Instituto Superior de Engenharia de Coimbra

Programação Avançada

Trabalho Prático

Jogo de Xadrez

Licenciatura Engenharia Informática 2024 / 2025

Duarte Xavier de Oliveira Santos

a2022149622@isec.pt

Gustavo Trigo Costa

a2023145800@isec.pt

Índice

1. Introdução

Este projeto tem como objetivo o desenvolvimento de um jogo de xadrez utilizando a linguagem Java e a biblioteca gráfica JavaFX. Foram implementados não só as funcionalidades básicas (como o mover/capturar peças), mas também os movimentos especiais (roque, en passant e promoção de peões), exportação e importação de jogos, feedback sonoro para tornar o jogo mais acessível a pessoas com alguma deficiência e por fim um modo de aprendizagem onde é possível fazer undo/redo e ver os movimentos possíveis de cada peça.

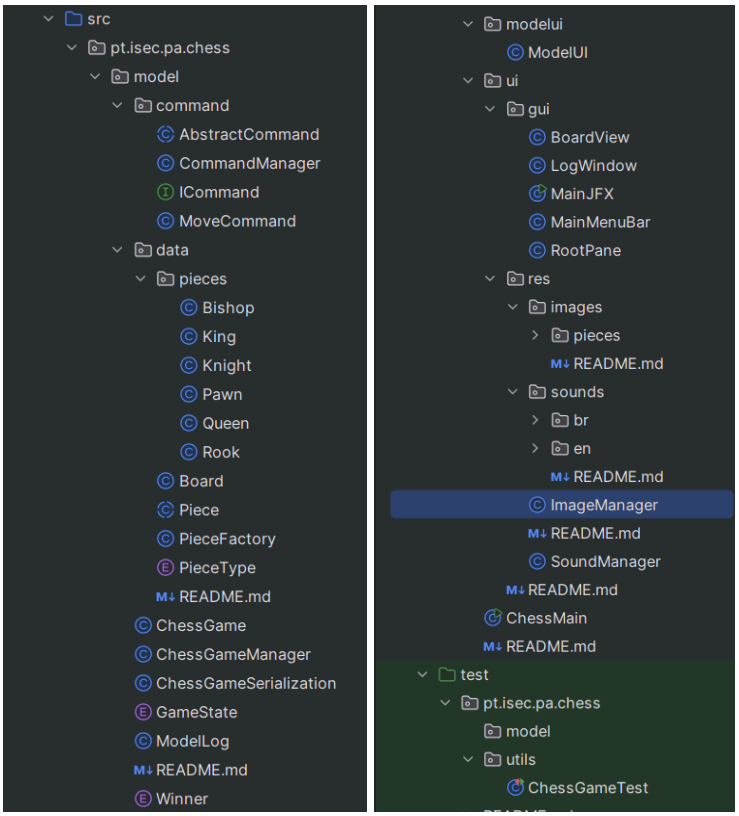
Ao longo do trabalho foram aplicados vários patterns de design abordados em aula, como por exemplo o MVC, Command, Observer e Singleton.

A separação entre a lógica do jogo e interface, foi tomada sempre bastante em conta, permitindo assim uma clara distinção entre a camada do modelo e a de visualização/controlador.

Neste relatório iremos descrever as principais decisões de design tomadas, os patterns aplicados e as relações entre as classes envolvidas.

2. Principais decisões de Design e Implementação

O nosso trabalho está organizado da seguinte maneira:



Para tornar o processo da criação de peças mais simples, optámos por implementar diferentes métodos de fábrica, seguindo o padrão **Factory Method.**

- **Fábrica por tipo de peça:** foi desenvolvido um método que gera peças com base no seu tipo (através da enumeração 'PieceType'), recebendo como parâmetros o tipo da peça, a coluna/linha onde esta seria criada, a sua cor (branca ou preta) e se a peça já teria efetuado algum movimento. Este método permite-nos agilizar o processo da criação de peças em momentos específicos do jogo, tais como no começo do jogo ou por exemplo quando um peão está a ser promovido.

- **Fábrica por representação textual (string):** foi ainda implementado um segundo método, que interpreta uma representação textual, por exemplo "Pb7" (onde 'P' simboliza o tipo da peça e a sua cor, através de ser maiúscula ou não, 'b' a sua coluna e '7' a sua linha).
  Este método irá receber uma String, e irá extrair a informação necessária para a criação da peça (tipo, coluna, linha, cor, e se já se moveu, caso a string contenha '*' , após a representação da peça significa que esta ainda não se moveu). Este método é útil, por exemplo em casos de importação de jogos.

Foi seguido um padrão **MVC** (Model-View-Controller), onde se segue uma separação clara entre a lógica do jogo ('model' como por exemplo ChessGame, ChessGameManager que serve de facade de ChessGame, Board, Piece, etc…) , interface gráfica ('View-Controller' como por exemplo na classe BoardView, RootPane, MainMenuBar, MainJFX) e controlo de interações.

A classe ChessGame foi feita para lidar com toda a lógica do jogo de xadrez, que seguindo rigorosamente o padrão de Facade, nos permite gerir toda a complexidade interna do jogo, ao mesmo tempo que oferece uma interface simples e segura aos componentes externos. Isso apenas é possível, pois nesta classe não retornamos ou aceitamos instâncias de objetos internos.

Para a parte da interface gráfica, continuamos a seguir a arquitetura MVC, e para tal tivemos que criar outra classe que segue rigorosamente o padrão de Facade, a ChessGameManager que fornece os métodos públicos da ChessGame.

Fazemos uso também do PropertyChangeSupport (padrão Observer), que está presente em vários locais no código, de modo a notificar os componentes da interface sobre as alterações feitas ao longo do jogo. Onde depois BoardView, RootPane entre outros componentes da UI se registam como 'Listeners'. Atualizando assim a view automaticamente.

Também usamos serialização em métodos da classe ChessGameSerilization de modo que com a lógica presente em ChessGame/ChessGameManager permitam guardar e restaurar por completo o estado do jogo.

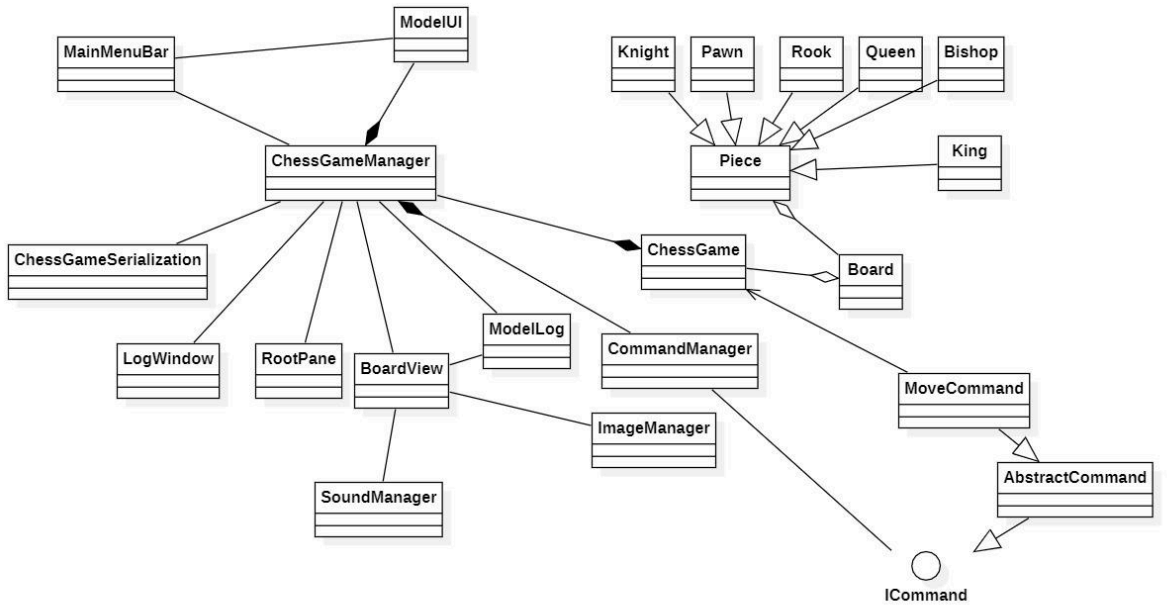
Usamos o padrão Singleton em ModelLog, SoundManager e ImageManager para garantir que apenas existe uma única instância e de acesso global controlado.
Por fim usamos o padrão Command, para as funcionalidades de undo e redo, MoveCommand fica responsável por encapsular uma jogada de xadrez, CommandManager por manter uma “stack” dos comandos e ChessGameManager que mais uma vez atua como facade chamando ou revertendo os comandos.

3. Descrição das principais classes

ChessGame	Contém a lógica principal do jogo de xadrez, incluindo regras como xeque, roque, promoção.
Board	Representa o tabuleiro, guarda e gere a colocação e movimentação das peças.
Piece (abstract)	Classe base para todas as peças de xadrez. Define atributos comuns e movimentos.
Bishop, Knight, etc.	Subclasses de Piece que implementam os movimentos específicos de cada peça.
PieceFactory	Aplica o padrão Factory para criar peças a partir de texto (ex: importação CSV).
ChessGameManager	Enum que define os tipos de peça (PAWN, KING, QUEEN, etc).
ModelLog	Singleton que armazena o histórico de eventos do jogo (log).
GameState	Enum que representa o estado atual do jogo: ONGOING, CHECK, CHECKMATE, STALEMATE.
Winner	Enum que indica o vencedor do jogo ou empate.
MoveCommand	Implementa o padrão Command. Representa uma jogada (inclui suporte a roque, promoção, etc).
CommandManager	Gere o histórico de comandos para permitir undo/redo.
ICommand	Interface para comandos com métodos execute() e undo().
AbstractCommand	Classe abstrata base para comandos.
ChessGameSerialization	Lida com a serialização e desserialização do estado do jogo para ficheiros .dat.

BoardView	Lida com a serialização e desserialização do estado do jogo para ficheiros .dat.
RootPane	Layout principal da aplicação, organiza a disposição da interface.
MainMenuBar	Barra de menus com opções para novo jogo, salvar, carregar, etc.
LogWindow	Janela com o histórico de eventos do jogo (ligado ao ModelLog).
MainJFX	Classe principal que inicia a aplicação JavaFX.
ImageManager	Carrega e gere imagens das peças do xadrez.
SoundManager	Toca ficheiros de áudio para movimentos, capturas, etc.
ModelUI	Gere o estado do som e do modo de aprendizagem da interface.

4. Relação entre classes (UML)



5. Funcionalidades implementadas

Todas as funcionalidades do trabalho foram implementadas exceto a funcionalidade extra (opcional) - o Board editor.