same Wi-Fi network. One player hosts (QTcpServer on port 45678), the other joins by IP address (QTcpSocket). No server required.
Medieval UI	Kingdom-themed interface with dark wooden board, gold typography, and independent visual layers: last-move highlight, selection highlight, legal-move dots, and check overlay.

1.2 How to Run the Application

Prerequisites

- macOS 12 or later
- Qt 6.10.2 — free open-source edition from qt.io/download
- CMake 3.16 or later
- Apple Clang: run xcode-select --install in Terminal

Build from Qt Creator (recommended)

1. Open Qt Creator. Choose File > Open File or Project and select CMakeLists.txt from the chess2 folder.
2. On the Configure Project screen tick Qt 6.10.2 for macOS and click Configure Project.
3. Go to Projects > Qt 6.10.2 for macOS > Run Settings > Environment and add the variable:
   `QTFRAMEWORK_BYPASS_LICENSE_CHECK = 1`
4. Press Cmd+R. Qt Creator compiles and launches Royal Chess.

Build from Terminal

```
cd ~/Documents/CMPE202/chess2
mkdir build && cd build
/usr/local/Cellar/cmake/4.3.1/bin/cmake .. -DCMAKE_PREFIX_PATH=~/Qt/6.10.2/macos
make -j4
QTFRAMEWORK_BYPASS_LICENSE_CHECK=1 ./RoyalChess
```

Playing the game

- Register: type your name on the first screen and click ENTER THE KINGDOM. Returning players are listed — one click to log in.
- Play vs AI: select White or Black, choose difficulty, click ENTER BATTLE.
- Moving: click a piece — legal destination squares appear as dots. Click a dot to move.
- Multiplayer: Player 1 clicks HOST GAME (app shows IP). Player 2 clicks JOIN GAME and enters that IP. Both must be on the same Wi-Fi network.

2. Software Architecture

Royal Chess is organized as a multi-file C++ project with a clean Model-View-Controller (MVC) architecture. The source is split across separate header (.h) and implementation (.cpp) files for each major class, plus Types.h which defines the shared data structures used throughout the application.

2.1 Project Structure

File	MVC Role	Responsibility
include/Board.h + src/Board.cpp	**Model**	Chess game state, all rules, move generation, make/undo
include/BoardWidget.h + src/BoardWidget.cpp	**View**	Renders the board, handles mouse input, visual layers
include/MainWindow.h + src/MainWindow.cpp	**Controller**	Manages screens, game lifecycle, clocks, saves results
include/AI.h + src/AI.cpp	**AI Engine**	alphaBeta() minimax search + bestMove() entry point
include/DataMgr.h + src/DataMgr.cpp	**Data Layer**	Singleton + Facade for JSON persistence and ELO

include/NetworkManager.h + src/NetworkManager.cpp	**Network Layer**	Singleton for TCP peer-to-peer multiplayer
include/Types.h	Shared	Color, PType, Pos, Square, Move structs shared by all classes

2.2 MVC Architectural Pattern

The application is structured around the Model-View-Controller pattern. Each layer has a single, clearly bounded responsibility and the three layers communicate through well-defined interfaces.

Board (Model) contains all chess logic and owns the game state — the 8×8 grid, the current turn, en passant target, and move history. It exposes legalMoves(), make(), and undo() as its public interface. Critically, Board.h includes no Qt widget headers: it compiles and runs with no knowledge that a user interface exists.

BoardWidget (View) reads the Board to render the chess board on screen and handles mouse click events. It never calls DataMgr or NetworkManager. It communicates with MainWindow exclusively through two std::function callbacks: onEvent (game state changes such as checkmate or check) and onNetworkMove (to relay a local move to the TCP layer).

MainWindow (Controller) manages a QStackedWidget with three pages: Registration (index 0), Home with Play/Profile/Ranks/History tabs (index 1), and Game (index 2). It creates all components, manages the dual QTimer chess clocks, subscribes to onEvent, and calls DataMgr::inst().saveMatch() when a game ends. It implements no chess logic and performs no drawing.

MVC is an architectural pattern rather than a GoF structural pattern. It is listed here as the overarching design decision that enables all other patterns to work without cross-layer dependencies.

3. Key Classes and Associations

Board — chess engine (Board.h)

```
class Board {
public:
   std::array<std::array<Square,8>,8> g;  // 8×8 grid
   Color turn = Color::White;
   Pos   ep;                          // en passant target
   std::vector<Move> history;
   // Template Method Pattern skeleton
   std::vector<Move> pseudoMoves(Color col) const;
```

```
    std::vector<Move> legalMoves (Color col);
    // Command Pattern execution / reversal
    void make(const Move& m);
    void undo(const Move& m);
    int  evaluate() const;
};
```

Board has no Qt includes and no knowledge of BoardWidget, MainWindow, DataMgr, or NetworkManager. This is the strongest possible expression of the Single Responsibility Principle: Board exists solely to manage chess state.

BoardWidget — view and observer (BoardWidget.h)

```
class BoardWidget : public QWidget {
    Q_OBJECT
public:
    Board board;
    Color playerColor = Color::White;
    int   aiDepth     = 3;
    // Observer callbacks — wired by MainWindow at game start
    std::function<void(const QString&)>  onEvent;
    std::function<void(int,int,int,int)> onNetworkMove;
protected:
    void paintEvent    (QPaintEvent*) override;
    void mousePressEvent(QMouseEvent* e) override;
};
```

Key associations between classes

From	Relationship	Details
BoardWidget	OWNS	Board — BoardWidget contains a Board by value; the game engine is embedded in the view.
MainWindow	USES	BoardWidget — creates, owns, and wires onEvent callback.
MainWindow	DEPENDS ON	DataMgr::inst() — calls loginPlayer(), saveMatch(), loadAll().
MainWindow	DEPENDS ON	NetworkManager::inst() — calls startHost(), connectToHost(), sendMove().
Board	USES	Move — legalMoves() returns vector<Move>; make() and undo() consume Move objects.

3.1 Loose Coupling in Detail

The most important coupling decision in the application is how BoardWidget communicates with MainWindow. Instead of calling MainWindow methods directly, BoardWidget exposes two std::function callbacks:

```
// BoardWidget.h — the only interface between View and Controller
std::function<void(const QString&)>  onEvent;
std::function<void(int,int,int,int)> onNetworkMove;
```

```
// BoardWidget.cpp — fires the event; has no knowledge of who listens
if(onEvent) onEvent("checkmate:" + winner);
if(onEvent) onEvent("stalemate");
if(onEvent) onEvent("check");
```

```
// MainWindow.cpp — subscribes to the callback at game start
board->onEvent = [this](const QString& ev){
   if(ev.startsWith("checkmate")){
      DataMgr::inst().saveMatch(result, opp, oppElo, moves, col, tc);
      refreshHomeData();
   }
   if(ev == "check") statusLbl->setText("Check!");
};
```

BoardWidget has zero compile-time dependency on MainWindow. The View can be compiled, tested, and used in a completely different application without any modification. This is a direct application of the Principle of Least Knowledge.

3.2 UML Class Diagram

MainWindow sits at the top as the Controller. It uses BoardWidget to display the game, depends on DataMgr for saving player data, and depends on NetworkManager for multiplayer connections.

BoardWidget is the View. It owns a Board object directly — the chess engine is embedded inside the view component. BoardWidget exposes two callbacks: onEvent (fires game events to MainWindow) and onNetworkMove (relays moves to the network layer).

Board is the Model. It holds the 8×8 grid, the current turn, and move history. It provides legalMoves(), make(), undo(), inCheck(), and evaluate() — the complete chess engine interface. Board has no knowledge of any other class.

DataMgr and **NetworkManager** are both Singletons accessed via inst(). DataMgr is also a Facade — it hides JSON file I/O and ELO calculation behind saveMatch(). NetworkManager handles TCP connections for multiplayer.

Move is the Command pattern data carrier. It stores everything needed to both execute and reverse any chess move, including special cases like castling (castle flag), en passant (capPos stores the actual captured pawn position), and promotion.

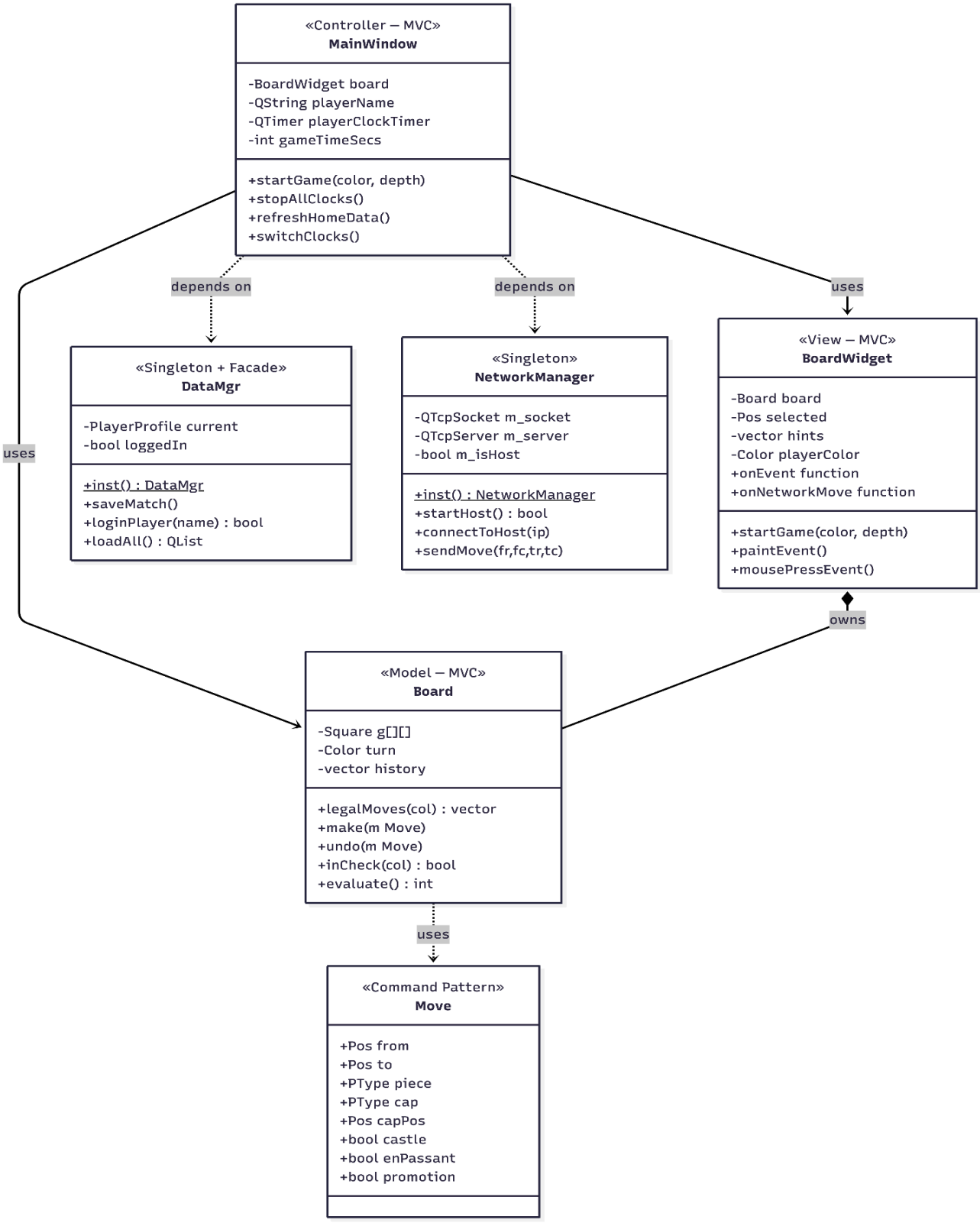


Figure 1 — UML Class Diagram showing all major classes, attributes, methods, and relationships

The class diagram shows the five major classes in Royal Chess and how they relate to each other.

3.3 UML Sequence Diagram — A Player Makes a Move

The sequence diagram shows exactly what happens when a player makes a chess move from start to finish across all five classes.

Section 1 — Player selects and moves a piece. The player clicks a piece; BoardWidget calls legalMoves() on Board and receives the list of valid moves; BoardWidget renders the hint dots. The player clicks a destination; BoardWidget calls make() on Board (Command pattern); the board repaints.

Section 2 — AI calculates and responds. BoardWidget calls bestMove() on Board passing the depth parameter (Strategy pattern); Board runs the alphaBeta() recursive search and returns the best move; BoardWidget calls make() with the AI move and repaints.

Section 3 — Checkmate detected, result saved. BoardWidget detects no legal moves remain and fires onEvent("checkmate") to MainWindow (Observer pattern); MainWindow calls DataMgr.saveMatch() (Facade pattern) which updates the ELO, appends the match record, and writes the JSON file; MainWindow shows the Game Over popup.

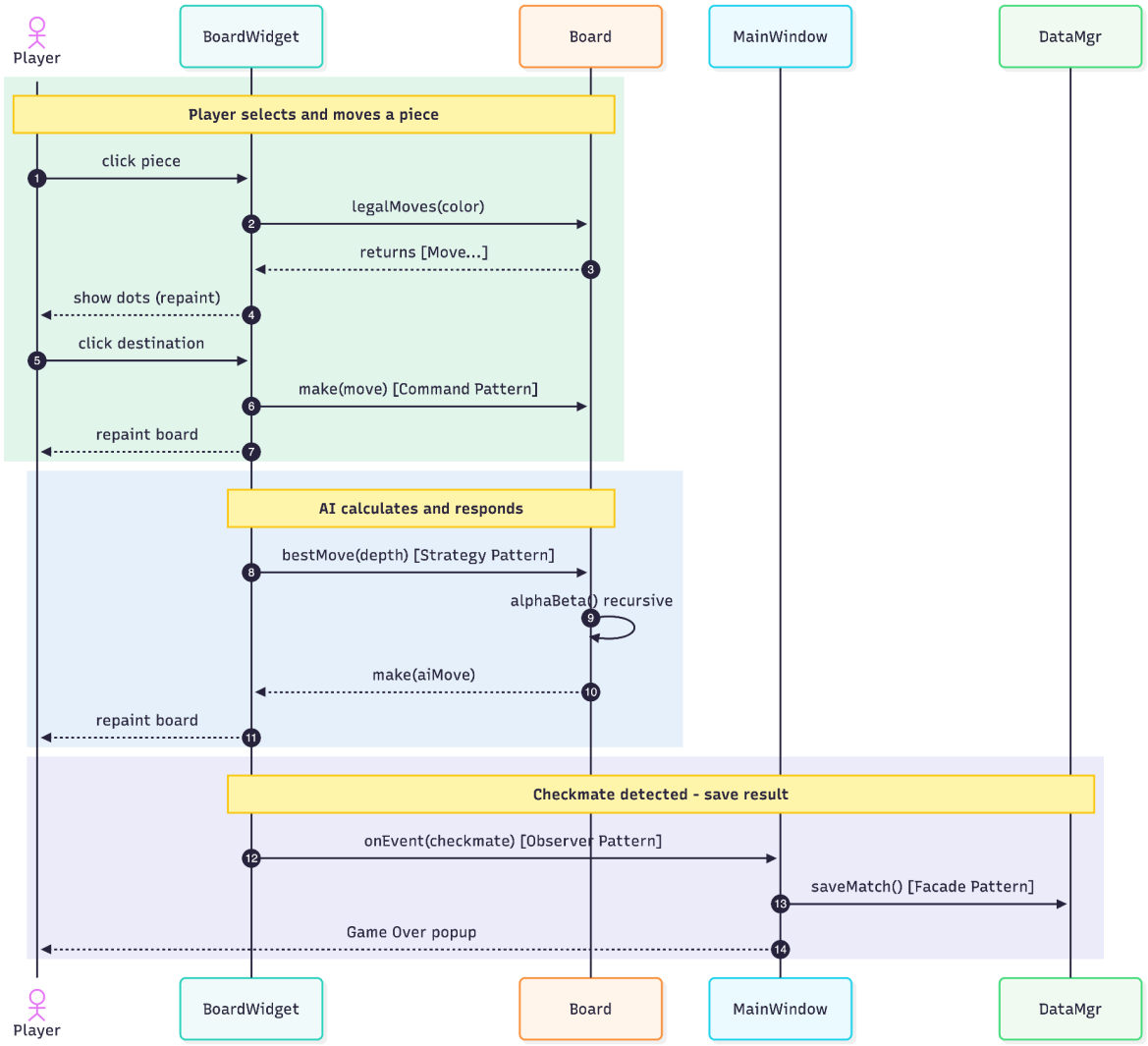


Figure 2 — Sequence Diagram: from mouse click through AI response, checkmate detection, and data save

4. ELO Rating System

Royal Chess implements a simplified ELO-inspired rating system. ELO is named after its inventor Arpad Elo, a Hungarian-American physics professor who developed the rating system in the 1960s. It is not an acronym but his surname.

In the standard ELO formula the rating change depends on the difference between the two players' current ratings. Royal Chess uses a simplified fixed-delta variant that is transparent and easy for players to understand at a glance.

4.1 Rating Formula

```
// DataMgr.h — PlayerProfile::calcElo()
static int calcElo(int myElo, int oppElo, const QString& result){
    Q_UNUSED(myElo) Q_UNUSED(oppElo)
    if(result == "WIN")  return  10;
    if(result == "LOSS") return -10;
    return 0;  // DRAW
}
```

Every player begins at 1,000 points. A win adds 10 points; a loss subtracts 10 points; a draw leaves the rating unchanged. A player's rating can never fall below zero.

4.2 Player Titles

ELO Range	Title Awarded
1,000 – 1,399	Squire (starting level)
1,400 – 1,599	Knight
1,600 – 1,799	Expert
1,800 – 1,999	Master
2,000+	Grandmaster

4.3 Persistence

```
// DataMgr.cpp — saveMatch() — the Facade call
void DataMgr::saveMatch(const QString& result, const QString& opp,
                        int oppElo, int moves,
                        const QString& col, const QString& tc) {
    int ec = PlayerProfile::calcElo(current.elo, oppElo, result);
    current.elo = qMax(0, current.elo + ec);
    if(result=="WIN")       ++current.wins;
    else if(result=="LOSS") ++current.losses;
```

```
    else                ++current.draws;
    MatchRecord m;
    m.opponent=opp; m.result=result; m.eloChange=ec;
    m.date=QDateTime::currentDateTime().toString("MMM dd yyyy").toUpper();
    current.history.prepend(m);
    auto players=loadAll();
    for(auto& p:players)
        if(p.name.toLower()==current.name.toLower()){ p=current; break; }
    saveAll(players);
}
```

5. Network Multiplayer

Royal Chess implements peer-to-peer TCP multiplayer using Qt's QTcpServer and QTcpSocket. Two players on the same Wi-Fi network can play a full game against each other in real time with no central server required.

5.1 Connection Flow

HOST PLAYER (plays as White)	JOIN PLAYER (plays as Black)
1. Clicks HOST GAME	1. Clicks JOIN GAME
2. QTcpServer binds port 45678	2. Enters host's IP address
3. App displays local IP address	3. QTcpSocket connects to host
4. Shares IP with joining player	4. Connection established
5. Game begins — plays as White	5. Game begins — plays as Black

5.2 Move Protocol

Every move is serialized as a plain-text line over the TCP connection:

```
// NetworkManager.cpp — sendMove()
void NetworkManager::sendMove(int fromR, int fromC, int toR, int toC){
    if(!isConnected()) return;
    QString msg = QString("MOVE %1 %2 %3 %4\n")
                  .arg(fromR).arg(fromC).arg(toR).arg(toC);
    m_socket->write(msg.toUtf8());
}
```

The receiving side buffers incoming data, splits on newline characters, and dispatches complete messages:

```
// NetworkManager.cpp — processLine()
void NetworkManager::processLine(const QString& line){
    if(line.startsWith("MOVE")){
        auto parts = line.split(' ');
        if(parts.size()==5 && onMoveReceived)
            onMoveReceived(parts[1].toInt(), parts[2].toInt(),
                           parts[3].toInt(), parts[4].toInt());
    }
}
```

5.3 Singleton Design

NetworkManager is a Singleton. Only one TCP connection can exist per application instance — having two would attempt to bind the same port twice and fail. The Singleton guarantee ensures this constraint is enforced at the language level:

```
// NetworkManager.h
class NetworkManager : public QObject {
public:
    static NetworkManager& inst(){ static NetworkManager n; return n; }
private:
    NetworkManager() = default;
    NetworkManager(const NetworkManager&) = delete;
    NetworkManager& operator=(const NetworkManager&) = delete;
};
```

6. Data Persistence

Player data is persisted to a JSON file at ~/Documents/RoyalChess/players.json using Qt's native QJsonDocument API. The file is a JSON array where each element is a player object containing their name, ELO rating, win/loss/draw counts, join date, and the full match history.

6.1 JSON Structure

```
{
 "name":     "Anuradha",
 "elo":      1020,
 "wins":     3,
 "losses":   1,
 "draws":    0,
 "joinDate": "MAY 2026",
 "history": [
  {
   "opponent":    "Knight AI",
   "result":      "WIN",
   "playerColor": "WHITE",
   "timeControl": "10 min",
   "moves":       42,
   "eloChange":   10,
   "date":        "MAY 20 2026"
  }
 ]
}
```

6.2 Why JSON and Not a Database

Qt provides native JSON support through QJsonDocument — zero external dependencies are required. The data structure is shallow: one array of player records, each with a flat history list. A relational database would add setup complexity, a runtime dependency, and migration overhead with no benefit at this data volume. The DataMgr Facade pattern means the storage format is completely hidden from the rest of the application. Switching from JSON to SQLite in the future would require changing only the four private methods in DataMgr — no other class would need to change.

6.3 Serialization

```
// DataMgr.cpp — saveAll() and loadAll()
void DataMgr::saveAll(const QList<PlayerProfile>& players){
   QDir().mkpath(QFileInfo(path()).absolutePath());
```

```
    QJsonArray arr;
    for(auto& p : players) arr.append(p.toJson());
    QFile f(path());
    if(f.open(QIODevice::WriteOnly)){
        f.write(QJsonDocument(arr).toJson());
        f.close();
    }
}

QList<PlayerProfile> DataMgr::loadAll(){
    QFile f(path());
    if(!f.exists() || !f.open(QIODevice::ReadOnly)) return {};
    auto arr = QJsonDocument::fromJson(f.readAll()).array();
    f.close();
    QList<PlayerProfile> list;
    for(auto v : arr) list.append(PlayerProfile::fromJson(v.toObject()));
    return list;
}
```

7. Design Principles

Royal Chess demonstrates seven core design principles. Each was applied to solve a concrete engineering problem.

Principle	Application in Royal Chess	Code Location
Single Responsibility	Board = chess rules only. BoardWidget = rendering only. DataMgr = data only. Each class has exactly one reason to change.	*Board.h has zero Qt widget includes*
Encapsulate What Varies	AI difficulty varies — encapsulated as the depth parameter. Storage format varies — encapsulated in DataMgr. Rendering varies — encapsulated in BoardWidget.	*AI.h: alphaBeta(Board&, int depth,...)*
Open-Closed	alphaBeta() is never modified to add difficulty levels — only the depth argument changes at the call site.	*AI.cpp: one function, three depth values*
Loose Coupling	Board has zero compile-time dependency on any UI class. BoardWidget communicates with MainWindow only through std::function callbacks.	*BoardWidget.h: onEvent callback*
Principle of Least Knowledge	BoardWidget never calls DataMgr::inst(). Board never calls BoardWidget::update(). Each class talks only to its direct collaborators.	*BoardWidget.cpp: zero DataMgr calls*
DRY	alphaBeta() used for all three difficulty levels. formatTime() used for both clocks. legalMoves() called from player handler, AI engine, and game-over detection.	*AI.cpp: one alphaBeta function*
Delegation	MainWindow delegates chess logic to BoardWidget, data operations to DataMgr::inst(), and network operations to NetworkManager::inst().	*MainWindow.cpp: board->startGame(pc, depth)*

8. Design Patterns

Royal Chess implements nine GoF design patterns. Each is demonstrated with real code from the repository.

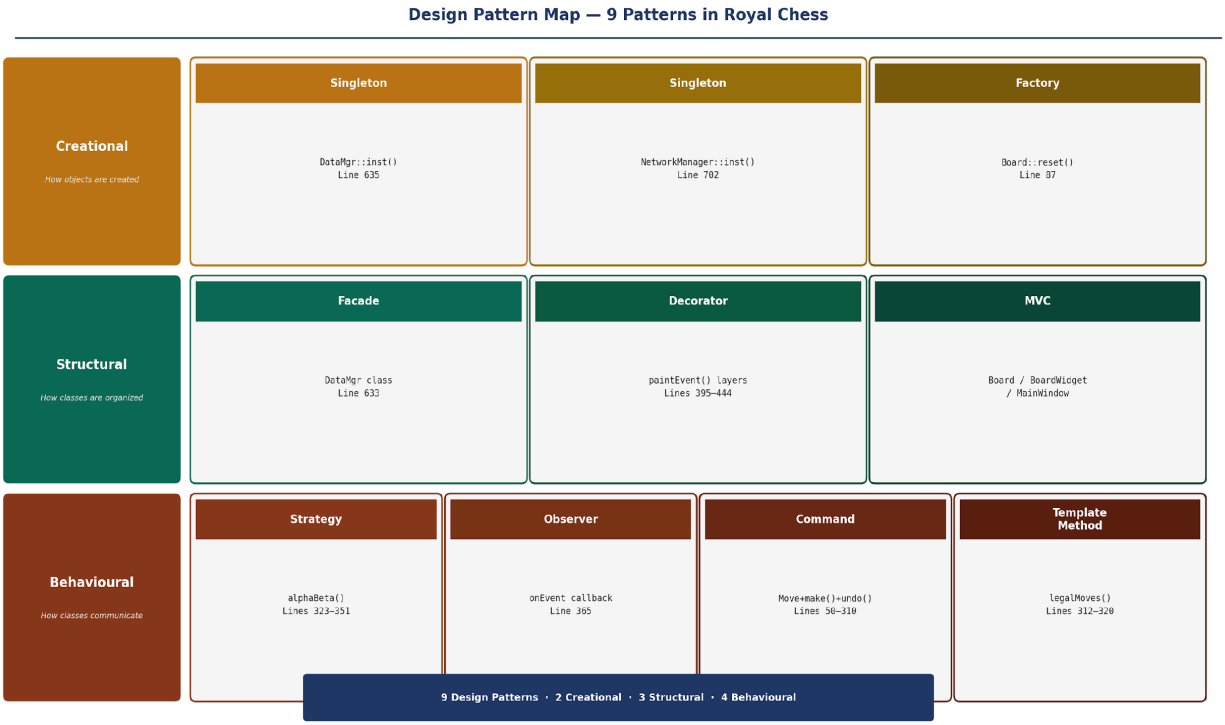


Figure 3 — Design Pattern Map: 9 patterns grouped by GoF category

8.1 Singleton Pattern — DataMgr and NetworkManager

> GoF: "Ensure a class has only one instance, and provide a global point of access to it."

```
// DataMgr.h — Singleton accessor + copy prevention
class DataMgr {
public:
   static DataMgr& inst(){ static DataMgr d; return d; }
private:
   DataMgr() = default;
   DataMgr(const DataMgr&) = delete;
   DataMgr& operator=(const DataMgr&) = delete;
};
```

DataMgr is a Singleton because it writes to a single file. Two instances writing simultaneously would corrupt the JSON. NetworkManager is a Singleton for the same reason — only one socket can bind

port 45678. The static local variable in inst() is initialized exactly once on the first call. The deleted copy constructor makes it impossible to create a second instance accidentally.

8.2 Strategy Pattern — AI Difficulty

GoF: "Define a family of algorithms, encapsulate each one, and make them interchangeable."

```
// AI.cpp — one algorithm, three strategies
int alphaBeta(Board& b, int depth, int alpha, int beta, bool maxing){
    if(depth == 0) return b.evaluate();
    Color col = maxing ? Color::White : Color::Black;
    auto moves = b.legalMoves(col);
    if(moves.empty()) return b.inCheck(col) ? (maxing?-15000:15000) : 0;
    if(maxing){
        int best = INT_MIN;
        for(auto& m : moves){
            b.make(m);
            best = std::max(best, alphaBeta(b,depth-1,alpha,beta,false));
            b.undo(m);
            alpha = std::max(alpha, best);
            if(beta <= alpha) break;  // β cut-off
        }
        return best;
    }
    // ... minimizing side mirrors the above
}
```

```
// MainWindow.cpp — strategy selected at runtime
int depth = easyBtn->isChecked()  ? 1 :
            hardBtn->isChecked()  ? 4 : 3;
board->startGame(playerColor, depth);
```

The alphaBeta function never changes when a new difficulty level is added — only the depth argument at the call site changes. Alpha-Beta pruning cuts the search from O(b^d) to O(b^(d/2)): at depth 4, from 810,000 nodes to approximately 900 nodes, a 900× speedup.

8.3 Observer Pattern — onEvent Callback

GoF: "Define a one-to-many dependency so that when one object changes state, all its dependents are notified automatically."

```
// BoardWidget.h — publisher declares the callback
std::function<void(const QString&)> onEvent;
```

```
// BoardWidget.cpp — publisher fires event (no knowledge of subscriber)
```

```
if(onEvent) onEvent("checkmate:" + winner);
if(onEvent) onEvent("stalemate");
if(onEvent) onEvent("check");
```

```
// MainWindow.cpp — subscriber wires up at game start
board->onEvent = [this](const QString& ev){
    if(ev.startsWith("checkmate")){
        DataMgr::inst().saveMatch(result, opp, oppElo, moves, col, tc);
        refreshHomeData();
    }
    if(ev == "check") statusLbl->setText("Check!");
};
```

BoardWidget fires events without knowing who will handle them. MainWindow decides the response — save the match, update the UI, show a popup. BoardWidget has zero dependency on MainWindow; it can be reused in any application unchanged.

8.4 Command Pattern — Move struct

GoF: "Encapsulate a request as an object, thereby allowing undoable operations."

```
// Types.h — the Command object
struct Move {
    Pos   from, to;
    PType piece=PType::None, cap=PType::None;
    Color pCol=Color::None,  cCol=Color::None;
    Pos   capPos;  // exact capture square (differs for en passant)
    bool  castle=false, enPassant=false, promotion=false;
    bool  kSide=false;
};
```

```
// Board.cpp — make() executes the command
void Board::make(const Move& m){
    if(m.enPassant) g[m.capPos.r][m.capPos.c]={PType::None,Color::None,false};
    if(m.castle){
        int rFrom=m.kSide?7:0, rTo=m.kSide?5:3;
        g[m.from.r][rTo]=g[m.from.r][rFrom];
        g[m.from.r][rFrom]={PType::None,Color::None,false};
    }
    // ... move piece, handle promotion, update ep, flip turn
}
```

```
// Board.cpp — undo() reverses it perfectly
void Board::undo(const Move& m){
    history.pop_back();
    auto& from=g[m.from.r][m.from.c];
    auto& to  =g[m.to.r  ][m.to.c  ];
    from=to; from.type=m.piece; from.moved=!history.empty();
    to={PType::None,Color::None,false};
```

```
    if(m.cap!=PType::None && !m.enPassant)
        g[m.capPos.r][m.capPos.c]={m.cap,m.cCol,true};
    if(m.enPassant)
        g[m.capPos.r][m.capPos.c]={PType::Pawn,m.cCol,true};
    // ... reverse castling, restore en passant target
}
```

Every chess move is stored as a Move object containing all information needed to both execute and reverse it. This enables the undo feature (board.undo(history.back())) and the AI tree search (the AI calls make/undo millions of times per second while exploring the game tree).

8.5 Template Method Pattern — Board::legalMoves()

GoF: "Define the skeleton of an algorithm, deferring some steps to subclasses."

```
// Board.cpp — fixed skeleton: generate → make → test → undo → collect
std::vector<Move> Board::legalMoves(Color col){
    std::vector<Move> legal;
    for(auto& m : pseudoMoves(col)){  // Step 1: generate candidates
        make(m);                      // Step 2: try the move
        if(!inCheck(col))             // Step 3: keep if king is safe
            legal.push_back(m);
        undo(m);                      // Step 4: restore board
    }
    return legal;
}
```

The four-step skeleton is invariant — it is always generate, try, filter, restore. The variant step is pseudoMoves(): pawns move forward and capture diagonally; knights jump in L-shapes; rooks slide in straight lines; bishops slide diagonally; queens combine rook and bishop movement; kings move one square in any direction. Template Method separates the fixed algorithm structure from the varying per-piece behavior.

8.6 Facade Pattern — DataMgr

GoF: "Provide a simplified interface to a complex subsystem."

```
// The one call MainWindow makes after any game:
DataMgr::inst().saveMatch(result, opp, oppElo, moves, col, tc);
```

```
// What DataMgr::saveMatch() does internally (DataMgr.cpp):
//  1. Calculates ELO change via PlayerProfile::calcElo()
//  2. Updates current.elo, current.wins/losses/draws
//  3. Prepends MatchRecord to current.history
```

```
//  4. Calls loadAll() — opens file, parses JSON array
//  5. Finds and updates the current player in the list
//  6. Calls saveAll() — serializes array, writes file
```

MainWindow never opens a file and never writes JSON. It calls one line and DataMgr handles everything. If the storage format were replaced with SQLite, the change would be entirely contained within DataMgr — no other class would need to change.

8.7 Factory Pattern — Board::reset()

GoF: "Define an interface for creating objects, letting the factory decide which class to instantiate."

```
// Board.cpp — Board::reset() creates all 32 pieces
void Board::reset(){
    for(auto& row:g) for(auto& s:row) s={PType::None,Color::None,false};
    turn=Color::White; ep={}; history.clear();
    auto place=[&](int r,int c,PType t,Color col){
        g[r][c]={t,col,false};
    };
    place(0,0,PType::Rook,  Color::White);
    place(0,1,PType::Knight,Color::White);
    // ... all 32 pieces placed here
    for(int c=0;c<8;c++) place(1,c,PType::Pawn,Color::White);
}
```

All piece creation is centralized in one function. No other class creates Square objects with piece types. Changing the starting position — for example implementing Chess960 (Fischer Random) — requires modifying only Board::reset().

8.8 MVC Pattern

Architectural pattern: separates an application into Model (data/logic), View (presentation), and Controller (coordination).

The three MVC layers in Royal Chess have no cross-layer dependencies beyond what MVC requires:

- Board (Model) — Board.h includes only Types.h and <array>. It compiles with no Qt widget headers. The chess engine is fully testable without a GUI.
- BoardWidget (View) — reads Board to render; communicates with MainWindow only through callbacks; never calls DataMgr or NetworkManager.
- MainWindow (Controller) — creates and coordinates all components; implements no chess logic; performs no drawing.

8.9 Decorator Pattern — paintEvent() visual layers

GoF: "Attach additional responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality."

```
// BoardWidget.cpp — paintEvent() applies layers sequentially
void BoardWidget::paintEvent(QPaintEvent*){
    QPainter p(this);
    int sz = squareSize();
    // Layer 1: base board squares
    for(int r=0;r<8;r++) for(int c=0;c<8;c++){
        bool light=(r+c)%2==0;
        p.fillRect(c*sz,(7-r)*sz,sz,sz,
            light?QColor("#D4A96A"):QColor("#8B5E1A"));
    }
    if(!gameActive){ drawCoords(p,sz); return; }
    // Layers 2-7: visual decorators
    drawLastMoveHighlight(p, sz);  // yellow tint on last move
    drawSelectionAndHints(p, sz);  // bright yellow + dots
    drawCheckOverlay     (p, sz);  // red king square
    drawPieces           (p, sz);  // Unicode piece glyphs
    drawCoords           (p, sz);  // rank/file labels
}
```

Each layer is a private method that decorates the canvas surface. Removing the check overlay requires deleting or commenting out one line. Adding a new visual effect — for example, highlighting threatened pieces — requires adding one new private method and one call site, with no modification to any existing layer.

9. Overall Application Quality

9.1 Technical Completeness

Royal Chess is a complete, working application that can be demonstrated live. All planned features are fully implemented:

- All standard chess rules work correctly including the two most complex special moves. Castling validates that neither the king nor rook has moved, that the king is not currently in check, and that the king does not pass through or land on an attacked square. En passant tracks the target square across turns and handles the case where the captured pawn is not on the destination square.
- The AI plays at a genuine intermediate level on Hard mode and responds in under two seconds. The evaluation function scores piece material correctly, and Alpha-Beta pruning makes depth-4 search feasible in real time.
- The player profile system persists across application restarts. Registration, login, points update, match history, and the leaderboard all work correctly from first launch through many sessions.
- TCP multiplayer connects two players on the same Wi-Fi network with synchronized board state. Move transmission is reliable and bidirectional.

9.2 Engineering Quality

- The nine design patterns and seven design principles were each applied to solve a real problem — none were added to satisfy a checklist.
- The multi-file architecture cleanly separates concerns. Each header file documents its own pattern and responsibility in comments.
- Re-entrancy is handled correctly: BoardWidget checks a gameActive flag before processing mouse events and the AI defers its move via QTimer::singleShot to avoid blocking the event loop.
- The application is stable for live demonstration of all features: registration, AI gameplay at all three difficulty levels, profile and history display, and the multiplayer HOST/JOIN flow.

9.3 Metrics Summary

Metric	Value	Significance
Project source files	**6 .cpp + 7 .h**	Clean multi-file structure
GoF design patterns implemented	**9**	All major categories covered
Design principles demonstrated	**7**	Full CMPE 202 coverage
AI search speedup (Alpha-Beta vs Minimax)	**~900×**	810,000 → ~900 nodes at depth 4

Classes with zero cross-layer dependencies	**Board, DataMgr, NetworkManager**	Strong cohesion
Data storage	**Qt JSON**	Zero external dependencies
Network protocol	**TCP port 45678**	Standard Qt networking

10. Challenges and Solutions

10.1 Legal Move Validation

The hardest chess rules to implement correctly were castling and en passant. Castling requires checking that the king is not currently in check, does not pass through an attacked square, and neither the king nor rook has moved. En passant requires tracking the target square from the previous move and storing the capture position separately from the destination square in the Move struct.

10.2 Preventing UI Freeze During AI Search

At depth 4, the AI evaluates hundreds of positions. Running this on the main thread would freeze the UI. The solution was to defer the AI call using QTimer::singleShot(100, this, &BoardWidget::doAI), which lets Qt process pending paint events before starting the search.

10.3 Maintaining MVC Separation

As features were added, there was constant pressure to let BoardWidget call DataMgr directly or let MainWindow implement chess logic. Strict discipline was required to route all game events through the onEvent callback and keep all chess rules inside Board.

10.4 Multiplayer State Synchronization

TCP delivers data as a stream, not as discrete messages. A partial MOVE message could arrive in one read. The solution was a line-based protocol — messages end with a newline — and a buffer that accumulates incoming bytes and only dispatches complete lines.

11. Testing

11.1 Chess Engine Testing

Legal move generation was tested systematically for all six piece types. Key cases verified: pawns cannot capture forward; castling is blocked when the king is in check or would pass through check; en passant can only be played on the move immediately following the opponent's two-square pawn advance; a move that leaves the king in check is not returned by legalMoves().

Checkmate and stalemate detection were verified against several well-known positions including Fool's Mate (checkmate in 2 moves) and a known stalemate position with only kings remaining.

11.2 Undo Correctness

Undo was tested across all special move types: normal captures, castling (both sides), en passant, and pawn promotion. After undo(), the board state including en passant target and the moved flag on all pieces was verified to match the pre-move state.

11.3 Data Persistence

Player profiles were verified to survive application restarts. A player was registered, games were played, the application was closed and reopened, and the profile including ELO, win/loss counts, and full match history was confirmed to load correctly.

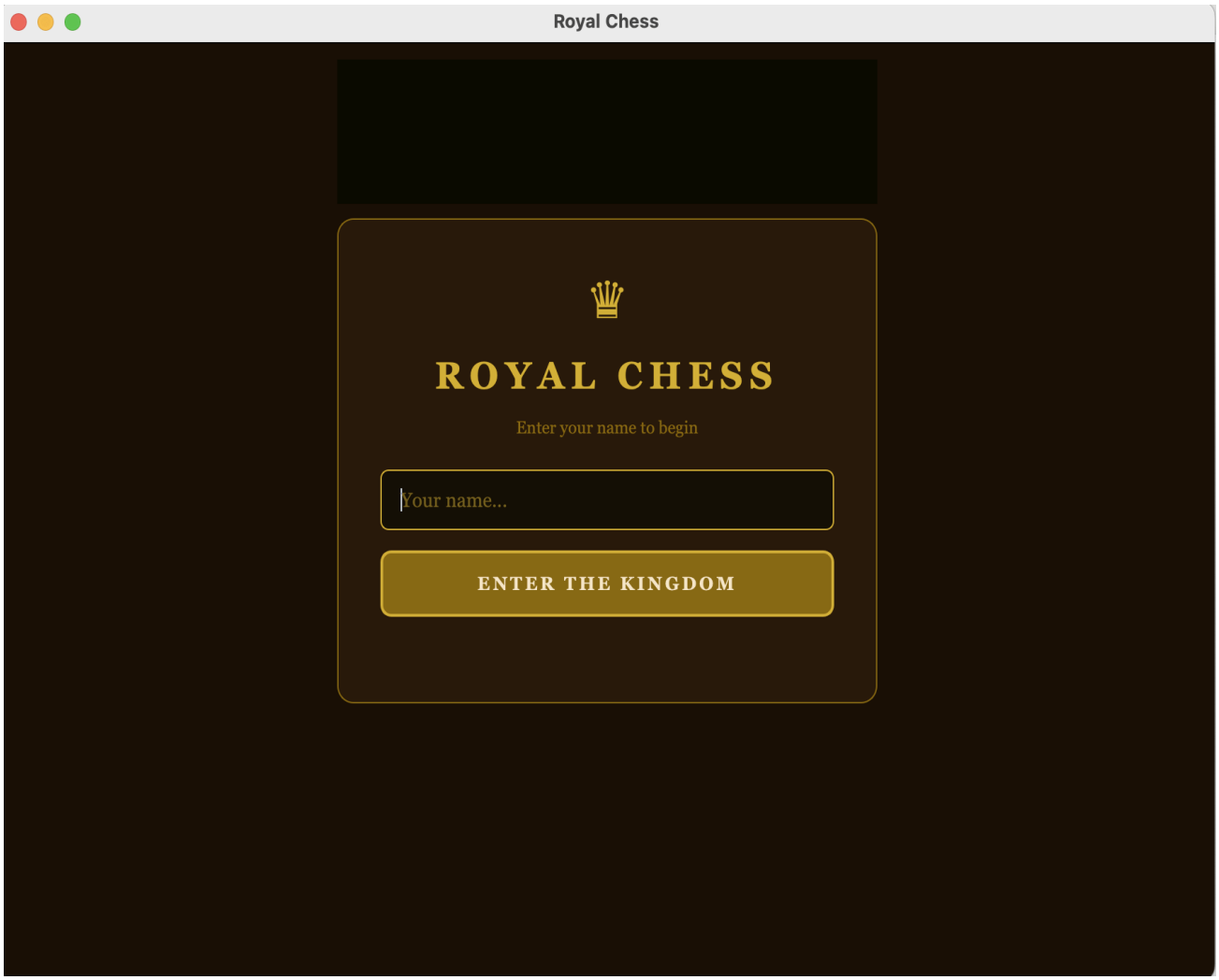
11.4 Multiplayer Synchronization

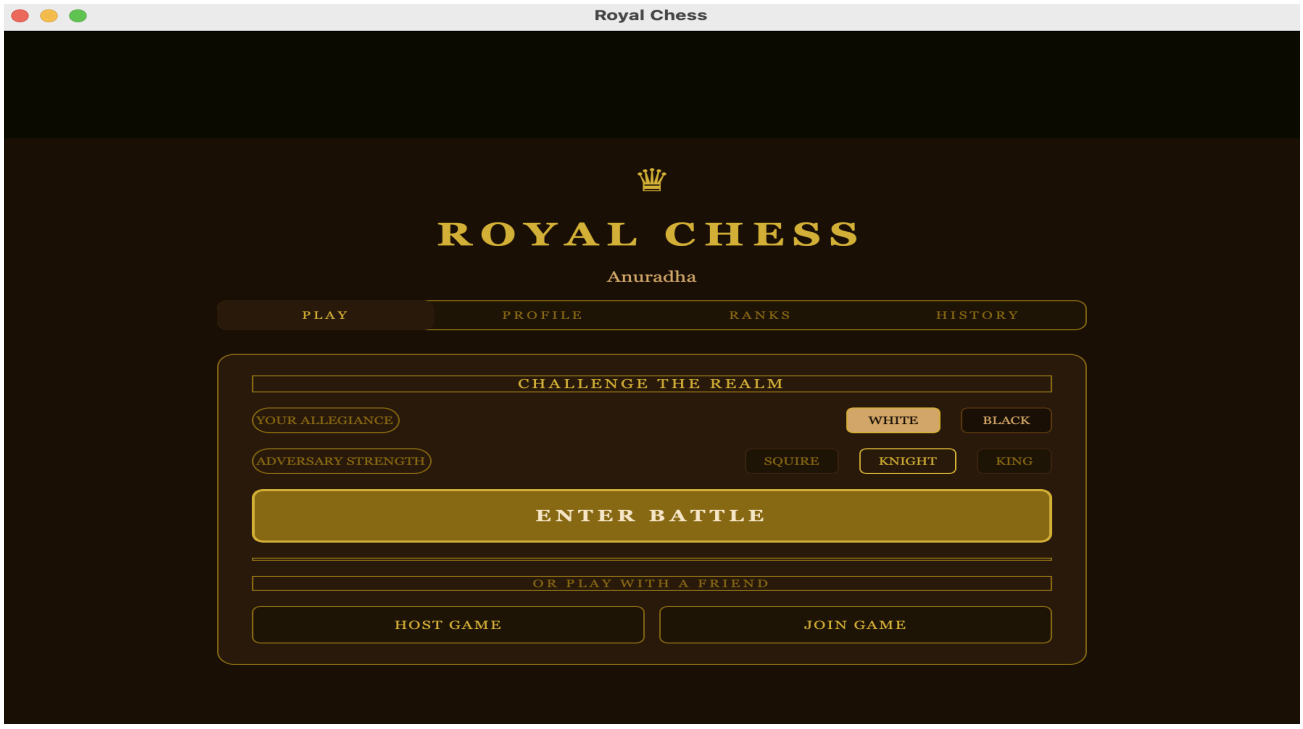
Two instances of the application were run on the same machine (one as host, one as client using 127.0.0.1). Move transmission was verified in both directions, and board state was confirmed identical on both sides after each move.

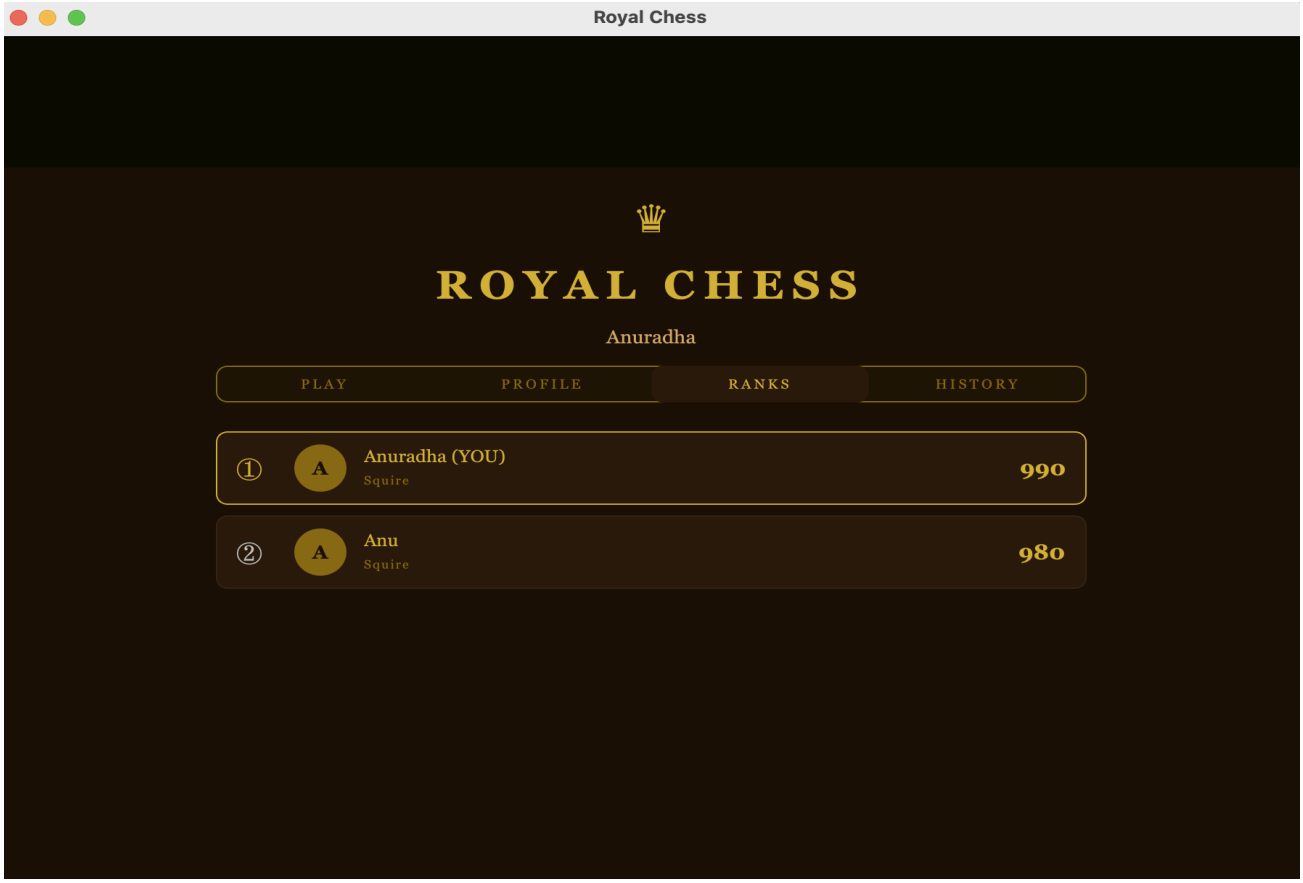
12. Future Improvements

- SQLite backend — replacing the flat JSON file with SQLite would support larger player databases and concurrent access, requiring only changes to DataMgr.
- Stronger AI evaluation — the current evaluation function scores only material balance. Adding positional scores (piece-square tables), pawn structure analysis, and king safety would significantly improve Hard mode play strength.
- Online multiplayer — the current TCP implementation requires both players on the same Wi-Fi network. A relay server with WebSocket support would enable play across the internet.
- Move notation and replay — recording games in standard algebraic notation (e.g. e4, Nf3, O-O) and providing a replay mode to step through completed games.
- Animations — smooth piece movement animations using Qt's QPropertyAnimation would improve the visual experience.

13. Screenshots

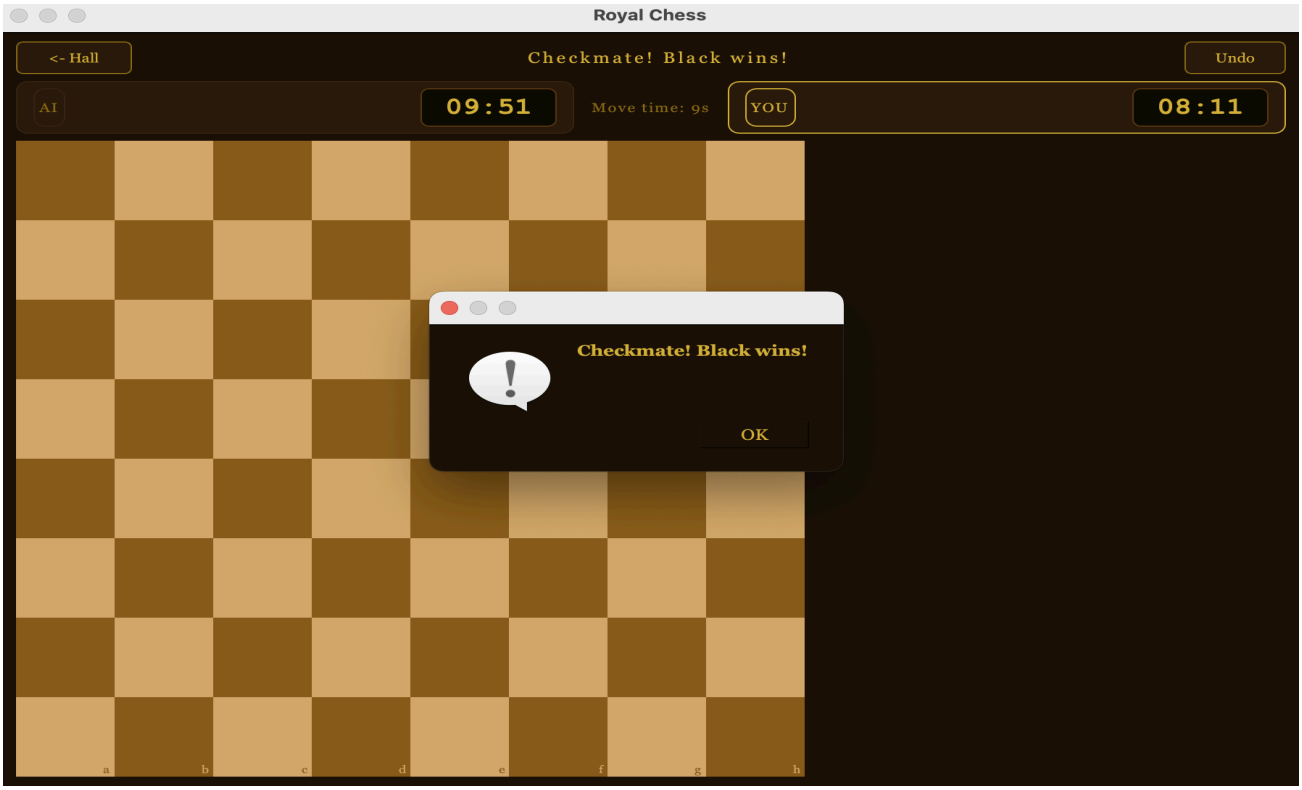


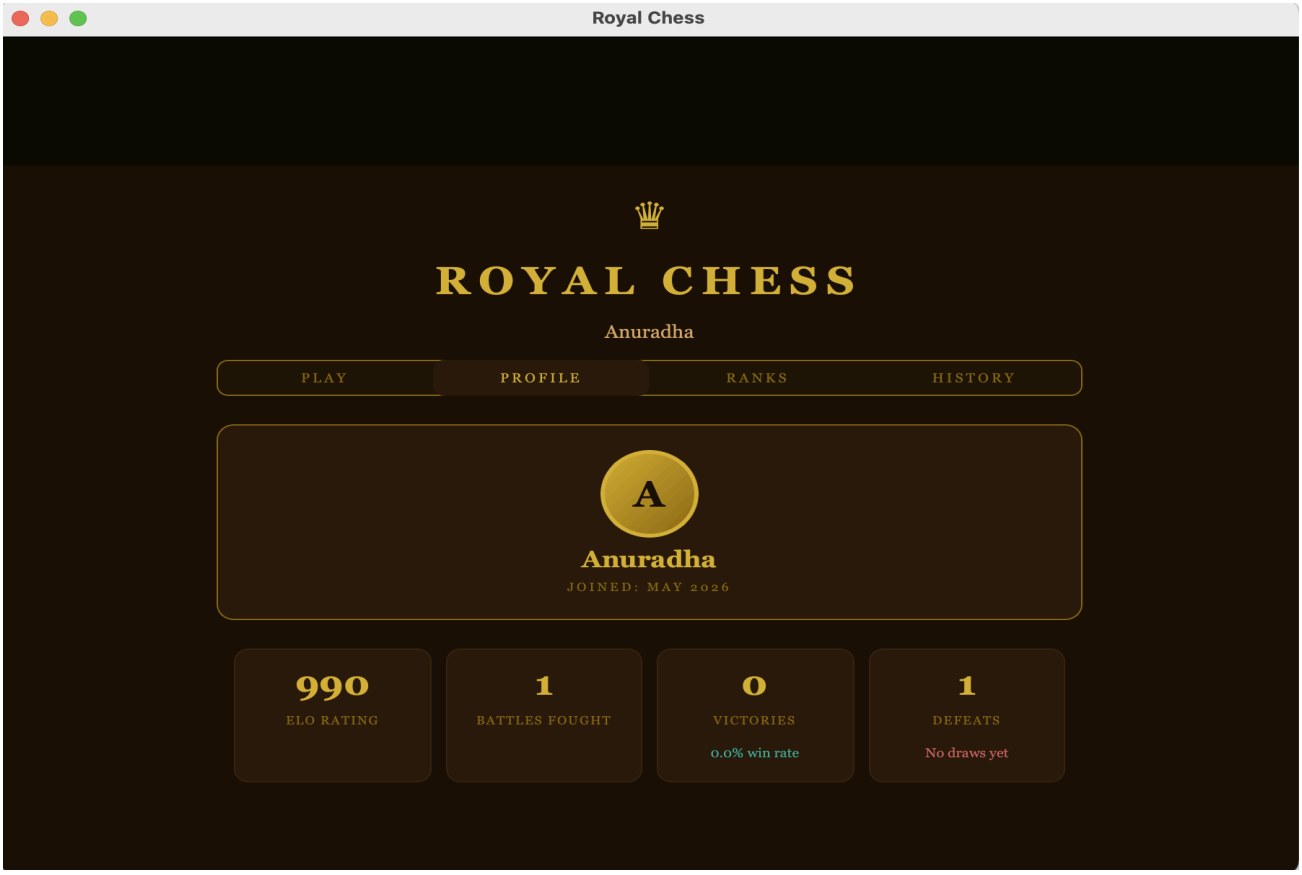

Royal Chess
ROYAL CHESS
Anuradha
PLAY
PROFILE
RANKS
HISTORY
CHALLENGE THE REALM
YOUR ALLEGIANCE
WHITE
BLACK
ADVERSARY STRENGTH
SQUIRE
KNIGHT
KING
ENTER BATTLE
OR PLAY WITH A FRIEND
HOST GAME
JOIN GAME

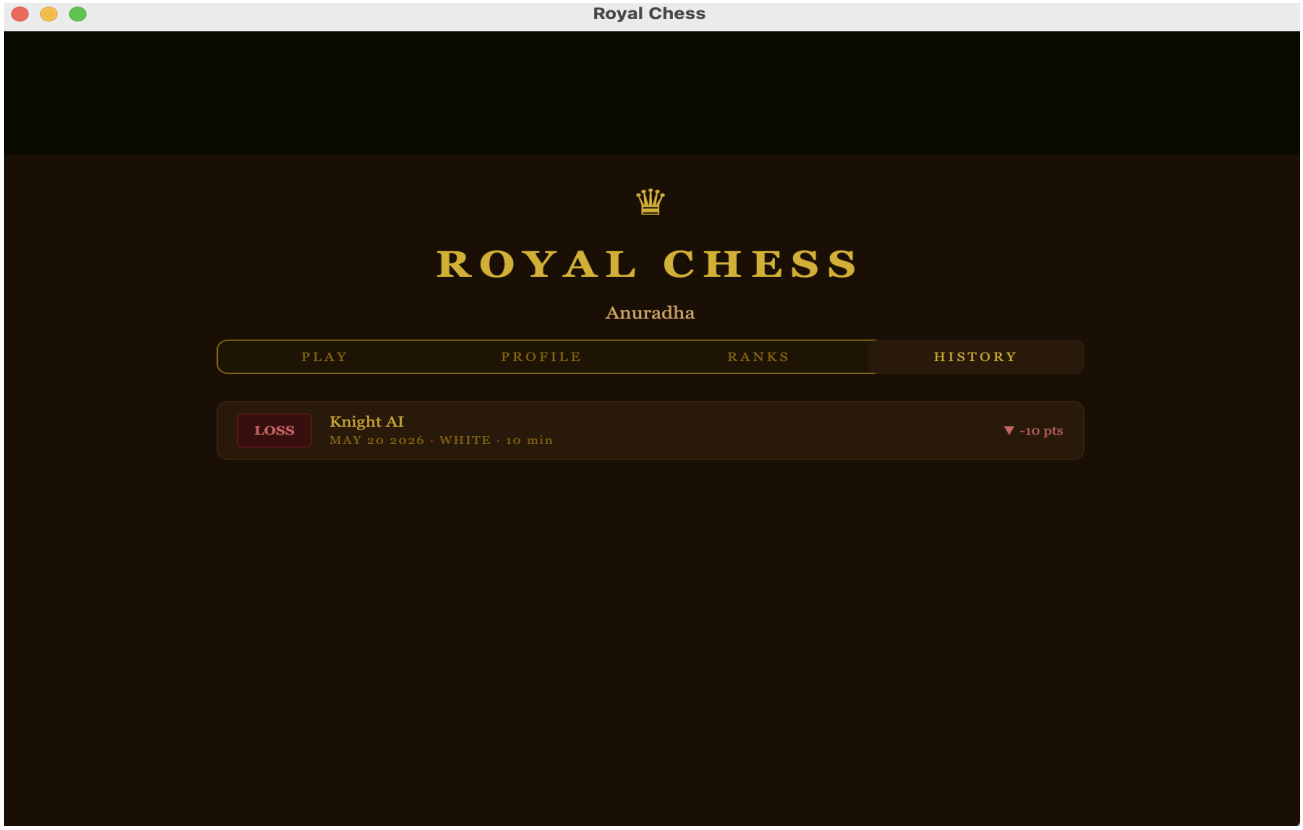

Royal Chess
ROYAL CHESS
Anuradha
PLAY
PROFILE
RANKS
HISTORY
① A Anuradha (YOU) Squire 990
② A Anu Squire 980


Royal Chess
<- Hall
White's Turn (Your Move)
Undo
AI
10:00
Move time: 32s
YOU
09:18


Royal Chess
<- Hall
White's Turn (Your Move)
Undo
AI
10:00
Move time: 8s
YOU
08:34


Royal Chess
<- Hall
Checkmate! Black wins!
Undo
AI
09:51
Move time: 9s
YOU
08:11
Checkmate! Black wins!
OK


Royal Chess
ROYAL CHESS
Anuradha
PLAY
PROFILE
RANKS
HISTORY
A
Anuradha
JOINED: MAY 2026
990
ELO RATING
1
BATTLES FOUGHT
0
VICTORIES
0.0% win rate
1
DEFEATS
No draws yet


Royal Chess
ROYAL CHESS
Anuradha
PLAY
PROFILE
RANKS
HISTORY
LOSS
Knight AI
MAY 20 2026 · WHITE · 10 min
▼ -10 pts