Utilização de Programação Orientada a Objetos para o Desenvolvimento de uma Engine de Xadrez

1st Pedro Antonio de Souza Campos Rodrigues
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brasil
pedro.ufmg2@gmail.com

***Resumo*—Este trabalho descreve o desenvolvimento de uma engine de xadrez utilizando uma representação matricial, onde cada elemento é uma peça (*Piece*). O tabuleiro é modelado como uma matriz 8x8, facilitando a manipulação do estado do jogo e a implementação das regras do xadrez. A relação de herança é fundamental no design das peças, com a classe *Piece* servindo como base para peças como *Pawn*, *Rook*, *Knight*, *Bishop*, *Queen* e *King*. Cada classe derivada herda propriedades e métodos da classe base, permitindo a reutilização de código e a implementação de comportamentos específicos de cada peça. Esta abordagem garante um sistema modular e eficiente.**

***Index Terms*—Chess, Matrix Representation, Piece Inheritance, Chess Engine, Game Development, Graphical User Interface, User Account Management, Historical Game Data.**

I. Introdução

A. Contextualização Histórica

O xadrez é um dos jogos de estratégia mais antigos e reverenciados do mundo. Suas origens apontam ao século VI na Índia, onde era conhecido como ”chaturanga”. Este jogo se espalhou através da Europa e Ásia . Ao longo dos séculos, o xadrez evoluiu, adotando as regras que conhecemos hoje e se estabelecendo como um dos principais jogos de estratégia.

No século XX, com o desenvolvimento do computador, o xadrez encontrou um novo campo de expansão: a programação de computadores. Em 1950, o matemático e cientista da computação Claude Shannon publicou um artigo pioneiro sobre a programação de computadores para jogar xadrez, delineando os fundamentos dos algoritmos de busca que ainda são usados hoje. Este trabalho lançou as bases para o desenvolvimento de programas de xadrez e inteligência artificial .

Nos anos seguintes, diversos programas de xadrez foram desenvolvidos, com avanços significativos na capacidade de processamento e na sofisticação dos algoritmos. Em 1997, o supercomputador Deep Blue, desenvolvido pela IBM, fez história ao derrotar o então campeão mundial Garry Kasparov em uma partida de xadrez. Este evento marcou um ponto de virada na interação entre inteligência humana e artificial, demonstrando a capacidade dos computadores de realizar tarefas complexas de raciocínio e estratégia.

Richard Lang, um dos pioneiros no desenvolvimento de programas de xadrez, enfatizou a importância de um gerador de movimentos eficiente e preciso para o sucesso de qualquer motor de xadrez. Este trabalho é focado justamente no desenvolvimento de uma engine de xadrez robusta, que servirá como a base para futuros avanços e melhorias.

Os códigos utilizados neste trabalho estão disponíveis em um repositório do GitHub [1], que pode ser acessado através do seguinte link: TP-POO-XADREZ

B. Regras do Xadrez

Para o desenvolvimento de um jogo de xadrez, é essencial o entendimento detalhado de suas regras. Nesta subseção, serão discutidas as principais regras desse jogo.

No início de uma partida de xadrez, cada jogador controla dezesseis peças que podem ser claras ou escuras, onde as brancas devem fazer o primeiro movimento. Durante o transcorrer da partida, o objetivo de cada jogador é colocar o rei adversário em uma posição de risco iminente, conhecida como ”xeque”. Quando o rei de um dos enxadristas está sob ataque, é dito que ele está em xeque. Nesta situação, o jogador cujo rei está ameaçado deve fazer um movimento que o retire dessa condição. Se esse jogador não tem nenhuma opção de movimento que proteja o rei, ele está em ”xeque-mate”, e o adversário ganha a partida.

Além disso, cada peça possui uma lógica de movimento própria, desse modo pode-se fazer a seguinte subdivisão:

1) Peão: Como pode ser observado na figura 1, os peões movem-se uma casa para a frente em direção à linha final do tabuleiro do adversário. Na imagem, o peão branco em c4 pode mover-se para c5, e o peão branco em f2 pode mover-se para f3. Na sua primeira jogada, um peão pode avançar duas casas. Por exemplo, o peão branco em f2 pode avançar diretamente para f4.

Os peões capturam peças adversárias em movimento diagonal para a frente, uma casa. Na imagem, o peão branco em c4 pode capturar uma peça inimiga d5, movendo-se para d5. O peão branco em f2 pode capturar em e3, movendo-se para e3. Quando um peão alcança a linha final do tabuleiro do adversário (8ª linha para peões brancos, 1ª linha para peões pretos), ele é promovido a uma rainha.

2) Torre: Na Figura 2, a movimentação da torre é representada com marcações ”X”em posições onde a torre pode se mover ou capturar peças adversárias. A torre move-se em linha reta tanto na horizontal quanto na vertical por qualquer número de casas. Na imagem, a torre em d5 pode mover-se para qualquer casa na coluna ”d”(d1, d2, d3, d4, d6, d7, d8) ou na linha 5 (a5, b5, c5, e5, f5, g5, h5). A torre captura da mesma forma que se move, ocupando a casa onde a peça adversária

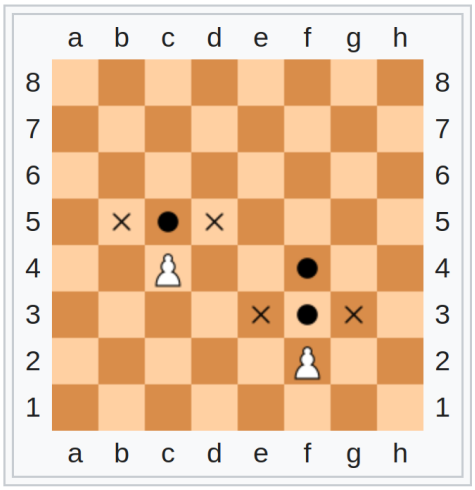

Figura 1. Movimentos do peão

está localizada. Porem, se tivessemos uma peça inimiga em f5, a torre não poderia mover-se para g5 ou h5.

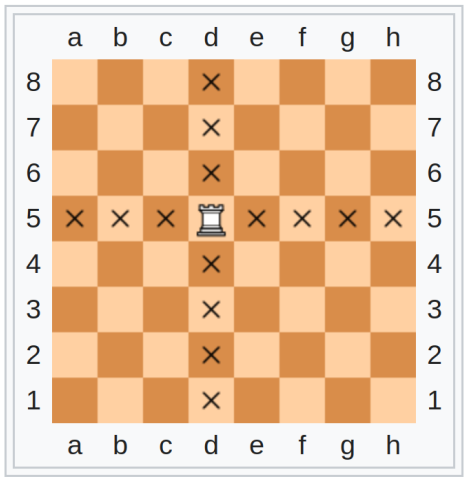

Figura 2. Movimentos da torre

3) Cavalo: Na Figura 3, a movimentação do cavalo é representada com marcações ”X”em posições onde o cavalo pode se mover ou capturar peças adversárias. O cavalo move-se em forma de ”L”, ou seja, ele pode se mover duas casas em uma direção (horizontal ou vertical) e depois uma casa em uma direção perpendicular, ou uma casa em uma direção e depois duas casas em uma direção perpendicular.

Na imagem, o cavalo em d5 pode mover-se para qualquer uma das seguintes casas: b6, b4, c7, c3, e7, e3, f6 e f4. O cavalo captura peças adversárias da mesma forma que se move, ocupando a casa onde a peça adversária está localizada. Diferente da torre, o movimento do cavalo pode ”pular”sobre outras peças, não sendo impedido por peças que estejam entre sua posição inicial e a posição final.

Por exemplo, se houvesse uma peça inimiga em f4, o cavalo poderia se mover para f4 e capturá-la, mas a presença de outras peças entre d5 e f4 não afetaria esse moviment

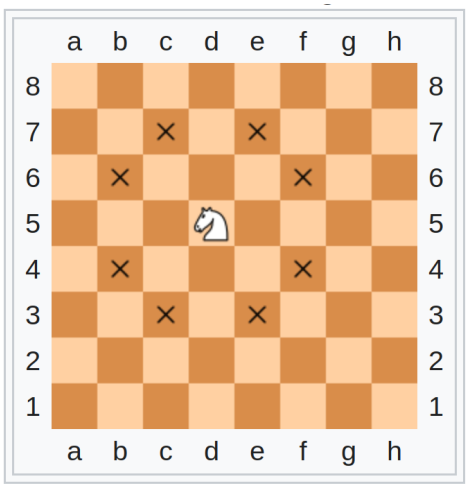

Figura 3. Movimentos da cavalo

4) Bispo: Na Figura 4, a movimentação do bispo é representada com marcações ”X”em posições onde o bispo pode se mover ou capturar peças adversárias. O bispo move-se em linhas retas na diagonal por qualquer número de casas, desde que não haja peças obstruindo seu caminho.

Na imagem, o bispo em d4 pode mover-se para qualquer casa nas diagonais que atravessam d4, ou seja, ele pode mover-se para as casas a1, b2, c3, e5, f6, g7, h8, a7, b6, c5, e3, f2 e g1. O bispo captura peças adversárias da mesma forma que se move, ocupando a casa onde a peça adversária está localizada. Se uma peça adversária estivesse em f6, por exemplo, o bispo poderia mover-se para f6 e capturá-la, mas não poderia se mover para além de f6, como para g7 ou h8, enquanto a peça adversária estiver em f6.

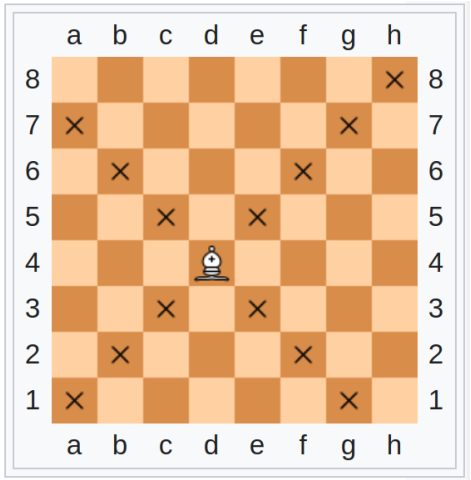

Figura 4. Movimentos do bispo

5) Rainha: Na Figura 5, a movimentação da rainha é representada com marcações ”X”em posições onde a rainha pode se mover ou capturar peças adversárias. A rainha combina os movimentos da torre e do bispo, movendo-se tanto em linhas

retas horizontais, verticais, quanto nas diagonais por qualquer número de casas.

Na imagem, a rainha em d4 pode mover-se para qualquer casa na coluna ”d”(d1, d2, d3, d5, d6, d7, d8), na linha 4 (a4, b4, c4, e4, f4, g4, h4) e nas diagonais que atravessam d4 (a1, b2, c3, e5, f6, g7, h8, a7, b6, c5, e3, f2, g1).

A rainha captura peças adversárias da mesma forma que se move, ocupando a casa onde a peça adversária está localizada. Por exemplo, se houvesse uma peça adversária em f6, a rainha poderia mover-se para f6 e capturá-la, mas não poderia mover-se além de f6, como para g7 ou h8, enquanto a peça adversária estiver em f6.

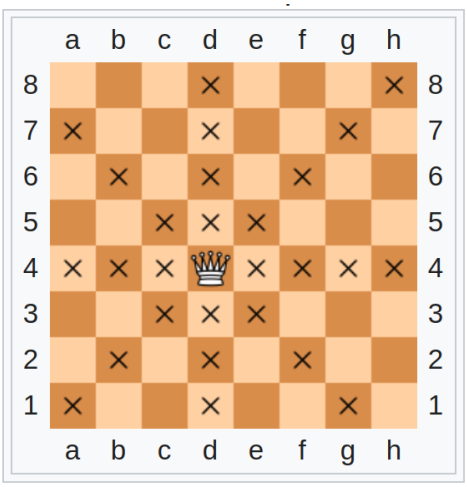

Figura 5. Movimentos da rainha

6) Rei: Na Figura 6, a movimentação do rei é representada com marcações ”X”em posições onde o rei pode se mover ou capturar peças adversárias. O rei move-se uma casa em qualquer direção: horizontalmente, verticalmente ou diagonalmente.

Se o rei estiver em f5, ele pode se mover para qualquer uma das casas adjacentes: e6, f6, g6, e5, g5, e4, f4, e g4. O rei captura peças adversárias da mesma forma que se move, ocupando a casa onde a peça adversária está localizada. Diferente de outras peças, o rei não pode se mover para uma casa que esteja sob ataque de uma peça adversária, pois isso o colocaria em xeque.

Por exemplo, se houvesse uma peça adversária em e6, o rei poderia mover-se para e6 e capturá-la, desde que a casa e6 não esteja sendo atacada por outra peça adversária.

C. Objetivos do Projeto

Este projeto tem como objetivo desenvolver um jogo de xadrez funcional, abordando os seguintes aspectos principais: Este projeto tem como objetivo desenvolver um jogo de xadrez funcional, abordando os seguintes aspectos principais:

- Implementar as principais regras e movimentos do xadrez, garantindo a fidelidade ao jogo tradicional.
- Desenvolver uma interface gráfica intuitiva e amigável, proporcionando uma experiência agradável ao usuário.

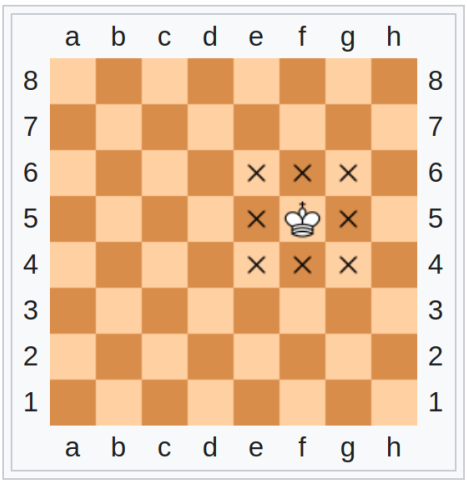

Figura 6. Movimentos do rei

- Adicionar funcionalidades de monitoramento de histórico de partida para acompanhar o progresso e os resultados das partidas.

D. Descrição Geral do Projeto

O projeto foi desenvolvido utilizando a linguagem de programação *Python*, junto com bibliotecas como *Pygame* e *Numpy*. As principais funcionalidades implementadas incluem:

- Movimentação das peças de acordo com as regras do xadrez.
- Detecção de situações especiais como xeque, xeque-mate e empate.
- Interface gráfica para a interação com o usuário.
- Funcionalidades de monitoramento de histórico de partidas.

II. Metodologia

A. Estrutura do projeto

O projeto de desenvolvimento de um sistema de xadrez foi organizado de forma a abranger todas as funcionalidades necessárias para simular um jogo de xadrez completo e interativo. Abaixo, detalhamos a estrutura do projeto, destacando as principais classes e suas responsabilidades.

O diagrama UML foi criado para ilustrar a estrutura do sistema e as interações entre as classes. No entanto, devido ao tamanho e complexidade do diagrama, ele foi omitido deste documento. O arquivo pdf deste diagrama está disponível em [10], que pode ser acessado através do seguinte link: Diagrama UML

1) Classe Piece: A classe *Piece* é uma classe abstrata que representa uma peça no tabuleiro. Ela contém os seguintes atributos e métodos:

Atributos: *Position*: representa a posição da peça no tabuleiro. *Color*: representa a cor da peça *Type*: tipo da peça (peão, cavalo, bispo, torre, rainha, rei). *numofmoves*: número de movimentos realizados pela peça.

Métodos: *getPosition()*: retorna a posição da peça. *attPosition(Position pos)*: atualiza a posição da peça. *getColor()*: retorna a cor da peça. *getType()*: retorna o tipo da peça. *IsValidMove(Position target)*: método abstrato que verifica se o movimento para a posição alvo é válido.

2) Classes Derivadas de Piece: As classes derivadas herdam da classe base Piece e implementam o método isValidMove de acordo com as regras específicas de cada tipo de peça. Por exemplo, o Bishop (Bispo) herda de Piece e implementa movimentos diagonais, enquanto o King (Rei) implementa movimentos de uma casa em qualquer direção. O Knight (Cavalo) realiza movimentos em "L", o Pawn (Peão) movimentos verticais, captura na diagonal e regras especiais como *en passant*. A Queen (Rainha) efetua movimentos horizontais, verticais e diagonais, o Rook (Torre) realiza movimentos horizontais e verticais, e a classe Empty representa casas vazias no tabuleiro, sem comportamento específico.

Sendo assim, pode-se dizer que a herança é fundamental neste projeto para garantir a reutilização de código e a extensibilidade do sistema de peças de xadrez. Ao definir uma classe base Piece que encapsula atributos comuns e métodos essenciais como getPosition, setColor, getType, e isValidMove (um método abstrato), cada tipo de peça pode compartilhar uma estrutura básica enquanto ainda possui comportamentos específicos.

O polimorfismo entra em jogo quando as classes derivadas como Bishop, King, Knight, etc., implementam o método abstrato isValidMove de acordo com as regras de movimento específicas de cada tipo de peça. Isso significa que, ao chamar isValidMove para diferentes objetos de peça, o comportamento será adaptado automaticamente conforme a implementação na classe concreta correspondente.

Essa abordagem não apenas simplifica a implementação do jogo de xadrez, mas também melhora a legibilidade e a manutenibilidade do código, garantindo que cada tipo de peça seja responsável apenas pela sua própria lógica de movimento. Dessa forma, o código se torna mais coeso e menos acoplado, facilitando tanto o desenvolvimento inicial quanto as futuras modificações e extensões do sistema de peças de xadrez.

3) Classe ChessBoard: A classe ChessBoard encapsula toda a lógica necessária para gerenciar um jogo de xadrez. Ela mantém o estado do tabuleiro, controla os movimentos das peças, verifica condições de jogo como xeque e xeque-mate, e registra o histórico de movimentos. Isso promove uma estrutura organizada e modular onde as regras do jogo são implementadas de maneira centralizada e eficiente.

Robert C. Martin, conhecido como Uncle Bob, enfatiza a importância da alta coesão e baixo acoplamento no design de software. Alta coesão garante que cada módulo (ou classe) tenha uma responsabilidade clara e única, alinhando-se com o princípio de responsabilidade única de Martin, onde "um módulo deve ser responsável por um, e apenas um, ator"([12]). No contexto da classe ChessBoard, este princípio guia a separação de responsabilidades entre o gerenciamento do estado do jogo e a implementação dos comportamentos das peças individuais como bispos, reis, entre outros.

Por outro lado, o baixo acoplamento minimiza as dependências entre módulos, permitindo que alterações em uma parte do código tenham um impacto mínimo em outras áreas. Isso é crucial para a ChessBoard, pois facilita a adição de novas funcionalidades sem perturbar o funcionamento existente ([12]).

Portanto, a ChessBoard não apenas representa eficazmente o estado do jogo, mas também fornece métodos que operam sobre este estado de acordo com as regras estabelecidas do xadrez tradicional. Esta abordagem de design está alinhada com os princípios de Uncle Bob, garantindo manutenibilidade, escalabilidade e robustez no código.

4) Classe GameManagement: A classe *GameManagement* é responsável por gerenciar usuários e partidas em um sistema através de métodos como carregar e salvar usuários em arquivos, gerar hashes de senhas, validar senhas, realizar login, criar novas contas de usuário, adicionar e imprimir o histórico de partidas de usuários específicos. Os dados são armazenados em formato JSON, que é leve, de fácil leitura e organiza informações em pares de chave e valor.

B. Interfaces utilizadas

No desenvolvimento deste projeto de xadrez, foram definidas diversas interfaces utilizando o módulo `abc` da biblioteca padrão do Python. As interfaces são uma parte fundamental para a modularidade e o acoplamento entre as classes, respeitando os princípios da Programação Orientada a Objetos (POO). Ao garantir que as classes implementem as interfaces definidas (o contrato), os detalhes de implementação se tornam irrelevantes, facilitando a manutenção e a extensibilidade do código.

A `ChessBoardInterface` define um conjunto de métodos abstratos que devem ser implementados por qualquer classe que represente um tabuleiro de xadrez. Isso inclui métodos para mover peças, verificar a validade de um movimento, obter todos os movimentos válidos e obter os movimentos possíveis a partir de uma posição específica.

A `ChessRenderInterface` define métodos para a renderização do tabuleiro e para a interação com o usuário. Isso inclui métodos para obter e atualizar a posição da peça selecionada, renderizar o tabuleiro, lidar com eventos do usuário e controlar o estado de fechamento da janela gráfica.

A `PieceInterface` define métodos que devem ser implementados por qualquer classe que represente uma peça de xadrez. Isso inclui métodos para atualizar a posição da peça, obter a cor e o tipo da peça, verificar se um movimento é válido e obter a posição atual da peça.

Essas interfaces garantem que as diferentes partes do sistema de xadrez possam interagir de maneira coesa e flexível. Ao definir contratos claros, as interfaces permitem que as implementações específicas sejam modificadas ou substituídas sem afetar outras partes do sistema, desde que o contrato seja respeitado. Isso promove um design orientado a objetos robusto, facilitando a manutenção e a evolução do projeto ao longo do tempo.

C. Estrutura da Função main

A função `main` é o ponto central do programa e coordena a interação com o usuário. Inicialmente, ela cria uma instância de `GameManagement`, responsável pelo gerenciamento de usuários e partidas. Em seguida, a função entra em um loop `while` que exibe continuamente o menu principal até que o usuário decida fechar o jogo. O menu principal oferece quatro opções: iniciar uma partida, acessar uma conta existente, criar uma nova conta e fechar o jogo.

Se o usuário optar por iniciar uma partida, a função `startGame` é chamada. Esta função solicita que os jogadores façam login utilizando a função `loginUserForGame`, uma vez para as peças brancas e outra para as pretas. Após o login dos jogadores, a função `game` é executada, gerenciando o jogo de xadrez, incluindo a renderização do tabuleiro, o processamento de movimentos e a verificação de condições de vitória ou empate. O resultado da partida é então registrado no histórico dos jogadores através da função `addGame` do `GameManagement`.

Caso o usuário escolha acessar uma conta existente, a função `accessAccount` é chamada. Esta função solicita o nome de usuário e a senha, e se as credenciais estiverem corretas, o usuário pode acessar o menu da conta (`accountMenu`), onde pode ver o histórico de partidas ou sair da conta. Se as informações estiverem incorretas, o usuário pode optar por tentar novamente ou retornar ao menu principal.

Se o usuário desejar criar uma nova conta, a função `createAccount` é chamada. Ela solicita um nome de usuário e uma senha, e tenta criar a conta através do método `addUser` do `GameManagement`. Se o nome de usuário já existir, o usuário é informado e pode optar por tentar novamente ou voltar ao menu principal.

O loop principal continua executando até que o usuário escolha a opção de fechar o jogo, momento em que o programa é encerrado. Essa estrutura garante uma navegação intuitiva e uma gestão eficiente de usuários e partidas, proporcionando uma experiência organizada e coerente para o usuário.

D. Interface Gráfica

A classe *ChessRender* é responsável pela renderização do tabuleiro e pela interface gráfica do jogo. Utilizando a biblioteca *pygame*, ela encapsula todos os detalhes gráficos. Isso facilita o seu uso, pois proporciona mais uma camada de abstração, permitindo que os desenvolvedores se concentrem na lógica do jogo sem se preocupar com os detalhes da renderização gráfica. Ademais, *ChessRender* promove alta coesão ao concentrar todas as responsabilidades da interface gráfica em uma única classe, seguindo o princípio de responsabilidade única e delegando à *ChessBoard* a lógica do jogo.

Atributos:

- *shouldclose*: indica se a janela deve ser fechada.
- *board*: referência ao tabuleiro de xadrez (*ChessBoard*).
- *screen*: tela para renderização.
- *SelectedPiece*: posição da peça selecionada.
- *possibleDest*: lista de posições de destino possíveis.

Métodos:

- *getSelectedPiecePos()*: retorna a posição da peça selecionada.
- *updateSelectedPiece()*: atualiza a peça selecionada.
- *quit()*: encerra a renderização.
- *setShouldclose()*: define que a janela deve ser fechada.
- *getShouldclose()*: verifica se a janela deve ser fechada.
- *render()*: renderiza o tabuleiro e as peças.
- *handle_events()*: lida com os eventos do usuário.

Além disso, a presença do atributo *board* na *ChessRender* permite que as atualizações no tabuleiro de xadrez ocorram automaticamente na interface gráfica. Isso acontece porque a *ChessRender* mantém uma referência direta ao objeto *ChessBoard*, que encapsula toda a lógica do jogo e o estado atual do tabuleiro. Sempre que o estado do *ChessBoard* é alterado, como quando uma peça é movida, a *ChessRender* pode acessar imediatamente essas mudanças através da referência ao *board* e atualizar a interface gráfica automaticamente.

E. Desenvolvimento da Lógica do Jogo

Para a representação do tabuleiro, foi escolhido um tabuleiro matricial 8x8, onde cada elemento do tabuleiro é uma instância da classe `Piece`. Esta abordagem permite uma fácil manipulação e acesso às peças durante o jogo, garantindo que as regras de movimentação e as verificações de estado possam ser aplicadas possam ser implementadas mais facilmente, de forma mais intuitiva.

Para verificar se um movimento é válido (sem considerar o xeque), aplicamos regras específicas para cada tipo de peça. Usando o bispo como exemplo, a classe `Bishop` implementa a lógica do movimento diagonal, onde a função `IsValidMove` verifica se a posição de destino está em uma diagonal em relação à posição atual. A função `IsValidMove` sobrescreve o método abstrato da classe `Piece`. Abaixo está o pseudocódigo para esta função:

Algorithm 1 IsValidMove - Bishop

```
0: function IsValidMove(to)
0:     drow ← |to.row − current_position.row|
0:     dcol ← |to.col − current_position.col|
0:     return (drow == dcol)
0: end function=0
```

A função IsValidMove para o bispo no xadrez verifica se um movimento é diagonal ao calcular a diferença absoluta entre as linhas e colunas da posição de destino em relação à posição atual. Se essa diferença for igual tanto nas linhas quanto nas colunas, o movimento é considerado válido, pois indica uma movimentação diagonal

Além de verificar se o movimento é válido para a peça, também precisamos garantir que o caminho até a posição de destino esteja livre. A função `__isPossibleMove` realiza essa verificação. Abaixo está o pseudocódigo para esta função, com partes não essenciais para o bispo omitidas:

A lógica para outras peças é semelhante, onde cada peça tem sua própria implementação da função `IsValidMove` que

Algorithm 2 isPossibleMove

```
0: function ISPOSSIBLEMOVE(from, to)
0:     is_clear ← isPathClear(from, to)
0:     valid ← piece_at(from).IsValidMove(to)
0:     return (valid and is_clear)
0: end function=0
```

verifica se o movimento proposto está de acordo com as regras específicas daquela peça.

Para gerar todos os movimentos possíveis de um jogador, utilizamos o método `getAllMoves()`, que percorre todo o tabuleiro e gera todos os movimentos pseudolegais de um dado jogador. Um movimento pseudolegal é um movimento que respeita as regras básicas de movimentação da peça, mas não necessariamente considera situações especiais como o xeque.

O pseudocódigo simplificado para o método `getAllMoves()` é apresentado a seguir:

Algorithm 3 getAllMoves

```
0: function GETALLMOVES
0:     moves ← []
0:     for each position pos on the board do
0:         if piece_at(pos) belongs to the current player then
0:             for each position to on the board do
0:                 if isPossibleMove(pos, to) then
0:                     moves.append((pos, to))
0:                 end if
0:             end for
0:         end if
0:     end for
0:     return moves
0: end function=0
```

O método `getAllMoves()` percorre todas as posições do tabuleiro, verifica quais peças pertencem ao jogador atual e, para cada peça, gera todos os movimentos possíveis que são considerados pseudolegais. Esses movimentos são coletados em uma lista e retornados para posterior análise.

Na realidade, a implementação do método `getAllMoves` é mais complexa, pois busca uma maior eficiência e conta com vários métodos auxiliares e otimizações. Esses métodos auxiliares ajudam a reduzir a quantidade de verificações necessárias, e lidam com regras mais específicas do jogo.

Após a geração dos movimentos pseudolegais pelo método `getAllMoves`, a função `getValidMoves` é utilizada para filtrar esses movimentos e garantir que apenas os movimentos que não deixam o rei em xeque sejam considerados válidos. No algoritmo 4 tem-se um pseudocódigo simplificado.

O método `getValidMoves` é responsável por determinar quais movimentos são permitidos para o jogador ativo em uma partida de xadrez. Primeiramente, ele utiliza o método `getAllMoves` para gerar todos os possíveis movimentos pseudolegais para o jogador atual. Em seguida, para cada movimento gerado, temporariamente aplica o movimento utilizando `makeMove`, verifica se isso deixaria o rei do jogador em xeque alternando o turno temporariamente e utilizando `inCheck()`. Se um movimento coloca o rei em xeque, ele é removido da lista de movimentos válidos. Após a filtragem, o método verifica se não há movimentos válidos, marcando `checkMate` se o jogador estiver em xeque ou `staleMate` se não houver movimentos possíveis sem deixar o rei em xeque.

A solução apresentada para filtrar os movimentos válidos no xadrez, utilizando o método `getValidMoves`, é mais custosa em termos de desempenho por várias razões. Pois, o método `getAllMoves` precisa gerar todos os movimentos possíveis várias vezes ao longo do processo. Esta geração repetitiva de movimentos pseudolegais adiciona uma carga computacional significativa.

Apesar do custo computacional mais elevado, essa abordagem oferece duas vantagens importantes: confiabilidade e facilidade de implementação. A verificação individual de cada movimento garante que todos os movimentos considerados válidos realmente não deixam o rei em xeque. Isso proporciona uma segurança adicional ao jogo, assegurando a conformidade com as regras do xadrez. Além disso, a implementação é mais direta e intuitiva, pois lidamos com cada movimento individualmente.

Algorithm 4 getValidMoves

```
0: function GETVALIDMOVES
0:     moves ← getAllMoves()
0:     for each move in moves do
0:         makeMove(move)
0:         toggleTurn()
0:         if inCheck() then
0:             moves.remove(move)
0:         end if
0:         toggleTurn()
0:         undoMove()
0:     end for
0:     if moves is empty then
0:         if inCheck() then
0:             checkMate ← true
0:         else
0:             staleMate ← true
0:         end if
0:     else
0:         checkMate ← false
0:         staleMate ← false
0:     end if
0:     moves.extend(getCastleMoves())
0:     return moves
0: end function=0
```

III. RESULTADOS

O desenvolvimento do projeto de xadrez resultou em um jogo funcional, com interface gráfica e suporte a histórico de partidas. A implementação seguiu princípios de Programação

Orientada a Objetos (POO), utilizando interfaces para garantir a modularidade e a facilidade de manutenção do código.

A interface gráfica do jogo permite aos jogadores interagirem com o tabuleiro de xadrez de maneira intuitiva. A Figura 7 mostra o estado inicial do tabuleiro de xadrez, com todas as peças posicionadas corretamente. Durante o jogo, os movimentos válidos para cada peça são destacados em vermelho, como ilustrado na Figura 8. Isso ajuda os jogadores a visualizarem suas opções e tomarem decisões informadas durante a partida.

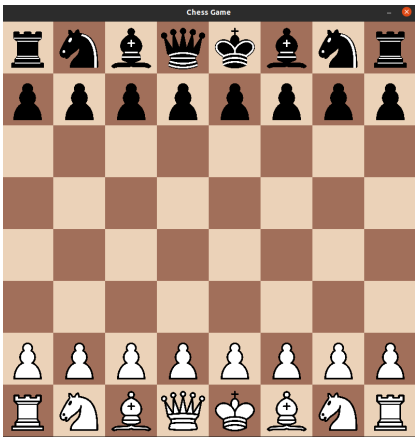

Figura 7. Estado inicial do tabuleiro de xadrez.

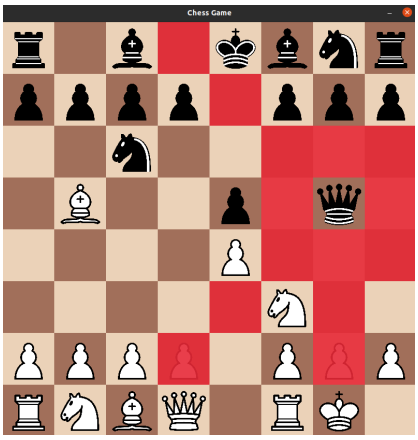

Figura 8. Movimentos válidos destacados em vermelho.

Além da interface gráfica, o projeto inclui funcionalidades para gerenciamento de contas de usuário e histórico de partidas. Os jogadores podem criar contas, acessar suas contas existentes e visualizar o histórico de suas partidas anteriores. Essa funcionalidade é gerenciada pela classe `GameManagement`, que mantém um registro das partidas jogadas e das contas dos usuários.

A Figura 9 apresenta um exemplo do monitoramento do histórico de partidas, onde é possível visualizar o nome dos oponentes e os resultados das partidas. A interface de gerenciamento de contas permite que os jogadores acessem suas contas e verifiquem seus históricos de forma fácil e rápida.

```
------------------------------------------------
         Bem-vindo ao jogo de xadrez!
         Você está no menu principal.
 O que deseja fazer?
------------------------------------------------
Iniciar uma partida com histórico [0]
Acessar minha conta [1]
Criar conta [2]
Fechar jogo [3]
------------------------------------------------
Digite sua opção: 1
------------------------------------------------
------------------------------------------------
Acessando a conta
------------------------------------------------
Digite o nome da sua conta: Pedro
Digite sua senha: 1313
------------------------------------------------
------------------------------------------------
        Seja bem-vindo Pedro!
O que você gostaria de fazer?
------------------------------------------------
Ver histórico [0]
Sair da conta [1]
Digite sua opção: 0
------------------------------------------------
         Histórico:
------------------------------------------------
Oponente: Jose
Resultado: vitoria

Oponente: Heitor
Resultado: derrota

Oponente: Vitoria
Resultado: derrota

Oponente: Henrique
Resultado: vitoria

Oponente: João
Resultado: vitoria

Oponente: Jose
Resultado: vitoria

Oponente: Luana
Resultado: vitoria

------------------------------------------------
```

Figura 9. Exemplo do monitoramento do histórico de partidas.

O uso de interfaces como `ChessBoardInterface`, `ChessRenderInterface` e `PieceInterface` foi crucial para a separação de preocupações e a manutenção da coesão do sistema. As interfaces permitiram que diferentes componentes do sistema fossem desenvolvidos e testados de forma independente, enquanto ainda garantiam que todas as partes do sistema pudessem interagir de maneira coesa.

IV. Conclusões

O projeto de desenvolvimento do jogo de xadrez foi concluído com sucesso, resultando em uma aplicação funcional e robusta. A interface gráfica desenvolvida permite aos jogadores interagirem facilmente com o tabuleiro de xadrez, oferecendo uma experiência de uso intuitiva. A implementação de destaques para movimentos válidos contribuiu significativamente para a experiência do usuário, auxiliando na tomada de decisões durante as partidas.

Além disso, a funcionalidade de criação e gerenciamento de contas foi bem-sucedida, permitindo que os jogadores mantenham um registro de suas partidas e acessem seus históricos de maneira prática. A visualização do histórico de partidas fornece uma forma eficaz para que os jogadores revisem suas performances e resultados anteriores.

Para futuras versões do projeto, consideramos a adoção de métodos mais avançados e eficientes para melhorar o desempenho e a precisão do jogo. Um exemplo disso é a utilização do método 0x88 para representar o tabuleiro de

xadrez. Conforme detalhado em [14], o método 0x88 oferece uma maneira mais eficiente de indexar e validar movimentos no tabuleiro, reduzindo a complexidade computacional e facilitando a detecção de limites e movimentos válidos.

Além disso, para melhorar a filtragem dos movimentos pseudolegais e lidar com situações de xeque e xeque-mate de forma mais eficiente, propomos trabalhar com peças travadas (pin pieces) e verificações diretas de xeque ao invés do método atual. Este ajuste permitirá uma avaliação mais precisa das ameaças no tabuleiro, melhorando a resposta do sistema em cenários críticos e aumentando a eficiência dos cálculos de movimentos legais.

Em resumo, enquanto o projeto atual oferece uma base sólida e funcional, as sugestões de melhorias propostas, como a adoção do método 0x88 e a otimização do tratamento de xeques e movimentos legais, visam aperfeiçoar ainda mais a eficiência do sistema em relação a situações adversas.

REFERÊNCIAS

[1] P. A. S. C. Rodrigues, "Código do Trabalho: TP-POO-XADREZ,"GitHub repository, 2024. [Online]. Available: https://github.com/pedro55562/TP-POO-XADREZ

[2] P. A. S. C. Rodrigues and Í. N. P. Vieira, GitHub repository, 2024. [Online]. Available: https://github.com/pedro55562/TP-PDS2-XADREZ

[3] S. Lague, "Chess Coding Adventure: Chess-V1-Unity,"GitHub repository, 2024. [Online]. Available: https://github.com/SebLague/Chess-Coding-Adventure/tree/Chess-V1-Unity

[4] R. Hyatt, "Chess program board representations,"Archived from the original on 12 February 2013. Retrieved 15 January 2012. [Online]. Available: https://web.archive.org/web/20130212063528/http://www.cis.uab.edu/hyatt/boardrep.html

[5] "Board Representation,"Chess Programming Wiki. [Online]. Available: https://www.chessprogramming.org/Board_Representation. [Accessed: 11-Jun-2024].

[6] "FEN (Forsyth-Edwards Notation),"chess.com. [Online]. Available: https://www.chess.com/terms/fen-chess.

[7] "Move Generation: Pseudo-legal moves,"Chess Programming Wiki. [Online]. Available: https://www.chessprogramming.org/Move_Generation#Pseudo-legal. [Accessed: 11-Jun-2024].

[8] "Legal Move,"Chess Programming Wiki. [Online]. Available: https://www.chessprogramming.org/Legal_Move. [Accessed: 11-Jun-2024].

[9] "Chess,"Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Chess#Movement. [Accessed: 13-Jun-2024].

[10] "Chess UML Diagram,"GitHub. [Online]. Available: https://github.com/pedro55562/TP-POO-XADREZ/blob/main/Chess_UML.pdf. [Accessed: 19-Jun-2024].

[11] D. Levy and M. Newborn, "How Computers Play Chess,"2nd ed. Computer Science Press, 1991.

[12] M. Heusser, "The fundamentals of achieving high cohesion and low coupling,"TechTarget, 21 de Março de 2023. Disponível em: https://www.techtarget.com/searchapparchitecture/tip/The-fundamentals-of-achieving-high-cohesion-and-low-coupling. [Accessed: 16-Jun-2024]

[13] R. C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship,"Prentice Hall, 2008.

[14] Chess Programming Wiki. (n.d.). 0x88. Recuperado de https://www.chessprogramming.org/0x88