Chess

Game

Advanced Programming

António Domingos Gonçalves Pedroso- a2021132042

Carolina Vilas Boas Veloso- a2021140780

Luís Miguel Martins Duarte- a2021137789

INTRODUCTION

This work was carried out within the scope of the Advanced Programming Curricular Unit, in the 2nd Semester of the Degree in Computer Engineering - European Course in the academic year 2024/2025. It consists of the development of a chess game implemented according to the preferences and rules imposed by the head teacher.

CHESS GAME

The main objective of this work is to implement a chess game, which allows us to play with 2 players and make the three special moves such as castling, en passant and pawn promotion and still have the possibility to play with sound.

DECISIONS

Design Choices

The project adopts a modular architecture, separating responsibilities across distinct packages: game logic, board state, and individual chess pieces.

A **Facade Pattern** is employed through the *ChessGameManager* class, which acts as a central interface to manage the game's complexity. It coordinates the board, players, rules, and moves, hiding internal complexities from higher-level components.

To ensure a clear division of responsibilities, the system separates game state persistence, logging, and rule enforcement. This is used in classes such as *ChessGameSerialization* for saving/loading, and *ModelLog* for logging.

Additionally, the use of PropertyChangeSupport enables a reactive programming style. This introduces an **Observer/Observable Pattern**, allowing components (e.g., the UI) to listen for and respond to changes in the game state automatically.

Implementation Decisions

Each chess piece (e.g., *Pawn*, *Bishop*, *King*) is implemented in its own class with self-contained movement logic. Which can facilitate when being used for more complex rules such as *en passant*.

The **Memento Pattern** is implemented in the system to support undo/redo functionality. The *ChessGame* class acts as the Originator, creating memento objects (MyMemento) that encapsulate the current board state, active player, and a serialized snapshot of the game. These are managed by a *CareTaker* class. This allows for non-intrusive state saving and restoration, without violating encapsulation.

The use of a **Factory Pattern** simplifies the creation of piece instances. During game initialization or deserialization, this allows pieces to be instantiated dynamically based on their type, improving flexibility.

The **Singleton Pattern** is implemented in both the *SoundManager* and *ModelLog* classes. In *SoundManager*, it guarantees centralized and consistent audio control across the application. In *ModelLog*, it ensures unified logging without duplicating logger instances.

Diagrams

1. Design Patterns Used

This chess project efficiently applies several well-known design patterns to organize the code, separate responsibilities, and facilitate maintainability and extensibility.

1.1. Observer Pattern

Objective:

Allow different GUI components to be automatically updated whenever the game state changes.

Where it is applied:

- *ChessGameManager* uses PropertyChangeSupport to fire events to the observers using:

```
pcs.firePropertyChange(PROP_BOARD_UPDATE, oldValue: null, cgGame.getBoardPiecesString());
pcs.firePropertyChange(PROP_GAME_STATE, oldValue: null, newValue: null);
```

- *CanvasBoard* and *RootPane* listen to events with addPropertyChangeListener to react to state changes:

```
data.addPropertyChangeListener(ChessGameManager.PROP_GAME_STATE, PropertyChangeEvent evt -> {
    //System.out.println("Game state");
    updateStatus();
});
```

Observer Pattern Diagram

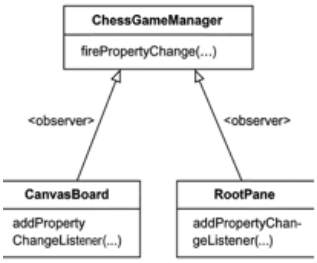


Figure 1 - Relationship between *ChessGameManager*, *CanvasBoard*, and *RootPane* using the Observer pattern.

1.2. Memento Pattern

Objective:

Allow undo and redo of moves by saving/restoring the full game state without violating encapsulation.

Where it is applied:

- *ChessGame* implements *IOriginator*
- *CareTaker* stores previous states (as mementos)

Memento:

```
public interface IMemento {  10 usages  CarolinaVeloso
    default Object getSnapshot() { return null; }
}
```

Code usage:

```
public boolean movePiece(String SoldPos, String SnewPos) {  11 usages  CarolinaVeloso +3
    try {
        oldGameState = cgGame.getBoardPiecesString();
        careTaker.save();
        boolean pieceMoved = cgGame.movePiece(SoldPos, SnewPos);
```

Memento Pattern Diagram

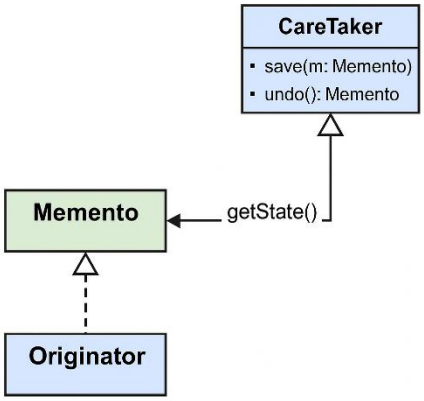


Figure 2 - Implementation of the Memento pattern with *ChessGame*, *CareTaker*, and *MyMemento*.

1.3. Factory Method Pattern

Objective:

Centralize and abstract the creation of chess pieces (Queen, Rook, etc.), especially useful for pawn promotion.

Where it is applied:

```
case 'Q':
    piece = QUEEN.createPiece(ePieceColor, board, column, row);
```

Real example:

```
Piece newPiece = switch (normalizedType) {
    case 'Q' -> PieceFactory.QUEEN.createPiece(color, board, column, row);
    case 'R' -> PieceFactory.ROOK.createPiece(color, board, column, row);
    case 'B' -> PieceFactory.BISHOP.createPiece(color, board, column, row);
    case 'N' -> PieceFactory.KNIGHT.createPiece(color, board, column, row);
    default -> throw new IllegalArgumentException("Invalid piece: " + pieceType);
};
```

Benefit:

Makes it easier to change or extend types of pieces without modifying promotion or initialization logic.

Factory Method Pattern Diagram

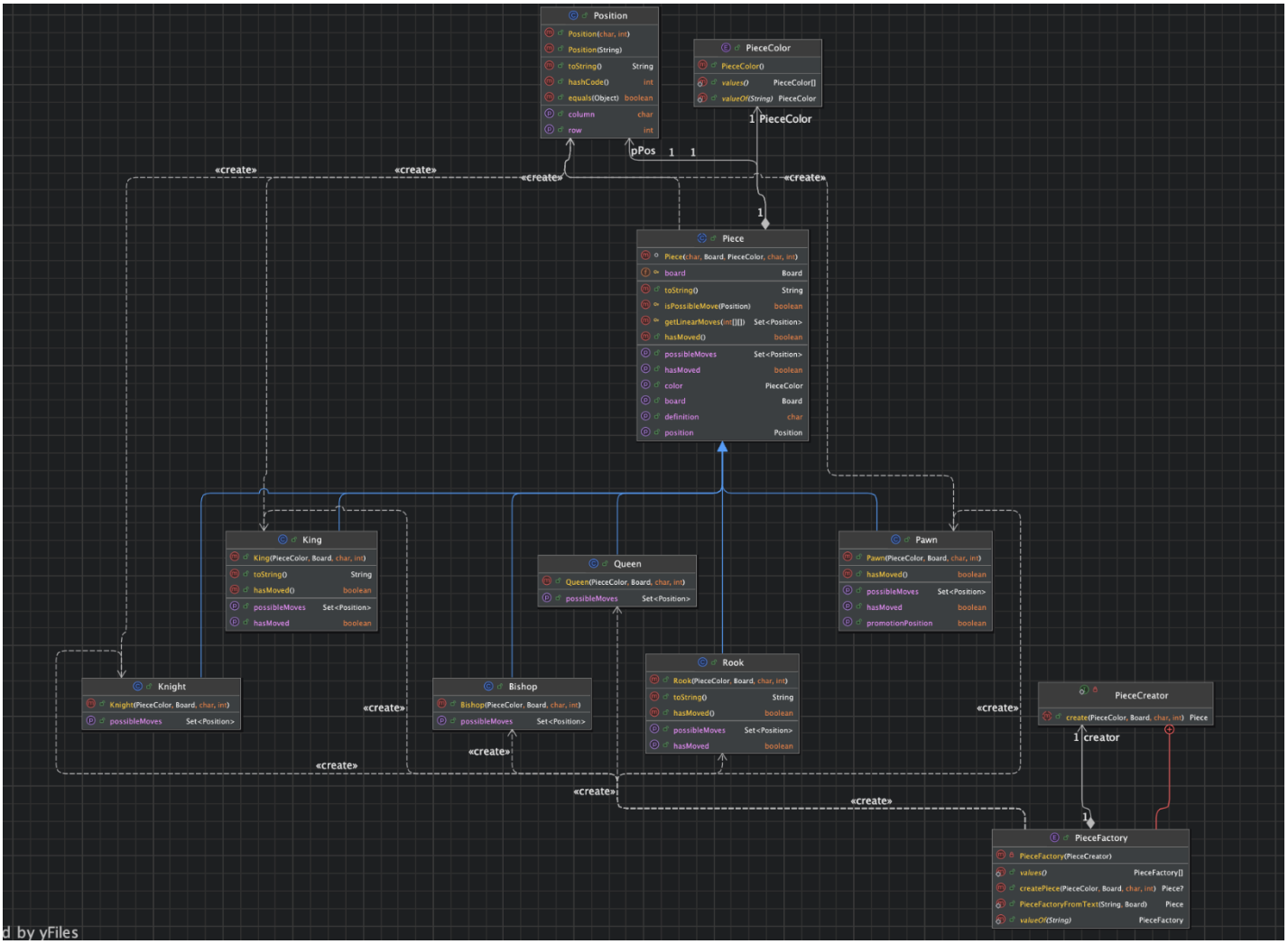


Figure 3 - *PieceFactory* creating instances of *Piece* subclasses.

1.4. Singleton Pattern

Objective:

Provide a single access point for logging game actions.

Where it is applied:

```
public static ModelLog getInstance() {   Luisuu
    if (_instance == null) _instance = new ModelLog();
    return _instance;
}
```

- *ModelLog* is a Singleton

Typical usage:

```
ModelLog.getInstance().log( msg: "New game started.");
```

Benefit:

Prevents multiple inconsistent log instances and allows central output to *LogList*.

Singleton Pattern Diagram

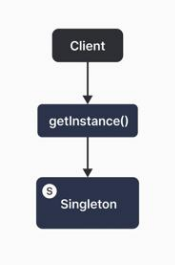


Figure 4 - Implementation of the Singleton pattern with *ModelLog*.

Classes Description

Classes	Description
Board	Represents the game board and manages piece positions and movement. Acts as a storage structure only — game logic is handled elsewhere.
Position	Utility class to represent coordinates on the board (column and row).
PieceColor	Enum representing the color of a piece: WHITE or BLACK.
Piece	Abstract base class for all chess pieces. Contains common behavior like position and color.
King	Represents the King piece and implements movement logic including castling.
Queen	Represents the Queen piece and combines Rook and Bishop move capabilities.
Rook	Represents the Rook and moves in straight lines (horizontal and vertical).
Bishop	Represents the Bishop and moves diagonally across the board.
Knight	Represents the Knight, moves in an L-shape and can jump over other pieces.
Pawn	Represents the Pawn and has unique forward movement, capturing logic, and "en passant" support.
PieceFactory	Implements the Factory Method pattern to create Piece objects from Enums or text representations.
ChessGame	Core logic of the chess game it also manages the board, turn order, move execution, and rule enforcement.

ChessGameManager	Acts as a Facade between the UI and the model. Exposes only necessary methods, hides internal details it also handles undo/redo and file operations.
ChessGameSerialization	Provides static methods to serialize and deserialize ChessGame objects for saving/loading.
ModelLog	Singleton class that stores a log of game events (e.g., moves, errors). Observable to support UI updates.
IOriginator	Interface implemented by objects (like ChessGame) that can generate mementos of their state and restore themselves from one.
IMemento	Interface representing a memento object — a snapshot of the originator's state at a given time.
CareTaker	Stores and manages mementos. Handles undo/redo functionality by maintaining a history stack of game states.
CanvasBoard	JavaFX component extending Canvas. Responsible for drawing the chess board and pieces. Uses MVC@PA structure.
RootPane	JavaFX root layout. Sets up the UI components, menu, and user interaction handlers. Acts as View-Controller.
MainJFx	JavaFX Application subclass that launches the app and initializes the main UI.
ChessMain	Contains the main method to launch the application.
AskName	Simple UI dialog to prompt and collect player names when starting a new game.
LogList	UI view for displaying logs from ModelLog using a JavaFX ListView.
ImageManager	Manages image loading and caching for drawing chess pieces.
SoundManager	Plays audio descriptions of moves and handles toggling sound on/off.

Class Relationships and Architecture

This project is structured following the **Model-View-Controller (MVC)** principle, supported by multiple design patterns that define relationships between components.

1. General Architecture

- **Model Layer**:
  - Core logic resides here: *ChessGame*, *Board*, *Piece*, *CareTaker*, *ModelLog*.
  - Responsible for game rules, state, undo/redo, and persistence.
  - *ChessGame* interacts with *Board*, which holds *Piece* objects (e.g., *Pawn*, *Queen*).
  - Uses **Memento** for state history and **Factory** for piece creation.
  - *ModelLog* is a **Singleton** managing logs across the system.
- **View Layer (UI)**:
  - Built with JavaFX: *CanvasBoard*, *RootPane*, *LogList*, etc.
  - *CanvasBoard* draws the board and listens to mouse clicks.
  - *RootPane* manages menus, layout, and dialog interactions.
- **Controller Layer**:
  - *ChessGameManager* acts as a **Facade/Controller**:
    - Mediates between model and view.
    - Uses **Observer Pattern** to notify UI on changes via PropertyChangeSupport.

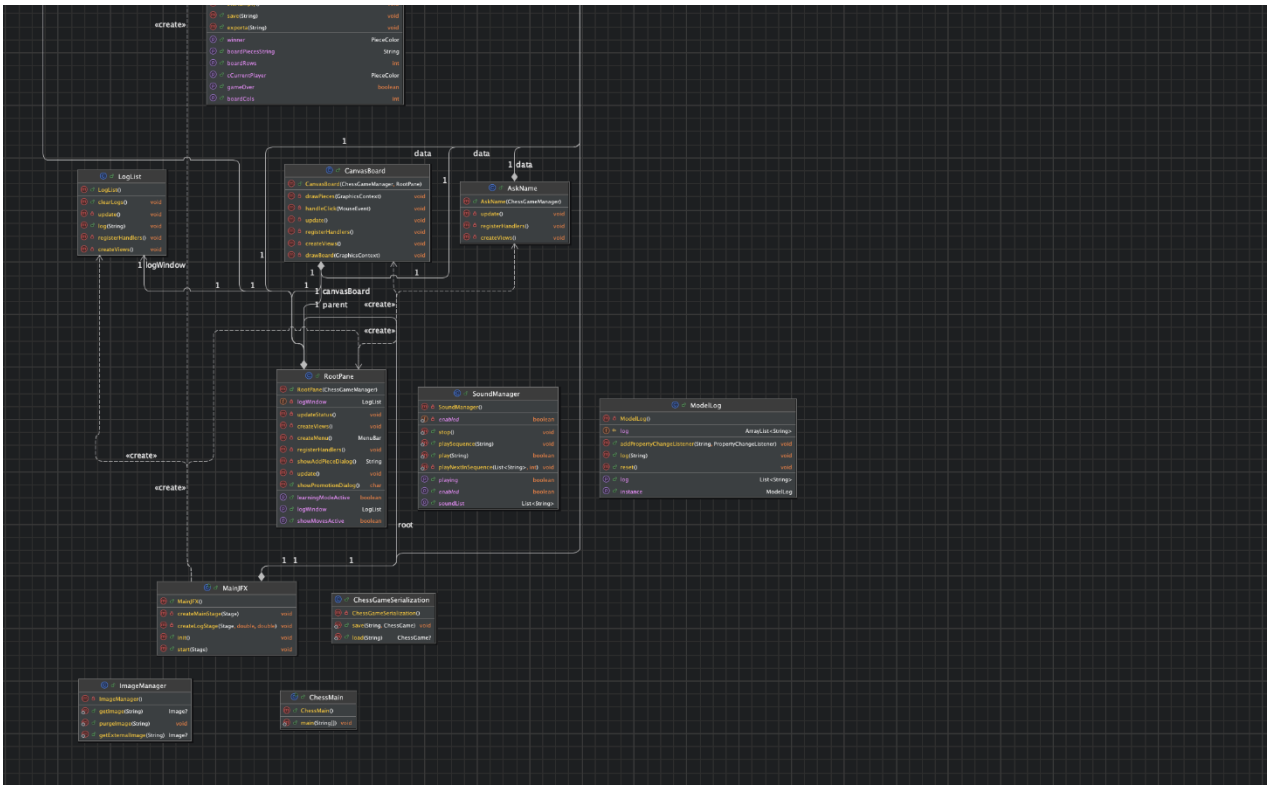
2. Full UML Class Diagram

The following image shows the class relationships and interactions between components:

Notable relationships:

- *ChessGameManager* links the model (*ChessGame*) to the UI.
- *CareTaker* and *ChessGame* form the **Memento** structure.
- *PieceFactory* dynamically generates *Piece* instances (**Factory Method**).
- *ModelLog* is accessed globally (**Singleton**).
- UI classes listen to model changes through property listeners (**Observer**).


Position
Board
PieceColor
IMemento
MyMemento
IOriginator
Piece
Queen
Knight
ChessGame
CareTaker
Rook
Bishop
King
Pawn
PieceCreator
ChessGameManager
PieceFactory
selectedPos
lastMoveFrom
mBoard
pPos
board
currentPlayer
cCurrentPlayer
history
originator
lastMovedPiece
cgGame
careTaker
gameManager
creator
«create»


«create»
data
data
data
LogList
CanvasBoard
AskName
logWindow
canvasBoard
parent
«create»
«create»
«create»
«create»
RootPane
SoundManager
ModelLog
root
MainJFX
ChessGameSerialization
ImageManager
ChessMain

Unit Test

testBoardInitialization()

Objective: Verifies that the game board initializes with the correct number and positions of pieces.

Algorithm: Confirms the presence of 32 pieces, with specific checks for the white king and a black pawn at expected starting positions.

testPieceMovement()

Objective: Tests basic piece movement and turn progression.

Algorithm: Moves a pawn from e2 to e4, checks that the piece moves correctly, updates the board state, and verifies the turn switches to black.

testCheckmateDetection()

Objective: Simulates the Fool's Mate to test checkmate detection.

Algorithm: Executes a series of moves that should result in white being checkmated. Confirms that checkmate is correctly detected and the winner is set.

testUndoFunctionality()

Objective: Tests the undo feature using the memento pattern.

Algorithm: Makes a move, verifies that undo is available, then undoes the move and checks that the board and player state revert to the original.

testPawnPromotion()

Objective: Tests pawn promotion mechanics.

Algorithm: Sets up a board where a white pawn is about to promote, moves it to the last rank, and promotes it to a queen. Verifies the promotion was successful.

testGameSerialization()

Objective: Verifies saving and loading the game state using serialization.

Algorithm: Saves the current game with player names and a move, performs another move, loads the saved state, and ensures the original state was restored correctly.

Implementation Scale Table

Classes	Implementation Scale
Board	☑ Fully Implemented
Position	☑ Fully Implemented
PieceColor	☑ Fully Implemented
Piece	☑ Fully Implemented
King	☑ Fully Implemented
Queen	☑ Fully Implemented
Rook	☑ Fully Implemented
Bishop	☑ Fully Implemented
Knight	☑ Fully Implemented
Pawn	☑ Fully Implemented

Classes	Implementation Scale
PieceFactory	☑ Fully Implemented
ChessGame	☑ Fully Implemented
ChessGameManager	☑ Fully Implemented
ChessGameSerialization	☑ Fully Implemented
Normalized text representation of pieces	☑ Fully Implemented
Import/export	☑ Fully Implemented
MenuBar with game options	☑ Fully Implemented
JavaFX UI (MainJFX, CanvasBoard, RootPane)	☑ Fully Implemented
Log system (Model Log, LogList)	☑ Fully Implemented
ChessMain	☑ Fully Implemented

Classes	Implementation Scale
Player name request (AskName)	☑ Fully Implemented
Board drawing via JavaFX Canvas	☑ Fully Implemented
Pawn Promotion	☑ Fully Implemented
En Passant	☑ Fully Implemented
Castling	☑ Fully Implemented
Learning mode menu and options	☑ Fully Implemented
Show possible moves	☑ Fully Implemented

Classes	Implementation Scale
Undo/Redo	☑ Fully Implemented
Observability with PropertyChangeSupport	☑ Fully Implemented
Sound toggle	☑ Fully Implemented
Audio feedback	☑ Fully Implemented
Board Editor	☑ Fully Implemented
Unit Test	☑ Fully Implemented
Javadoc	☑ Fully Implemented

Conclusion

This project provided a valuable opportunity to apply Java programming, and key design patterns such as Facade, Factory Method, Memento, and Observer. All required features were successfully implemented, along with the optional board editor. Features like undo/redo, learning mode, and audio feedback were also included. Overall, the project met all objectives and offered a technically rewarding experience.