|  |  |
| --- | --- |
|  | Jogo de xadrez |
|  |  |
|  | Francisco Andrade Carvalho – 2019129635 Afonso Andrade – 2021142937  Engenharia Informática Isec Deis 24/25  6/2/25 |

Sistemas Operativos 2

# Contents

[Contents 1](#_Toc199777119)

[1. Introdução 2](#_Toc199777120)

[2. Arquitetura Geral do Projeto 2](#_Toc199777121)

[3. Decisões de Design e Justificações 2](#_Toc199777122)

[3.1 Separação de responsabilidades 2](#_Toc199777123)

[3.2 Regras encapsuladas 2](#_Toc199777124)

[3.3 Undo/Redo com Memento 2](#_Toc199777125)

[3.4 Promoção de peões 3](#_Toc199777126)

[4. Padrões de Design Utilizados 3](#_Toc199777127)

[5. Relações entre as Classes 4](#_Toc199777128)

[6. Funcionalidades Implementadas 4](#_Toc199777129)

[7. Conclusão 6](#_Toc199777130)

# Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Programação Avançada e tem como objetivo principal a implementação de um jogo completo de xadrez. A aplicação foi construída em Java, com interface gráfica baseada em JavaFX, e procura seguir as boas práticas de organização de código, incluindo a separação entre o modelo de dados e a interface de utilizador.

# Arquitetura Geral do Projeto

O projeto segue uma arquitetura modular, organizada segundo os princípios do padrão **MVC@PA**, com separação clara entre modelo, interface gráfica e controlador. O núcleo da lógica de jogo encontra-se encapsulado nas classes do modelo, enquanto a interface gráfica é construída com JavaFX, comunicando com o modelo através da classe *ChessGameManager*, que funciona como Facade observável.

# Decisões de Design e Justificações

## Separação de responsabilidades

Desde o início, decidimos dividir o projeto de forma clara entre lógica de jogo e interface. A classe ChessGame ficou responsável por toda a lógica, desde regras e turnos até deteção de fim de jogo. A classe Board funciona como o estado interno do tabuleiro, e foi pensada para ser o mais independente possível, para facilitar testes e alterações no futuro.

Para ligar o modelo à interface, criámos a classe ChessGameManager, que serve como ponte entre os dois mundos. Esta classe usa PropertyChangeSupport para notificar automaticamente a UI sobre mudanças no estado do jogo, sem depender diretamente dela.

## Regras encapsuladas

A validação das jogadas está toda concentrada no método ChessGame.move(...). Não quisemos que a interface tivesse de verificar regras como se o rei está em xeque ou se o movimento é legal. Isso está tudo do lado do modelo, como deve ser. Assim, a UI só precisa de pedir uma jogada, e o modelo trata de dizer se é válida ou não.

## Undo/Redo com Memento

Para implementar o undo/redo, optámos pelo padrão Memento. Criámos a classe ChessGameCaretaker, que guarda cópias do estado completo do jogo antes de cada jogada. Desta forma, é possível voltar atrás ou refazer movimentos sem complicar a lógica principal. Isto também encaixa bem com o facto de ChessGame ser serializável.

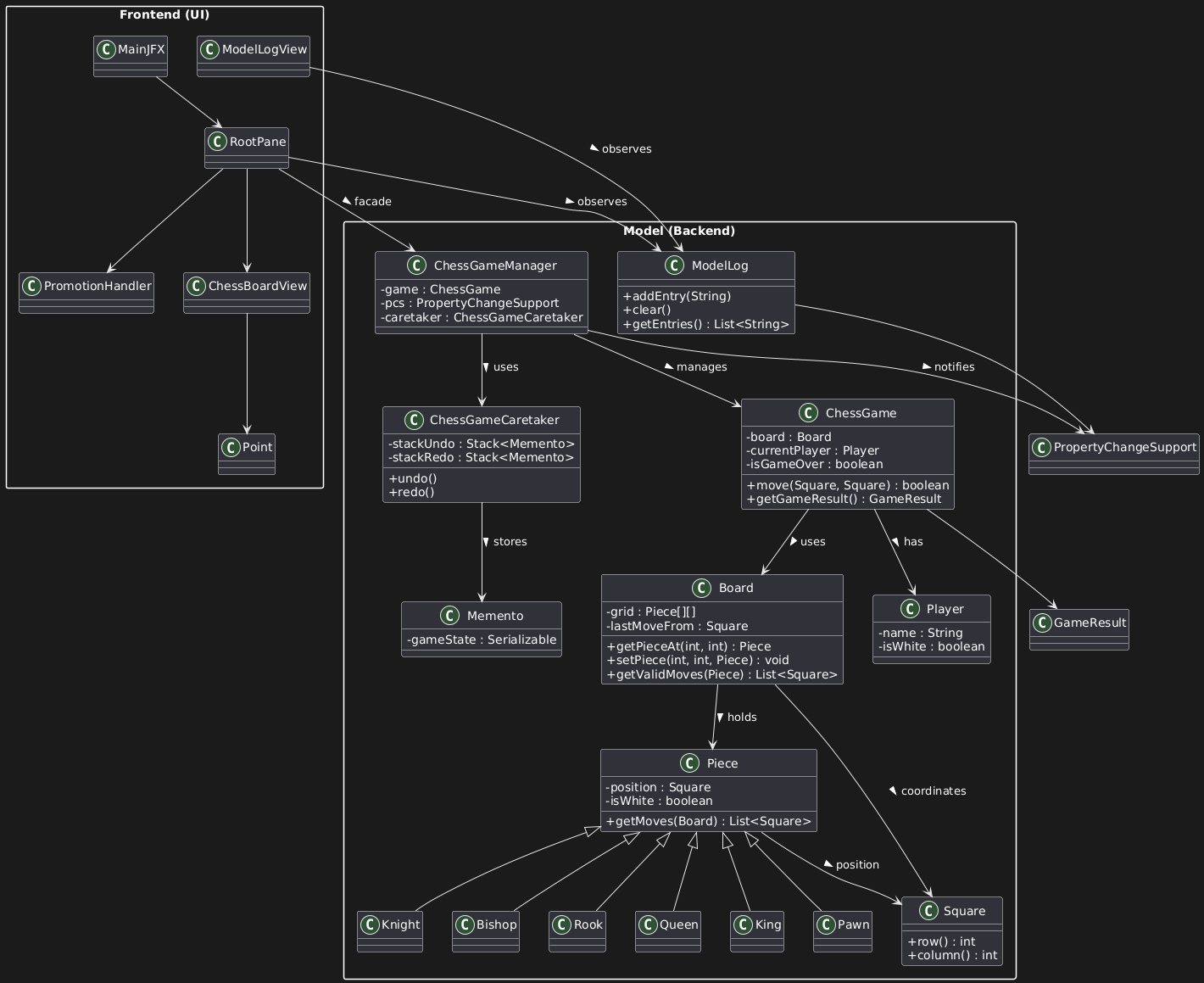
## Promoção de peões

A promoção foi feita de forma flexível usando uma interface chamada PromotionHandler. Esta interface é injetada no Board quando o jogo começa. Isto permite que o código da lógica continue separado da UI, mesmo quando é preciso perguntar ao utilizador que peça quer escolher na promoção.

# Padrões de Design Utilizados

| **Padrão** | **Onde foi usado** | **Função** |
| --- | --- | --- |
| Memento | ChessGame, ChessGameCaretaker | Undo/redo seguro do estado completo |
| Factory | PieceFactoryType.createPiece(...) | Criação de peças dinamicamente a partir de char ou tipo |
| Facade | ChessGame | Centraliza e expõe as operações do modelo |
| Observer (via PropertyChange) | ChessGameManager | Comunicação com a interface |

# Relações entre as Classes



# Funcionalidades Implementadas

|  |  |  |
| --- | --- | --- |
| Funcionalidade | Estado | Observações |
| Representação do tabuleiro com armazenamento das peças | Implementada | N/A |
| Representação das peças com subclasses específicas | Implementada | N/A |
| Representação textual das peças e estado inicial | Implementada | N/A |
| Padrão Factory de peças | Implementada | N/A |
| Classe ChessGame como fachada da lógica de jogo | Implementada | N/A |
| Classe ChessGameManager como fachada observável | Implementada | N/A |
| Salvar e carregar jogos com serialização | Implementada | N/A |
| Importar e exportar jogos parciais (texto) | Implementada | N/A |
| Integração com JavaFX e separação por MVC@PA | Implementada | N/A |
| Menu com opções de novo jogo, guardar, carregar, importar, exportar | Implementada | N/A |
| Input de nomes dos jogadores | Implementada | N/A |
| Desenho do tabuleiro com labels e dimensões dinâmicas | Implementada | N/A |
| Desenho das peças com imagens centradas | Implementada | N/A |
| Movimento de peças via clique (sem input textual) | Implementada | N/A |
| Validação de jogadas via modelo (sem trapaça) | Implementada | N/A |
| Detecção de fim de jogo: xeque-mate e empate | Implementada | N/A |
| Jogadas especiais: roque, promoção, en passant | Implementada | N/A |
| Modo de aprendizagem: mostrar jogadas possíveis | Implementada | N/A |
| Undo / Redo com padrão de design (Memento) | Implementada | N/A |
| Feedback áudio descritivo da jogada | Implementada | N/A |
| Toggle de som através da UI | Implementada | N/A |
| Logs de eventos e erros com ModelLog (Singleton + Observer) | Implementada | N/A |
| Suporte a múltiplas janelas (log + jogo) | Implementada | N/A |
| Atualizações automáticas via PropertyChangeListener | Implementada | N/A |
| Testes unitários variados (mín. 6) | Implementada | N/A |
| Documentação com Javadoc nas principais classes | Implementada | N/A |
| Funcionalidade extra: editor de tabuleiro | Não implementada |  |
|  |  |  |
|  |  |  |

# Conclusão

O desenvolvimento deste projeto foi uma oportunidade prática para consolidar conhecimentos teóricos adquiridos ao longo da unidade curricular, com especial foco em boas práticas de engenharia de software. A separação clara entre modelo, lógica de jogo e interface gráfica permitiu uma arquitetura robusta e facilmente extensível. A integração dos padrões Memento, Singleton e Observer demonstrou-se eficaz para funcionalidades como undo/redo, logging e reatividade da interface.

Para além da implementação das regras completas do jogo de xadrez, o projeto reforçou a importância de decisões como a injeção de dependências, encapsulamento de responsabilidades e teste de componentes individuais. A estrutura modular e testável garante que futuras alterações possam ser feitas de forma controlada e sem comprometer a estabilidade geral da aplicação.