Universidade Federal de São Carlos
Programação Orientada a Objetos

Projeto Xadrez

Alunos: Jesimiel Efraim Dias, Jhonata Campos Santana
Profª: Katti Facelli

Universidade Federal de São Carlos
Programação Orientada a Objetos

Projeto Xadrez

Projeto que simula um jogo de xadrez de acordo com as regras e exigências da orientadora do projeto;

Aluno: Jesimiel Efraim Dias

Jhonata Campos Santana

Prof[a]: Katti Facelli

06
2019

Sumário

1 Resumo

O projeto em questão busca simular um jogo de xadrez com todas as implementações feitas em C++. O projeto teve duas fases, cada uma com o objetivo de estruturar a construção do que foi pedido, e apresentar resultados satisfatórios. Neste relatório, apresentaremos a última fase do projeto, assim como seus resultados.

2 Especificações do projeto

O projeto a princípio não buscava ter a necessidade de ser funcional, e sim apenas estruturar o movimento das peças, e obter uma hierarquia de classes proporcionada pela orientadora do projeto, como exemplificado na imagem.

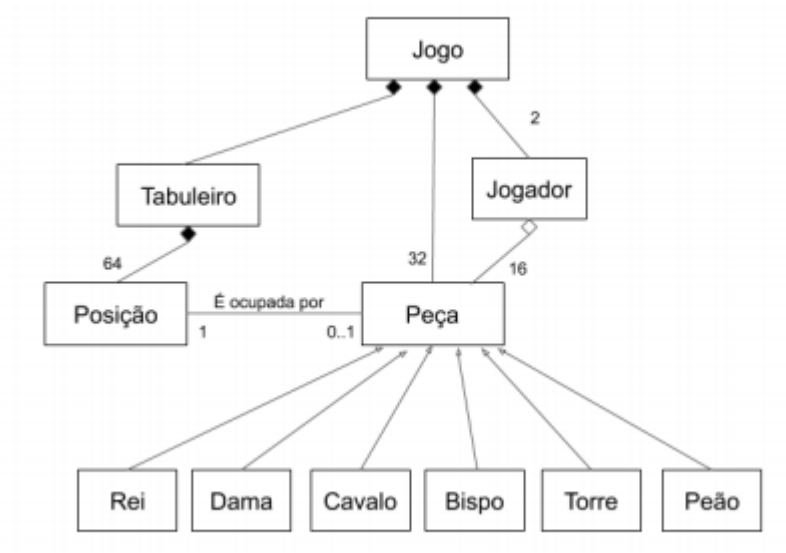

Figura 1: Diagrama de classes e suas relações

Foram especificadas regras e exigências para a elaboração do projeto, como:

- Proibido o uso de *namespaces*, como *"using namespace std"*, por exemplo;
- Proibido o uso de *"malloc, calloc e alloc"* para a criação de novas peças;

- Obrigatório o uso de *"Try Catch"* para qualquer espaço onde fosse necessário a manipulação de endereços;
- O uso de "C++"somente para a elaboração do projeto;
- Proibido o uso de *"Define"*;
- Proibido o uso de variáveis globais;

A segunda fase exigiu que todo o projeto fosse finalizado seguindo as regras do Xadrez, disponíveis em *https://pt.wikibooks.org/wiki/Xadrez/Regras*. O projeto exigiu conhecimentos profundos sobre conceitos de Programação Orientada a Objetos, como o uso de virtuais, hierarquia entre classes, herança, tratamento de erros, entre outros. O projeto todo foi realizado por volta de dois meses, e os resultados serão mostrados a seguir.

3 Descrição do Projeto - Classes e Métodos

Para a explicação simplificada de como o projeto opera, será feita uma subdivisão entre as classes, sendo explicitado como cada um das mesmas funcionam:

3.1 Peças Específicas

As peças específicas, como mostra a figura abaixo, não tem conhecimento entre as outras peças e nem o tabuleiro, então elas foram feitas de forma que todas tem apenas suas próprias características salvas:

Figura 2: Peças Específicas

Como cada classe tem seus atributos próprios, e elas não possuem conhecimento entre si, não é possível criar algo genérico, assim, cada classe terá sua particularidade.

3.1.1 Rei

O Rei foi construído da seguinte forma:

```
#ifndef REI_H
#define REI_H_

#include <iostream>
#include "Peca.h"

class Rei:public Peca{
    private:
        bool roque;
    public:
        Rei(bool color);//Iniciamos a peÃ§a com a respectiva cor.
        int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n);
        bool checaRoque(bool rock);

};
#endif
```

Figura 3: Rei.h

Quanto a seus métodos e atributos:

- bool roque: Um booleano que indica se o rei será capaz de executar o movimento de roque;
- Rei(bool color): O construtor da classe; sobrecarregado com o atributo de cor para definir o jogador;
- int** checaMovimento(): Método para verificar se o movimento do Rei é legal ou não. Suas sobrecargas serão explicadas mais a frente;
- bool checaRoque(bool rock): Verifica o movimento do roque;

Agora, quanto a sua implementação:

- Rei::Rei

```
#include "Rei.h"


//Construtor que inicializa o atributo cor e o atributo capturada para falso.
Rei::Rei(bool color):Peca(color,'r'){
    roque = true;
}
```

Figura 4: Rei.cpp - Rei::Rei

O objeto é inicializado com seu atributo "roque"setado para "1", dizendo que o roque é possível até que seu estado seja mudado. Ele também é sobrecarregado com "Peca(color,'r') - Explicado mais a frente - dizendo que o objeto é de fato, um rei.

- Rei::checaMovimento

```
int** Rei::checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n){
    //Existem 8 possíveis posições que o rei pode andar.
    if(((linhaOrigem+1 != linhaDestino) && (linhaOrigem-1 != linhaDestino) && (linhaOrigem != linhaDestino)) ||
        ((colunaOrigem+1 != colunaDestino) && (colunaOrigem-1 != colunaDestino) && (colunaOrigem != colunaDestino))){
        return 0;
    }

    int **posicoes = alocarMatrizPosicoes(n = 2);


    posicoes[0][0] = linhaOrigem;
    posicoes[0][1] = colunaOrigem;
    posicoes[1][0] = linhaDestino;
    posicoes[1][1] = colunaDestino;

    return posicoes;
}
```

Figura 5: Rei.cpp - Rei::checaMovimento

Este método verifica todos os movimentos possíveis do Rei baseados nos parâmetros do método, que são int linhaOrigem, int colunaOrigem, int linhaDestino e int colunaDestino; Estes atributos são responsáveis por apontar a origem e o destino do Rei. Se por acaso alguma dessas posições forem inválidas, o retorno é 0, e o movimento não será executado, do contrário, o método seta as novas posições na matriz, e retorna a matriz de posições;

- Rei::checaRoque

```
bool Rei::checaRoque(bool rock){
    return rock & roque;
}
```

Figura 6: Rei.cpp - Rei::checaRoque

Aqui, o método usa o parâmetro "rock"e faz uma comparação com o atributo da classe "roque", e retorna o booleano para verificar se o roque é verdadeiro ou não;

3.1.2 Bispo

O bispo foi construído da seguinte forma:

```
#ifndef BISPO_H_
#define BISPO_H_

#include <iostream>
#include "Peca.h"

class Bispo:public Peca{
    public:
        Bispo(bool color);//Iniciamos a peÃ§a com a respectiva cor.
        int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n);
};

#endif
```

Figura 7: Bispo.h

- Bispo(bool color): Construtor da classe;
- int** checaMovimento(): Método para verificar se o movimento do bispo é legal ou não. Suas sobrecargas serão explicadas mais a frente;

Agora, quanto a sua implementação:

- Bispo::Bispo

```
//Construtor que inicializa o atributo cor e o atributo capturada para falso.
Bispo::Bispo(bool color):Peca(color,'b'){
}
```

Figura 8: Bispo.cpp - Bispo::Bispo

O objeto é sobrecarregado com "Peca(color,'b')- Explicado mais a frente - dizendo que o objeto é de fato, um bispo;

- Bispo::checaMovimento

```
int** Bispo::checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n){

    if(fabs(linhaOrigem - linhaDestino) != fabs(colunaOrigem - colunaDestino)) return 0;

    n = fabs(linhaOrigem - linhaDestino)+1;
    int sinalI;
    int sinalJ;
    sinalI = sinalJ = 1;
    int **posicoes = alocarMatrizPosicoes(n);

    //primeiro quadrante.
    if(linhaOrigem > linhaDestino && colunaOrigem < colunaDestino){
        sinalI  = -sinalI ;
    }
    //segundo quadrante.
    else if(linhaOrigem > linhaDestino && colunaOrigem > colunaDestino){
        sinalI =-sinalI ;
        sinalJ =-sinalJ;
    }
    //terceiro quadrante.
    else if(linhaOrigem < linhaDestino && colunaOrigem > colunaDestino){
        sinalJ =-sinalJ;
    }
    //quarto quadrante, mantÃ©m os dois sinais = 1.

    for(int k = 0, i = 0, j = 0; k < n; k++,i+=1*sinalI,j+=1*sinalJ){
            posicoes[k][0] = linhaOrigem+i;
            posicoes[k][1] = colunaOrigem+j;
    }
    return posicoes;
}
```

Figura 9: Bispo.cpp - Bispo::checaMovimento

Da mesma forma que no rei, o método específico checaMovimento() verifica se as posições são legais, e então retorna uma matriz de movimentos.

Antes do movimento, os parâmetros int linhaOrigem, int colunaOrigem, int linhaDestino e int colunaDestino são verificados. Como o bispo somente se movimenta em diagonal, temos o primeiro if verificando se seu movimento sai disso. Depois de verificado se o movimento é diagonal, temos que determinar qual o quadrante o bispo se movimenta. Baseando-se nos parâmetros do método, usamos os ifs seguintes para ver em qual posição o bispo se movimentará. Após a verificação, o método chama o for, que é responsável por deslocar as casas na matriz de movimento na direção certa usando as váriaveis SinalI e SinalJ para determinar em qual quadrante o bispo se movimenta.

3.1.3 Cavalo

O Cavalo foi construído da seguinte forma:

```
#ifndef CAVALO_H_
#define CAVALO_H_

#include <iostream>
#include "Peca.h"

class Cavalo:public Peca{
    public:
        Cavalo(bool color); //Iniciamos a peÃ§a com a respectiva cor.
        int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n);

};

#endif
```

Figura 10: Cavalo.h

- Cavalo(bool color): Construtor da classe;
- int** checaMovimento(): Método para verificar se o movimento do cavalo é legal ou não. Suas sobrecargas serão explicadas mais a frente;

Agora, quanto a sua implementação:

- Cavalo::Cavalo

```
#include "Cavalo.h"

Cavalo::Cavalo(bool color):Peca(color,'c'){
}
```

Figura 11: Cavalo.cpp - Cavalo::Cavalo

O objeto é sobrecarregado com "Peca(color,'c')- Explicado mais a frente - dizendo que o objeto é de fato, um cavalo;

- Cavalo::checaMovimento

```
int** Cavalo::checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n){

    if( //Possui todas as 8 possíveis posições que um cavalo pode andar em L
   !((linhaOrigem+2 == linhaDestino && colunaOrigem+1 == colunaDestino) ||
    (linhaOrigem+1 == linhaDestino && colunaOrigem+2 == colunaDestino) ||
    (linhaOrigem-1 == linhaDestino && colunaOrigem+2 == colunaDestino) ||
    (linhaOrigem-2 == linhaDestino && colunaOrigem+1 == colunaDestino) ||
    (linhaOrigem-2 == linhaDestino && colunaOrigem-1 == colunaDestino) ||
    (linhaOrigem-1 == linhaDestino && colunaOrigem-2 == colunaDestino) ||
    (linhaOrigem+1 == linhaDestino && colunaOrigem-2 == colunaDestino) ||
    (linhaOrigem+2 == linhaDestino && colunaOrigem-1 == colunaDestino)
    )){
        return 0;
    }
    int **posicoes = alocarMatrizPosicoes(n = 2);

    posicoes[0][1] = linhaOrigem;
    posicoes[0][1] = colunaOrigem;
    posicoes[1][0] = linhaDestino;
    posicoes[1][1] = colunaDestino;

    return posicoes;
}
```

Figura 12: Cavalo.cpp - Cavalo::checaMovimento

Da mesma forma, o método específico checaMovimento() verifica se as posições são legais, e então retorna uma matriz de movimentos.

Como o cavalo tem um movimento peculiar, para facilitar a implementação, foi feita a verificação de todos os oito movimentos possíveis no cavalo, a fim de verificar a legalidade do movimento. Se possível, a função retorna a matriz de movimento, senão, retorna 0;

3.1.4 Torre

A Torre foi construído da seguinte forma:

```
#ifndef TORRE_H_
#define TORRE_H_

#include <iostream>
#include "Peca.h"


class Torre:public Peca{
    private:
        bool roque;
    public:
        Torre(bool color);//Iniciamos a peÃ§a com a respectiva cor.
        int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n); //Checa se o movimento Ã© vÃ¡lido
                                                                                                              //apenas dentro da matriz.
};

#endif
```

Figura 13: Torre.h

- Torre(bool color): Construtor da classe;
- int** checaMovimento(): Método para verificar se o movimento da torre é legal ou não. Suas sobrecargas serão explicadas mais a frente;

Agora, quanto a sua implementação:

- Torre::Torre

```
//Construtor que inicializa o atributo cor e o atributo capturada para falso.
Torre::Torre(bool color):Peca(color,'t'){
}
```

Figura 14: Torre.cpp - Torre::Torre

O objeto é sobrecarregado com "Peca(color,'t')- Explicado mais a frente - dizendo que o objeto é de fato, uma torre;

- Torre::checaMovimento

```
int** Torre::checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n){
    //A torre só anda se a o destino tiver a linha ou coluna igual ao da origem.
    if(linhaOrigem != linhaDestino && colunaOrigem != colunaDestino) return 0;

    int sinalI = 1;
    int sinalJ = 1;
    n = 1;

    if(linhaOrigem != linhaDestino) n += fabs(linhaOrigem - linhaDestino);
    else n += fabs(colunaOrigem - colunaDestino);

    int **posicoes = alocarMatrizPosicoes(n);


    if(linhaOrigem == linhaDestino){
        if(colunaOrigem > colunaDestino) /*((colunaOrigem - colunaDestino) > 0)*/ sinalJ = - sinalJ;
        sinalI = 0;
    }else{
        if(linhaOrigem > linhaDestino)/*((linhaOrigem - linhaDestino) > 0)*/ sinalI = - sinalI;
        sinalJ = 0;
    }

    for(int k = 0, i = 0, j = 0; k < n; k++, i+=1*sinalI, j+=1*sinalJ){
        posicoes[k][0] = linhaOrigem+i;
        posicoes[k][1] = colunaOrigem+j;
    }

    return posicoes;
}
```

Figura 15: Torre.cpp -Torre::checaMovimento

A movimentação da torre se assemelha bastante com a do bispo, já que são feitas ambas em quadrantes, então, os processos de verificação de movimentos são praticamente os mesmos, sendo a única diferença entre eles a movimentação. Como a torre é horizontal ou vertical, no primeiro if é verificada a movimentação da torre, se caso o movimento não for certo, retorna-se 0 .

Se caso o movimento for legal, é necessário saber qual quadrante a torre está se movimentando. Após a verificação, o método entra no for para

poder popular a matriz de posições de acordo com os SinalI e SinalJ, e logo após retorna a matriz.

3.1.5 Peão

O peão foi construído da seguinte forma:

```
#ifndef PEAO_H_
#define PEAO_H_


#include <iostream>
#include "Peca.h"

class Peao:public Peca{
    private:
        int primeiroMov;
    public:
        Peao(bool color);//Iniciamos a peÃ§a com a respectiva cor.
        int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n);
        int getPrimeiroMov();
};

#endif
```

Figura 16: Peao.h

- int primeiroMov: Atributo para determinar o primeiro movimento do peão;
- Peao(bool color): Construtor da classe;
- int** checaMovimento(): Método para verificar se o movimento do peão é legal ou não. Suas sobrecargas serão explicadas mais a frente;
- int getPrimeiroMov(): Verifica se este é o primeiro movimento do Peão;

Agora, quanto a sua implementação:

- Peao::Peao

```
//Construtor que inicializa o atributo cor e o atributo capturada para falso.
Peao::Peao(bool color):Peca(color,'p'){
    primeiroMov = 0;
}
```

Figura 17: Peao.cpp - Peao::Peao

O objeto é sobrecarregado com "Peca(color,'p')- Explicado mais a frente - dizendo que o objeto é de fato, um peão, e também seta o atributo

- Peao::checaMovimento

```
int** Peao::checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n){
    int aux = 1;
    int **posicoes = 0;

    if(!cor){ //Se a cor for branca, teremos que descer de um em um na nossa matriz com o piÃ£o, por isso, fazemos
              //a nossa auxiliar ficar negativa.
        aux = -aux;
    }

    if(linhaOrigem+aux == linhaDestino && (colunaOrigem == colunaDestino || colunaOrigem+1 == colunaDestino || colunaOrigem-1 == colunaDestino)){
        posicoes = alocarMatrizPosicoes(n = 2);
        posicoes[0][0] = linhaOrigem;
        posicoes[0][1] = colunaOrigem;
        posicoes[1][0] = linhaDestino;
        posicoes[1][1] = colunaDestino;
        primeiroMov = 1;
    } else if(primeiroMov == 0 && linhaOrigem+2*aux == linhaDestino && colunaOrigem == colunaDestino){
        posicoes = alocarMatrizPosicoes(n = 3);
        posicoes[0][0] = linhaOrigem;
        posicoes[0][1] = colunaOrigem;
        posicoes[1][0] = linhaOrigem+aux;
        posicoes[1][1] = colunaDestino;
        posicoes[2][0] = linhaDestino;
        posicoes[2][1] = colunaDestino;
        primeiroMov = 2;
    }

    return posicoes;
}
```

Figura 18: Peao.cpp - Peao::checaMovimento

A implementação do peão é bem peculiar, devido a natureza de seus movimentos. Ele só pode andar para a frente, uma casa de cada vez; porém em seu primeiro movimento, ele pode andar duas casas, e para isso, o método verifica a natureza de dois movimentos do peão: Se ele anda uma casa, ou se ele anda duas.

Se o peão já fez seu primeiro movimento, sua matriz de movimento apenas o faz andar uma casa, isso depois de sua cor ter sido determinada, e o atributo "primeiroMov"é setado para "1". Senão, o método entra no segundo if para verificar se ele ainda está em seu primeiro movimento. Se estiver, e o usuário quiser andar duas casas, o método permite o movimento e o atributo "primeiroMov"é setado para "2". Senão, retorna 0;

- Peao::getPrimeiroMov

```
int Peao::getPrimeiroMov(){
    return primeiroMov;
}
```

Figura 19: Peao.cpp - Peao::getPrimeiroMov

O método apenas retorna o estado de "primeiroMov";

3.1.6 Dama

A Dama foi construído da seguinte forma:

```
#ifndef DAMA_H_
#define DAMA_H_

#include <iostream>
#include "Peca.h"

//class Dama:public virtual Bispo, public virtual Torre{
class Dama:public Peca{
    public:
        Dama(bool color);   //Iniciamos a peÃ§a com a respectiva cor.
        int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n);


};
```

Figura 20: Dama.h

- Dama(bool color): Construtor da classe;
- int** checaMovimento(): Método para verificar se o movimento da dama é legal ou não. Suas sobrecargas serão explicadas mais a frente;

Agora, quanto a sua implementação:

- Dama::Dama

```
Dama::Dama(bool color):Peca(color,'d'){
}
```

Figura 21: Dama.cpp - Dama::Dama

  O objeto é sobrecarregado com "Peca(color,'d')- Explicado mais a frente - dizendo que o objeto é de fato, uma dama;

- Dama::checaMovimento

```
int** Dama::checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n){

    if((fabs(linhaOrigem - linhaDestino) != fabs(colunaOrigem - colunaDestino)) &&
    (linhaOrigem != linhaDestino && colunaOrigem != colunaDestino)) {
        return 0;
    }
    n = 1;

    if(linhaOrigem != linhaDestino && colunaOrigem != colunaDestino) n += fabs(linhaOrigem - linhaDestino);
    else if(linhaOrigem != linhaDestino && colunaOrigem == colunaDestino) n += fabs(linhaOrigem - linhaDestino);
    else n += fabs(colunaOrigem - colunaDestino);

    int sinalI;
    int sinalJ;
    sinalI = sinalJ = 1;
    int **posicoes = alocarMatrizPosicoes(n);

    //primeiro quadrante.
    if(linhaOrigem > linhaDestino && colunaOrigem < colunaDestino){
        sinalI  = -sinalI ;
    }
    //segundo quadrante.
    else if(linhaOrigem > linhaDestino && colunaOrigem > colunaDestino){
        sinalI =-sinalI ;
        sinalJ =-sinalJ;
    }
    //terceiro quadrante.
    else if(linhaOrigem < linhaDestino && colunaOrigem > colunaDestino){
        sinalJ =-sinalJ;
    }
    else if(linhaOrigem == linhaDestino){
        if(colunaOrigem > colunaDestino) /*((colunaOrigem - colunaDestino) > 0)*/ sinalJ = - sinalJ;
        sinalI = 0;
    }else if(colunaOrigem == colunaDestino){
        if(linhaOrigem > linhaDestino)/*((linhaOrigem - linhaDestino) > 0)*/ sinalI = - sinalI;
        sinalJ = 0;
    }

    //quarto quadrante, mantÃ©m os dois sinais = 1.

    for(int k = 0, i = 0,j = 0; fabs(i) != n; k++,i+=1*sinalI,j+=1*sinalJ){
            posicoes[k][0] = linhaOrigem+i;
            posicoes[k][1] = colunaOrigem+j;
    }
    return posicoes;
}
```

Figura 22: Dama.cpp - Dama::checaMovimento

A movimentação da Dama é basicamente uma junção de bispo, torre e rei. Então, para poder determinar qual é o movimento para a Dama, as três verificações foram implementadas dentro deste método.

No primeiro if, se caso o movimento não for diagonal, vertical ou horizontal, o método retorna 0;

Caso o movimento seja legal, precisamos saber para qual quadrante a Dama deve se movimentar. Logo, usando os ifs e os sinais SinalI e SinalJ, temos o conhecimento do quadrante que devemos ir, assim, indo para o for que movimenta a dama na matriz de posições, e assim a passa como retorno;

3.2 Classe Peça

A Classe Peça é a responsável por conectar as peças específicas ao resto do jogo, e também tem uma relaçao de composição com as pecas específicas. Ela possui relação de agregação com Jogador (16 peças para cada jogador) e Posição (1 - 0,1), assim como tem composição com Jogo. A Classe gera a matriz de posições, possui os atributos de captura, cor, peça específica e desenho da peça:

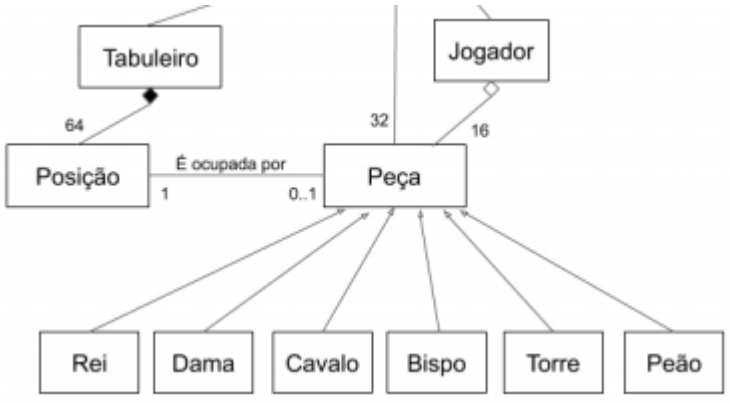

Figura 23: Diagrama de "Peça"

A classe foi feita desta maneira:

```
#ifndef PECA_H_
#define PECA_H_

#include <iostream>

class Peca{
    protected:
        bool capturada; //True se estiver capturada ou false se estiver livre.
        const bool cor; //True se a cor for preta ou false se a cor for branca.
        const char peca;
    public:
        Peca(bool color, char p);
        char desenha();//Retorna um char com a letra da peÃ§a (a cor dela farÃ¡ o retorno ser maiÃºscolo u minÃºsculo.)
        virtual int** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int &n) = 0;
        void captura();
        bool getCapturada(); //Retorna true caso a peÃ§a esteja capturada.
        bool getCor();
        int** alocarMatrizPosicoes(int &n);
};

#endif
```

Figura 24: Peca.h

- bool capturada: booleano que indica se a peça foi ou não capturada;
- const bool cor: booleano que definirá a cor da peça;
- const char peca: informa qual peça se trata;

- Peça(bool color, char p): Construtor da classe;
- char desenha(): Método que retorna a letra que representa a peça do tabuleiro;
- virtual int ** checaMovimento(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino, int n) = 0: É o método que verifica a movimentação das peças. O método é virtual pois cada classe específica tem sua movimentação própria.
- void captura();
- bool getCapturada(): Se a peça estiver capturada, retorna um booleano;
- getCor(): Método qu requisita a cor da peça;
- int** Peca::alocarMatrizPosicoes(int n): aloca a matriz de posições com as peças;

Quanto à implementação:

- Jogador:: Jogador e Jogador::Jogador

```
/*Como jÃ¡ foi explicado no Jogador.h temos dois construtores para diferentes ocasiÃµes. */
Jogador::Jogador(const string &n, const bool &color, Peca **p):nome(n),cor(color),pecas(p){
}

Jogador::Jogador(const Jogador &jogador):nome(jogador.nome),cor(jogador.cor),pecas(jogador.pecas){
}
Jogador::~Jogador(){
    for(int i = 0; i < quant; i++){
        pecas[i] = 0;
    }
}
```

Figura 25: Peca.cpp - Peca::Peca

Aqui temos o construtror do método. Nele são definidas a cor que a peça terá, qual peça será criada, e o booleano que informa se ela foi capturada é setado inicialmente como falso.

- Peca::captura

```
//O atributo capturada é trocado para true.
void Peca::captura(){
    capturada = true;
}
```

Figura 26: Peca.cpp - Peca::captura

No método Peca::captura, temos apenas um booleano que seta a variável para "true"quando a peça for capturada;

- Peca::desenha

```
char Peca::desenha(){
    //Se for preta, é maiúscula.
    if(cor) return toupper(peca);

    return peca;
}
```

Figura 27: Peca.cpp - Peca::desenha

Este método retorna a letra e a cor da peça: se for preta, a letra será maiúscula, do contrário, minúscula.

- Peca::getCapturada

```
//Retorna true caso a peÃ§a esteja capturada.
bool Peca::getCapturada(){
    return capturada;
}
```

Figura 28: Peca.cpp - Peca::getCapturada

O método getCapturada retorna um booleano que indica se a peça foi capturada ou não;

- Peca::getCor

```
bool Peca::getCor(){
    return cor;
}
```

Figura 29: Peca.cpp - Peca::getCor

Um método que captura a cor da peça;

- Peca::alocaMatrizPosicoes

```
int** Peca::alocarMatrizPosicoes(int &n){
    int **posicoes;
    try {
        posicoes = new int*[n];
        for(int i = 0; i < n; i++) posicoes[i] = new int[2];
    }
    catch(bad_alloc){
        cout<<"Memória insuficiente!"<<endl;
        exit(1);
    }
    return posicoes;
}
```

Figura 30: Peca.cpp - Peca::alocaMatrizPosicoes

Este método é interessante. A matriz de posições que foi citada anteriormente é feita para simular o movimento da peça. Este médoto é quem a aloca. O método recebe o número n de casas que o usuário quer andar, e localmente no método a matriz de posições é alocada. Para n posições que a peça buscar andar, uma nova posição de linha e coluna são geradas. Se caso não houver memória suficiente, o try catch acusa erro e mata o programa. Se houver até o laço finalizar, o método retorna a matriz de posições;

3.3 Classe Posição

A classe Posição é a responsável por determinar o que cada posição do tabuleiro terá. Ela tem relação de composição com Tabuleiro (64-1), visto que o tabuleiro tem 64 posições; e uma relação de agregação com peça (1 - 0,1), visto que em cada posição pode haver no máximo uma peça, mas ambas as classes são independentes.

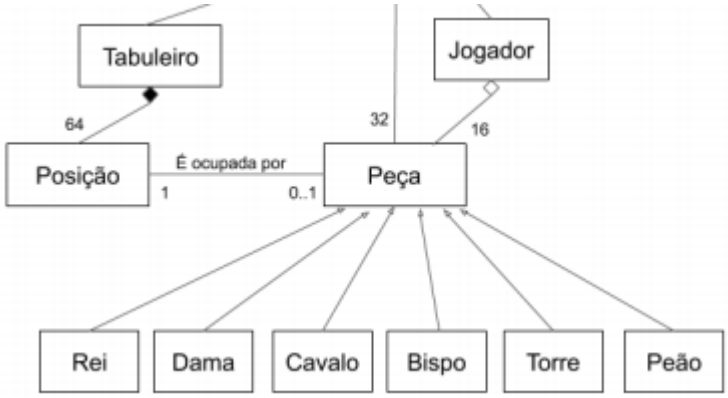

Figura 31: Diagrama de "Posição"

A classe foi feita desta maneira:

```
#ifndef POSICAO_H_
#define POSICAO_H_


#include "Peca.h"
#include <iostream>

class Posicao{
    private:
        Peca *peca;
    public:
        Posicao(Peca *p = 0);
        Peca* getPeca();
        void setPeca(Peca *p);

};
#endif
```

Figura 32: Posicao.h

- Peca *peca: Peça que está na posição
- Posicao(Peca *p = 0): Construtor da classe;
- Peca *getPeca(): método que verifica a peça;
- void setPeca(): método que seta a peça;

Quanto à implementação:

- Posicao::Posicao

```
//Construtor simples que inicializa a peca com a posição vazia.
Posicao::Posicao(Peca *p){
    peca = p;
}
```

Figura 33: Posicao.cpp - Posicao::Posicao

Aqui temos o construtror do método. Nele, a peça que ocupará a posição será setada. O método inicialmente é sobrecarregado com o valor 0, para dizer que a posição está vazia.

- Posicao::setPeca

```
//MÃ©todos set e get.
void Posicao::setPeca(Peca *p){ peca = p; }
```

Figura 34: Posicao.cpp - Posicao::setPeca

Este método recebe *p, contido nele uma peça, a qual é setada no atributo peca da classe;

- Posicao::getPeca

```
Peca* Posicao::getPeca(){
    return peca;
}
```

Figura 35: Posicao.cpp - Posicao::getPeca

E este método é nada mais que um retorno da peça que está na posição;

3.4 Classe Jogador

A classe Jogador é a responsável por gerar os dois jogadores do Xadrez. Jogador tem associação com peça (2 -16), e composição com Jogo.

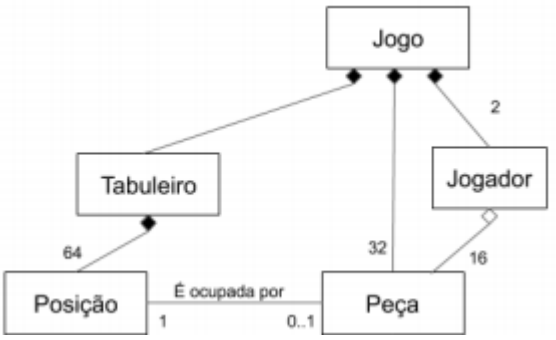

Figura 36: Diagrama de "Jogador"

A classe foi feita desta maneira:

```
#ifndef JOGADOR_H_
#define JOGADOR_H_


#include <iostream>
#include "Peca.h"
#include <iostream>

using std::string;

/*Essa Ã® uma classe genÃ©rica para podermos fazer a implementaÃ§Ã£o parcial do jogo, ou seja, ela estÃ¡ muito limitada.*/
class Jogador{
        private:
            static const int quant = 16;
            const string nome;                  //Nome do jogador.
            const bool cor;                 //Vai ser importante para definir a cor das peÃ§as do jogador.
            Peca **pecas;
        public:
            ~Jogador();
            Jogador(const Jogador &jogador);
            Jogador(const string &n, const bool &color, Peca **p);  //Construtor que recebe a cor e o nome do jogador.
            string getNome();               //Apenas retorna o nome do jogador.
            void imprimirCapturadas();      //Imprimi todas as peÃ§as capturadas.
};

#endif
```

Figura 37: Jogador.h

- static const int quant = 16: atributo que define a quantidade de peças iniciais. Sobrecarregado com 16.
- const string nome: atributo que define o nome do jogador;
- const bool cor: atributo que define a cor do jogador;
- Peça **pecas: matriz que armazenará todas as peças do jogador;
- Jogador(); Destrutor da classe;
- Jogador(const Jogador &jogador): primeiro construtor do jogador;
- Jogador(const string &n, const bool color, Peca **p): segundo construtor do jogador;
- void imprimirCapturadas: método que imprime as peças capturadas na tela;

Quanto à implementação:

- Jogador:: Jogador Jogador::Jogador

```
/*Como jÃ¡ foi explicado no Jogador.h temos dois construtores para diferentes ocasiÃµes. */
Jogador::Jogador(const string &n, const bool &color, Peca **p):nome(n),cor(color),pecas(p){
}

Jogador::Jogador(const Jogador &jogador):nome(jogador.nome),cor(jogador.cor),pecas(jogador.pecas){
}
Jogador::~Jogador(){
    for(int i = 0; i < quant; i++){
        pecas[i] = 0;
    }
}
```

Figura 38: Jogador.cpp - Jogador::J̃ogador & Jogador::Jogador

O destrutor apenas desaloca todas as peças do jogador, desfazendo assim jogo. Já os construtor recebe o nome do jogador, a cor de suas peças e um vetor de ponteiros do tipo Peca;

- Jogador::getNome

```
//Apenas retorna o nome.
string Jogador::getNome(){
    return nome;
}
```

Figura 39: Jogador.cpp - Jogador::getNome

Neste método, retornamos o nome do Jogador;

- Jogador::imprimeCapturadas

```
void Jogador::imprimirCapturadas(){
    int capturadas = 0;
    cout<<"                    ";
    for(int i = 0; i < quant; i++){
        if(capturadas == 8) cout<<endl<<"                    ";
        if(pecas[i]->getCapturada() == true){
            if(pecas[i]->getCor() == true) cout<<"\033[1;30m"<<pecas[i]->desenha()<<"\033[0m";
            else cout<<pecas[i]->desenha();
            cout<<"  ";
            capturadas+=1;
        }
    }
    cout<<endl;
}
```

Figura 40: Jogador.cpp - Jogador::imprimeCapturadas

Neste método, são imprimidas todas as peças capturadas, a cada linha, oito elementos (as peças adversárias capturadas) são impressos.

3.5 Classe Tabuleiro

A Classe Tabuleiro é a mais essencial de todo o jogo. Nela, todas as verificações das jogadas são feitas, assim como a sobrecarga de operadores, lista de jogadas e regras das peças. Os estados de cheque, cheque-mate, En Passant e movimento ilegal também são determinadas por esta classe. Ela compõe Posição (1-64), e é composta por Jogo.

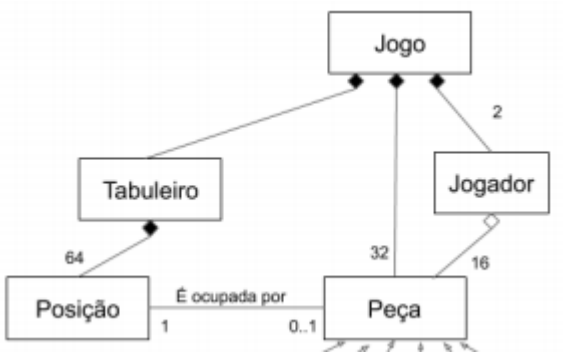

Figura 41: Diagrama de "Tabuleiro"

A classe foi feita desta maneira:

```
#ifndef TABULEIRO_H_
#define TABULEIRO_H_
#include <iostream>
#include <queue>
#include "Posicao.h"
#include "Tabuleiro.h"
#include "Peca.h"
using std::queue;
using std::string;
using std::ostream;
using std::istream;

class Tabuleiro{
    private:
        static const int tamanho = 8;       //Uma constante para a matriz.
        Posicao *tab[tamanho][tamanho];     //uma matriz de Posicao de 8 X 8.
        //Representa os estados do en Passant da linha D e E.
        int enPassantD[tamanho];
        int enPassantE[tamanho];
        queue<string> jogadas;  //Fila para guardar as jogadas em um arquivo.
        int estadoTabuleiro;

    public:
        //Construtor do tabuleiro que recebe as peÃ§as brancas e pretas e a quantidade para coloca-las no tabuleiro.
        Tabuleiro(Peca **pretas, Peca **brancas,const int quantPecas, const int &estado);
        ~Tabuleiro(); //Destrutor.
        void imprimirTabuleiro() const; //Imprimi o tabuleiro.
        void encontrarMeuRei(int &linha, int&coluna, const bool vez); //Encontra o rei do jogador.
        //Faz a checagem se o enPassant Ã© valido e se for retorna a peÃ§a que vai ser capturada por enPassant.
        Peca* checaEnPassant(const int &linhaOrigem, const int &colunaOrigem, const int &linhaDestino, const int &colunaDestino, const bool &vez, int &enPassant);
        //Controla o EnPassant atualizando os atributos enpassantD e enpassantE.
        void controleEnPassant(const int &linhaOrigem, const int &colunaOrigem, const int &linhaDestino, const int &colunaDestino);
        //Atualizamos os atributos de EnPassant a cada jogada.
        void atualizaEnPassant();
        //Checamos os movimentos e retornamos -1 para erro, 2 para vocÃª deixou o outro jogador em xeque e 3 para o outro jogador recebeu xeque mate.
        int checaMovimentoNoTabuleiro(const string& origem, const string& destino, const bool &vez);
        //Verificamos se a jogada passada Ã© vÃ¡lida.
        bool verificaJogada(const string& origem, const string& destino, int &linhaOrigem, int &colunaOrigem, int &linhaDestino, int &colunaDestino);
        //Verificamos se um dado jogador estÃ¡ em xeque.
        bool verificarXeque(const int &linha, const int &coluna, const bool &vez);
        //Verificamos se uma dada posiÃ§Ã£o ou peÃ§a pode ser alcanÃ§ada por um dado jogador.
        bool verificarCaptura(const int &linhaOrigem, const int &colunaOrigem,const int &linhaDestino, const int &colunaDestino, const bool vez);

        //Retorna a quantidade de jogadas que estÃ¡ no tabuleiro.
        int quantJogadas() const;
        //Guarda uma dada jogada.
        void guardarJogada(const string& j);
        //Retorna a fila de jogadas que Ã© o nosso atributo.
        queue<string>getJogadas() const;
        //Imprimi o tabuleiro.
        friend ostream & operator<< (ostream & o, Tabuleiro &tab);
        //Recebe uma jogada do jogador.
        friend istream &operator>> (istream & i, Tabuleiro &tab);
        int getEstadoTabuleiro() const;
        void setEstadoTabuleiro(const int &estado);
};
```

Figura 42: Tabuleiro.h

Para evitar confusão, explicaremos método a método diretmente, ao invés de desmembrar o .h. Os métodos são:

- Tabuleiro::Tabuleiro

```
//Criamos a matriz de posiÃ§Ãµes e passamos as peÃ§as para suas respectivas posiÃ§Ãµes iniciais.
Tabuleiro::Tabuleiro(Peca **pretas, Peca **brancas, const int quantPecas, const int &estado){
    int t = 0;
    estadoTabuleiro = estado;

    for(int i = 2; i < tamanho - 2; i++){
        for(int j = 0; j < tamanho; j++){
            tab[i][j] = new Posicao();
        }
    }

    for(int i = 0; i < 2; i++){
        for(int j = 0; j < quantPecas/2; j++,t++){

                tab[i][j] = new Posicao(pretas[t]);
                tab[tamanho -1 -i][j] = new Posicao(brancas[t]);
        }
    }

    for(int i = 0; i < tamanho; i++){
        enPassantD[i] = 0;
        enPassantE[i] = 0;
    }
}
```

Figura 43: Tabuleiro.cpp - Tabuleiro::Tabuleiro

É o construtor da classe. Ele seta o "estadoTabuleiro"através do parâmetro estado, aloca as posições das peças brancas e pretas, aloca as peças em si nas posições para o início do jogo, e deixa as linhas D e E que usam o enPassant como 0;

- Tabuleiro::imprimeTabuleiro

```
//Imprimi o tabuleiro e pinta as peÃ§as pretas.
void Tabuleiro::imprimirTabuleiro() const{
    Peca *aux;
    string linhas = "abcdefgh";
    cout<<endl<<endl<<"                    ";
    cout<<"    1   2   3   4   5   6   7   8"<<endl;
    cout<<"                  ";
    cout<<"  _____________________________ "<<endl;
    for(int i = 0; i < tamanho; i++){
        cout<<"                   ";
        cout<<linhas[i]<<" ";

        for(int j = 0; j < tamanho; j++){
            cout<<"| ";
            if((aux = tab[i][j]->getPeca()) == 0) cout<<" ";
            else if(aux->getCor() == true) cout<<"\033[1;30m"<<aux->desenha()<<"\033[0m";
            else cout<<aux->desenha();
            cout<<" ";
        }

        cout<<"|"<<endl;
        cout<<"                  ";
        cout<<"  |___|___|___|___|___|___|___|___|"<<endl;
    }
    cout<<endl<<endl;
}
```

Figura 44: Tabuleiro.cpp - Tabuleiro::imprimeTabuleiro

Este método basicamente imprime o tabuleiro posição a posição com as peças atualizadas para que os jogadores possam visualizar o que está havendo; Com o auxílio da variável do tipo Peca *aux. o tabuleiro consegue varrer as posições e identificar cada peça pertencente ao tabuleiro.

- Tabuleiro::encontrarMeuRei

```
//MÃ©todo para encontrar o rei, vasculhamos todo o tabuleiro atÃ© encontrar nosso rei.
void Tabuleiro::encontrarMeuRei(int &linha, int &coluna, const bool vez){
    Peca *aux;

    for(int i = 0; i < tamanho; i++){
        for(int j = 0; j < tamanho; j++){
            //Passamos por todas as peÃ§as.
            aux = tab[i][j]->getPeca();
            if(aux != 0 && toupper(aux->desenha()) == 'R' && aux->getCor() == vez){
                linha = i;
                coluna = j;
                return;
            }
        }
    }
}
```

Figura 45: Tabuleiro.cpp - Tabuleiro::encontrarMeuRei

O método simplesmente varre todo o tabuleiro a procura do Rei, sempre considerando a vez de cada jogador.

- Tabuleiro::checaEnPassant

```
//Cheamos se o enPassant poderÃ¡ ser aplicado.
Peca* Tabuleiro::checaEnPassant(const int &linhaOrigem, const int &colunaOrigem, const int &linhaDestino, const int &colunaDestino, const bool &vez, int &enPassant){
    int j = 1; //ServirÃ¡ para saber se o en Passant Â® para esquerda ou direita.

    //Para saber se o en Passant serÃ¡ para direita ou esquerda.
    if(colunaOrigem > colunaDestino) j = -j;
    if(tab[linhaDestino][colunaDestino]->getPeca() != 0 && colunaOrigem == colunaDestino) return 0;
    Peca *aux = tab[linhaOrigem][colunaOrigem+j]->getPeca();


    if(vez == false && linhaOrigem == 3 && enPassantD[colunaOrigem] == 5 &&
       enPassantD[colunaOrigem+j] == 2){
            enPassant = j;
            return aux;
    }

     if(vez == true && linhaOrigem == 4 && enPassantE[colunaOrigem] == 5 &&
       enPassantE[colunaOrigem+j] == 2){
            enPassant = j;
            return aux;
    }
    return 0;
}
```

Figura 46: Tabuleiro.cpp - Tabuleiro::checaEnPassant

O En Passant (movimento especial de captura do peão) foi complicado de implementar, já que é um movimento específico, mas que exigia conhecimento das outras peças, o que o peão não tem. Logo, foi tomada a decisão de o tornar uma regra de tabuleiro, tornando sua verificação com as regras bem mais fácil. Este método retorna um ponteiro do tipo Peça após verificar os possiveis movimentos do passant (Direita e Esqueda). Verifica se a peça que deseja fazer o movimento está cumprindo as exigências, e se o estado da peça permite a captura.

- Tabuleiro::controleEnPassant

```
//Controlamos o EnPassant setando estados para quem irÃ¡ ser capturada e para quem irÃ¡ capturar.
void Tabuleiro::controleEnPassant(const int &linhaOrigem, const int &colunaOrigem, const int &linhaDestino, const int &colunaDestino){

        Peao *pecaOrigem = (Peao*)tab[linhaOrigem][colunaOrigem]->getPeca();

        if(pecaOrigem->getCor() == false && linhaDestino == 3){
            enPassantD[colunaDestino] = 3;
        }
        else if(pecaOrigem->getCor() == false && linhaDestino == 4 && pecaOrigem->getPrimeiroMov() == 2){
            enPassantE[colunaDestino] = 1;
        }
        else if(pecaOrigem->getCor() == true && linhaDestino == 4 ){
            enPassantE[colunaDestino] = 3;
        }
        else if(pecaOrigem->getCor() == true && linhaDestino == 3 && pecaOrigem->getPrimeiroMov() == 2){
            enPassantD[colunaDestino] = 1;
        }
}
```

Figura 47: Tabuleiro.cpp - Tabuleiro::controleEnPassant

Seguindo o método anterior, este método atualiza o estado do EnPassantD e EnPassantE (passant pela direita ou esquerda) a cada jogada feita.

- Tabuleiro::atualizaEnPassant

```
void Tabuleiro::atualizaEnPassant(){

    for(int i = 0; i < tamanho; i++){

        if(enPassantD[i] == 1) enPassantD[i]++;
        else if(enPassantD[i] == 2) enPassantD[i] = 0;

        if(enPassantD[i] >= 3 && enPassantD[i] < 5) enPassantD[i]++;
        else if(enPassantD[i] == 5) enPassantD[i] = 0;
        //else if((enPassantD[i] == 3 && (aux = tab[3][i]->getPeca()) == 0) || aux->getCor() == false) enPassantD[i] = 0;



        if(enPassantE[i] == 1) enPassantE[i]++;
        else if(enPassantE[i] == 2) enPassantE[i] = 0;

        if(enPassantE[i] >= 3 && enPassantE[i] < 5) enPassantE[i]++;
        else if(enPassantE[i] == 5) enPassantE[i] = 0;

    }
```

Figura 48: Tabuleiro.cpp - Tabuleiro::atualizaEnPassant

E este método por fim, altera os estados de enPassant possíveis, além de verificar se o enPassant pode causar cheque ou cheque-mate;

- Tabuleiro::checaMovimento

```
int Tabuleiro::checaMovimentoNoTabuleiro(const string& origem, const string& destino, const bool &vez){

    //VariÃ¡veis que irÃ£o ser passadas para o mÃ©todo verificarJogada.
    int linhaOrigem = -1;
    int colunaOrigem = -1;
    int linhaDestino = -1;
    int colunaDestino = -1;

    //VariÃ¡vel que irÃ¡ ser passada para o checaMovimento das peÃ§as.
    int n;

    //VariÃ¡veis que irÃ¡ ser passada para o mÃ©todo encontrarMeuRei.
    int linhaRei;
    int colunaRei;

    //VariÃ¡vel que serÃ¡ passada para o mÃ©todo checaEnPassant.
    int enPassant = 0;
    int retorno = -1; //Para auxiliar nos vÃ¡rios retornos.

    //IrÃ£o verificar se Ã© possÃ-vel capturar ou movimentar.
    bool ocorreuCaptura = false;
    bool ocorreuMovimento = false;


    //Desmantelamos as string para posiÃ§Ãµes que podem ser usadas.
    if(verificaJogada(origem,destino,linhaOrigem,colunaOrigem,linhaDestino,colunaDestino) == false) return -1;
    Peca *pecaOrigem,*pecaDestino; //VariÃ¡veis para nos auxiliar na movimentaÃ§Ã£o e captura.


    //Verificamos se a origem nÃ£o Ã© nula e se Ã© a peÃ§a pertencente ao jogador.
    if((pecaOrigem = tab[linhaOrigem][colunaOrigem]->getPeca()) == 0  || pecaOrigem->getCor() != vez) return -1;
    //Verificamos se o destino nÃ£o estÃ¡ com uma peÃ§a da nossa cor.
    if((pecaDestino = tab[linhaDestino][colunaDestino]->getPeca()) != 0 && pecaDestino->getCor() == vez) return -1;

    //Solicitamos a matriz de posiÃ§Ãµes que retorna nulo caso seja invÃ¡lida.
    int **posicoes = pecaOrigem->checaMovimento(linhaOrigem, colunaOrigem, linhaDestino, colunaDestino,n);
    if(posicoes == 0) return -1;

    //Verificamos se nÃ£o tem peÃ§as ocupando a locomoÃ§Ã£o.
    for(int i = 1; i < n-1; i++){
        if(tab[posicoes[i][0]][posicoes[i][1]]->getPeca() != 0){
            return -1;
        }
    }
```

```
//Verificamos se Ã© um peÃ£o que tem capturas e movimentaÃ§Ãµes especiais.
if(toupper(pecaOrigem->desenha()) == 'P'){
    //Se forem as linhas nas condiÃ§Ãµes teremos de acionar o nosso controle dos atributos.
    if(linhaDestino == 3 || linhaDestino == 4) controleEnPassant(linhaOrigem, colunaOrigem, linhaDestino, colunaDestino);
    //Se a origem e destino possuem a mesma coluna e o mesmo Ã© nulo, entÃ£o podemos movimentar.
    if(colunaOrigem == colunaDestino && pecaDestino == 0){
        ocorreuMovimento = true;
    /* Se o destino e origem forem diferentes e o destino nÃ£o for nulo, temos a captura do peÃ£o normal
     * , porÃ©m, se nÃ£o for, chamamos o checaEnPassant para verificar se possui a captura por en passant.*/
    } else if((colunaOrigem != colunaDestino && pecaDestino != 0) ||
    (pecaDestino = checaEnPassant(linhaOrigem, colunaOrigem, linhaDestino, colunaDestino, vez, enPassant)) != 0){
        ocorreuCaptura = true;
    }
}
//PorÃ©m, se nÃ£o for um peÃ£o, o cÃ³digo em si Ã© polimorfico para as demais peÃ§as.
else {
    //Caso o destino seja vazio nÃ£o temos captura.
    if(pecaDestino == 0 ) {
        ocorreuMovimento = true;
    //Se nÃ£o, temos captura.
    } else {
        ocorreuCaptura = true;
    }
}

//Como nÃ£o iremos mais utilir a matriz de posiÃ§Ãµes podemos remover.
 for(int i = 0; i < n; i ++){
        delete posicoes[i];
}
delete []posicoes;

//Agora, podemos verificar as movimentaÃ§Ãµes, se movimentou ou capturou entÃ£o movemos a peÃ§a.
if(ocorreuMovimento == true || ocorreuCaptura == true){
    tab[linhaDestino][colunaDestino]->setPeca(pecaOrigem); //Setando a origem e destino.
    tab[linhaOrigem][colunaOrigem+enPassant]->setPeca(0); //Apagando os rastros.
    tab[linhaOrigem][colunaOrigem]->setPeca(0);
}

if(ocorreuCaptura == true){
    pecaDestino->captura();
}
//Caso a movimentaÃ§Ã£o nÃ£o nos deixou em xeque entÃ£o temos que verificar se o jogador contrÃ¡rio esta em xeque.
encontrarMeuRei(linhaRei,colunaRei,!vez);//Procuramos o rei dele.
//Se o rei dele estÃ¡ em xeque iremos retornar o estado 2.
if(verificarXeque(linhaRei,colunaRei,!vez) == true){
    retorno = 2;
} else { //Se o rei nÃ£o estiver em xeque o jogo estÃ¡ normal.
    retorno = 1;
}
encontrarMeuRei(linhaRei,colunaRei,vez);//Procuramos o rei dele.
if(verificarXeque(linhaRei,colunaRei,vez) == true){
    retorno = 3;
}
    //Atualizamos os atributos do enPassant.
atualizaEnPassant();
return retorno;//Retornamos.
```

Figura 49: Tabuleiro.cpp - Tabuleiro::checaMovimento

Este método é o mais longo de todos, porém é simples de compreender rasamente o que está havendo, por isso, vamos por partes:

- Primeiramente, o método zera todas as flags de controle (ocorreuCaptura, ocorreuMovimento, enPassant) para garantir que tudo ocorrerá somente após a verificação do movimento;
- Logo após, o método chama o método verificaJogada() para checar se o movimento que foi inserido pelo jogador da peça em questão é legal;
- Logo após, verificamos se o destino da peça pode ser alcançado ou não, ou se existe alguma peça que impeça o movimento até lá;

- Se todas as verificações passarem, o movimento é legal;
- Para casos especiais, como o peão, verificamos a legimitidade do movimento, como os dois pulos e o En Passant. As flags e os métodos de checagem dos movimentos especiais (encontarMeuRei, atualizaEnPassant, entre outros) são chamados e verificam o movimento. Por fim, o método retorna um estado indicando se o movimento foi legal ou não;

- Tabuleiro::verificaJogada

```
bool Tabuleiro::verificaJogada(const string& origem, const string& destino, int &linhaOrigem, int &colunaOrigem, int &linhaDestino, int &colunaDestino){
//Verificamos se a orgem e o destino sÃ£o iguais.
    if(origem == destino) return false;

    //Representa as linhas que serÃ£o passadas para inteiros.
    string linhas = "ABCDEFGH";
    int n = linhas.size(); //tamanho da string.

    //-------------------------------------
    colunaOrigem = stoi(origem.substr(1,1))-1; //Passamos o caracter em string para inteiro.
    //-------------------------------------
    colunaDestino = stoi(destino.substr(1,1))-1; //Passamos o caracter em string para inteiro.
    //-------------------------------------
    //Passamos a posiÃ§Ã£o que se encontra em caracter para inteiro.
    for(int i = 0; i < n; i++){

        if(linhas[i] == origem[0]){ linhaOrigem = i;}
        if(linhas[i] == destino[0]){ linhaDestino = i;}
        if(linhaOrigem > 0 && linhaDestino > 0 && colunaOrigem > 0 && colunaDestino > 0) break;
    }
    //Verificamos se as posiÃ§Ãµes origem e destino sÃ£o validas no tabuleiro.
    if(linhaOrigem < 0 || linhaOrigem >= n || linhaDestino < 0 || linhaDestino >= n || colunaOrigem < 0 || colunaOrigem >= n || colunaDestino < 0 || colunaDestino >= n) return false;

    return true;
}
```

Figura 50: Tabuleiro.cpp - Tabuleiro::verificaJogada

Este é o método que verifica se a entrada do usuário foi válida ou não. Ele vem com os parâmetros de origem e destino. Ele compara o que o usuário escreveu com os limites do tabuleiro e a sintaxe do jogo. Ele retorna um booleano: true se a jogada for legítima, false se não for.

- Tabuleiro::verificaXeque

```
//Verifica se determinada posiÃ§Ã£o pode ser capturada pelas peÃ§as contrÃ¡rias.
bool Tabuleiro::verificarXeque(const int &linha, const int &coluna, const bool &vez){

    for(int i = 0; i < tamanho; i++){
        for(int j = 0; j < tamanho; j++){
            //Passamos a vez do jogador contrÃ¡rio e como destino a nossa peÃ§a.
            if(verificarCaptura(i, j,linha,coluna,!vez) == true) return true;
        }
    }
    return false;
}
```

Figura 51: Tabuleiro.cpp - Tabuleiro::verificaXeque

Aqui, verificamos se o rei está em cheque. Basta que usemos verificarCaptura(), varrendo todas as casas do tabuleiro. Verificando se a

posição pode ser capturada por qualquer peça contrária, temos a certeza se é cheque ou não;

- Tabuleiro::verificaCaptura

```
bool Tabuleiro::verificarCaptura(const int &linhaOrigem, const int &colunaOrigem,const int &linhaDestino, const int &colunaDestino, const bool vez){
    int n;
    //Criamos dois apontadores do tipo Peca para receber as pecas origem e destino que iremos trabalhar.
    Peca *pecaOrigem,*pecaDestino;


    //Verificamos se existe determinada peÃ§a na origem e destino.
    if((pecaOrigem = tab[linhaOrigem][colunaOrigem]->getPeca()) == 0  || pecaOrigem->getCor() != vez) return false;
    if((pecaDestino = tab[linhaDestino][colunaDestino]->getPeca()) != 0 && pecaDestino->getCor() == vez) return false;

    //Solicitamos a matriz de posiÃ§Ãµes que retorna nulo caso seja invÃ¡lida.
    int **posicoes = pecaOrigem->checaMovimento(linhaOrigem, colunaOrigem, linhaDestino, colunaDestino,n);
    if(posicoes == 0) return false;

    //Verificamos se as posiÃ§Ãµes entre as peÃ§as origens e destino estÃ£o vazias, pois sÃ³ assim serÃ¡ possÃ-vel
    //deslocar, por exemplo, a torre ou bispo.
    for(int i = 1; i < n-1; i++){
        if(tab[posicoes[i][0]][posicoes[i][1]]->getPeca() != 0){
            return false;
        }
    }
    //Removemos a matriz de posiÃ§Ãµes.
    for(int i = 0; i < n; i ++){
        posicoes[i] = 0;
        delete posicoes[i];
    }
    delete []posicoes;

    //Retornamos true ou false caso as capturas sejam possÃ-veis ou nÃ£o.
    if(toupper(pecaOrigem->desenha()) == 'P'){
        if((colunaOrigem == colunaDestino && pecaDestino == 0) || (colunaOrigem != colunaDestino && pecaDestino != 0)){
            return true;
        }
        return false;
    }
    return true;
}
```

Figura 52: Tabuleiro.cpp - Tabuleiro::verificaCaptura

Aqui, vemos se o movimento pode gerar a captura de alguma peça adversária. A verificação é bem parecida com a do checaMovimento(). Se não existir nenhuma peça entre origem destino, e não houver peça amiga na posição de destino, a peça do destino é captura. No caso do peão, suas diagonais apenas são verificadas.

- Tabuleiro::quantJogadas

```
//Retorna a quantidade de jogadas
int Tabuleiro::quantJogadas() const{
    return jogadas.size();
}
```

Figura 53: Tabuleiro.cpp - Tabuleiro::quantJogadas

O método apenas retorna a quantidade de jogadas que foram feitas;

- Tabuleiro::guardarJogada

```
//Guarda a jogada na fila.
void Tabuleiro::guardarJogada(const string& j){
    jogadas.push(j);
}
//Retornamos nossa fila
```

Figura 54: Tabuleiro.cpp - Tabuleiro::guardarJogada

O método apenas insere a jogada atual na fila de jogadas;

- queue<string> Tabuleiro::getJogadas

```
queue<string> Tabuleiro::getJogadas() const{
        return jogadas;
}
```

Figura 55: Tabuleiro.cpp - queue<string>Tabuleiro::getJogadas

Este método é especial, mas simples; Ele na verdade é uma fila, com todas as jogadas feitas no jogo. Este método retorna uma fila com todas as jogadas armazenadas;

- Operadores

```
//Imprimimos o tabuleiro que Ã© a saÃ-da.
ostream & operator<< (ostream & o, Tabuleiro &tab){
    tab.imprimirTabuleiro();
    return o;
}

//A entrada colocarÃ¡ todas as jogadas validas em uma fila.
istream &operator>> (istream & i, Tabuleiro &tab){

    char aux;
    int linha,coluna, estado = -1;
    bool vez = tab.quantJogadas()%2;
    string s;
    string origem;
    string destino;


    for(int j = 0; j < 6; j++){
        i>>aux;
        s+=toupper(aux);
    }



    //Verificamos se possui algum caso especial de entrada.
    //tab.setEstadoTabuleiro(5);
    //return i;
    if(s == "PAUSAR"){
        tab.setEstadoTabuleiro(5);
        return i;
    }
    else if(s == "RENDER"){
        tab.setEstadoTabuleiro(6);
        return i;
    }else if(s.find(">>") == -1) throw -1;

    origem = s.substr(0,s.find(">>"));
    destino = s.substr(s.find(">>")+2,s.size());

    tab.encontrarMeuRei(linha,coluna,vez);

    if((estado = tab.checaMovimentoNoTabuleiro(origem,destino,vez)) != -1){
        tab.setEstadoTabuleiro(estado);
    } else throw -1;

    if(vez == false && estado == 2) estado = 3;
    else if(vez == true && estado == 2) estado = 2;
    else if(vez == false && estado == 3) estado = 4;
    else if(vez == true && estado == 3) estado = 4;
    tab.setEstadoTabuleiro(1);
    if(estado != 4) tab.guardarJogada(s);

    return i;
}
```

Figura 56: Tabuleiro.cpp - Operadores

Aqui nós temos sobrecargas de operadores de entrada e saída. Quando recebemos a saída (ostream), imprimimos o tabuleiro para o usuário, e retornamos a saída; Na entrada, desmantelamos a entrada char a char,

e verificamos se os caracteres batem. Dentro dos operadores, também verificamos os estados de cada jogada. Para cada estado, fazemos algo (xeque, pausa, render, xeque-mate).

- Tabuleiro::getEstadoTabuleiro

```
int Tabuleiro::getEstadoTabuleiro() const{
    return estadoTabuleiro;
}
```

Figura 57: Tabuleiro.cpp - getEstadoTabuleiro

Método que retorna o estado do tabuleiro;

- Tabuleiro::setEstadoTabuleiro

```
void Tabuleiro::setEstadoTabuleiro(const int &estado){
    estadoTabuleiro = estado;
}
```

Figura 58: Tabuleiro.cpp - Tabuleiro::setEstadoTabuleiro

Método que seta o estado do tabuleiro;

3.6 Classe Jogo

E por fim, a Classe Jogo. Esta classe é a que gera tudo. Faz composição com Tabuleiro (1-1) e Jogador (1-2). É a classe responsável por verificar os turnos, armazenar as jogadas feitas e gerar o jogo em si.

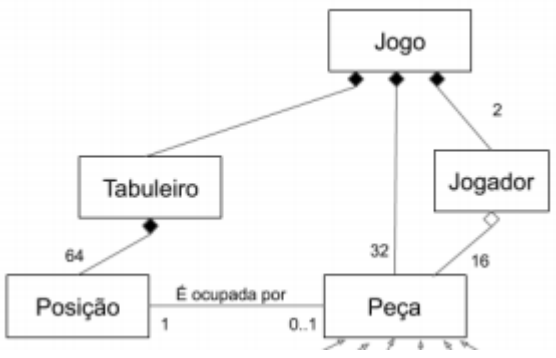

Figura 59: Diagrama de "Jogo"

A classe foi feita desta maneira:

```
#ifndef JOGO_H_
#define JOGO_H_

#include "Jogador.h"
#include "Tabuleiro.h"
#include "Peca.h"
#include <iostream>

using std::string;

class Jogo{
    private:
        //Por definiÃ§Ã£o, o jogador 0 sempre terÃ¡ as brancas, fazendo-o assim, sempre ser o primeiro.
        static const int quantPecas = 16;
        //Objetos que sÃ£o os jogadores.
        Jogador *jogador_0;
        Jogador *jogador_1;
        //PeÃ§as dos jogadores.
        Peca *pretas[quantPecas];
        Peca *brancas[quantPecas];
        //Enum que representa a vez.
        enum v{JOGADOR0,JOGADOR1};
        //Enum que representa os estados da variÃ¡vel estado.
        enum e{ENTRADA_INVALIDA = -1, INICIO_JOGO, MEIO_JOGO, BRANCO_EM_XEQUE, PRETO_EM_XEQUE, RECEBEU_XEQUE_MATE, PAUSAR, RENDER};

        //Criamos um atributo Tabuleiro.
        Tabuleiro *tab;

        //Atributos para representar a vez e o estado do jogo.
        bool vez;
        int estado;



    public:
        //O construtor do jogo recebe dois objetos Jogador que serÃ£o os jogadores.
        Jogo(const string &nome_jogador0, const string &nome_jogador1);
        ~Jogo();
        void estadoDoJogo();    /*O estado do jogo, imprimi o estado atual (inicio do jogo,
        void mudarVezDoJogador();   //Muda a vez do jogador.
        void interface();       //Uma sub interface.
        void play();            //MÃ©todo que inicia o jogo.
        void salvarJogo(queue<string> jogadas); //MÃ©todo para salvar o jogo em um arquivo.
        void carregarJogo(); //MÃ©todo para carregar o jogo.
        int getEstado() const;
};

#endif
```

Figura 60: Jogo.h

Para evitar confusão, explicaremos método a método diretmente, ao invés de desmembrar o .h. Os métodos são:

- Jogo::Jogo

```
//Recebemos o nome dos jogadores.
Jogo::Jogo(const string &nome_jogador0, const string &nome_jogador1){
    vez = JOGADOR0; //Inicia com false, pois o false representa as brancas e as brancas sempre serÃ£o as primeiras.
    estado = INICIO_JOGO; //Inicio do jogo Ã© o 0.
    //Tentamos alocar as peÃ§as de cada jogador.
    try{
        pretas[0] = new Torre(true);
        pretas[1] = new Cavalo(true);
        pretas[2] = new Bispo(true);
        pretas[3] = new Dama(true);
        pretas[4] = new Rei(true);
        pretas[5] = new Bispo(true);
        pretas[6] = new Cavalo(true);
        pretas[7] = new Torre(true);

        brancas[0] = new Torre(false);
        brancas[1] = new Cavalo(false);
        brancas[2] = new Bispo(false);
        brancas[3] = new Dama(false);
        brancas[4] = new Rei(false);
        brancas[5] = new Bispo(false);
        brancas[6] = new Cavalo(false);
        brancas[7] = new Torre(false);

        for(int j = quantPecas/2; j < quantPecas; j++){
            pretas[j] = new Peao(true);
            brancas[j] = new Peao(false);
        }
        tab = new Tabuleiro(pretas,brancas,quantPecas,estado);
        jogador_1 = new Jogador(nome_jogador1,true,pretas);
        jogador_0 = new Jogador(nome_jogador0,false,brancas);

            //Usamos o catch para indicar um bad_alloc.
            }catch(bad_alloc){
                cout<<"MemÃ³ria insuficiente"<<endl;
                exit(1);
            }
    }
```

Figura 61: Jogo.cpp - Jogo::Jogo

É o construtor da classe. Aloca todas as peças em suas posições específicas. Se caso a houver algum problema com a alocação, uma exceção e lançada;

- Jogo::~Jogo

```
Jogo::~Jogo(){

    delete tab;
    delete jogador_0;
    delete jogador_1;

    for(int i = 0; i < quantPecas; i++){
        delete pretas[i];
        delete brancas[i];
    }

}
```

Figura 62: Jogo.cpp - J̃ogo::Jogo

Destrutor da classe. Desaloca o tabuleiro, os jogadores e as peças.

- Jogo::mudarVezDeJogador

```
//ApÃ³s uma jogada do jogador atual, Ã© chamado esse mÃ©todo para trocar a vez.
void Jogo::mudarVezDoJogador(){
    vez = tab->quantJogadas()%2;
}
```

Figura 63: Jogo.cpp - mudarVezDeJogador

O método simplesmente faz o resto de quantas jogadas existem na fila, e popula o atributo "vez".

- Jogo::estadoDoJogo

```
void Jogo::estadoDoJogo(){
        system("clear"); //Vai ser usado para dar uma falsa impressÃ£o de dinamismo ao jogo.
        cout<<"\n\n\n"<<endl;
        ostream &operator<< (ostream & o, Tabuleiro &tab);

        cout<<"                    ";
        if(estado == MEIO_JOGO){
            cout<<"      JOGO EM ANDAMENTO:"<<endl;
            cout<<"                         ";
            cout<<"Vez do jogador ";
            if(vez == JOGADOR0)             cout<<jogador_0->getNome();
            else if(vez == JOGADOR1)        cout<<jogador_1->getNome();

            cout<<"!"<<endl;

        }else if(estado == BRANCO_EM_XEQUE){
            cout<<"    O jogador "<<jogador_0->getNome()<<" estÃ¡ em xeque!"<<endl;
        }else if(estado == PRETO_EM_XEQUE){
            cout<<"    O jogador "<<jogador_1->getNome()<<" estÃ¡ em xeque!"<<endl;
        }

        else if(estado == RECEBEU_XEQUE_MATE){
            cout<<"O jogador ";
            if(vez == JOGADOR0) cout<<jogador_0->getNome();
            else        cout<<jogador_1->getNome();

            cout<<" recebeu xeque-mate!"<<endl;
        }else if(estado == RENDER){
            cout<<"    O jogador ";
            if(vez == JOGADOR0) cout<<jogador_0->getNome();
            else        cout<<jogador_1->getNome();


            cout<<" rendeu-se!"<<endl;
        }
        else if(estado == INICIO_JOGO){
            cout<<"         Inicio do jogo!"<<endl;
            cout<<"                        ";
            cout<<"Vez do jogador "<<jogador_0->getNome()<<"!"<<endl;
        }else if(estado == PAUSAR){
            cout<<"     O jogo foi pausado!"<<endl;
        }
        jogador_0->imprimirCapturadas();
        cout<<*tab;
        jogador_1->imprimirCapturadas();
}
```

Figura 64: Jogo.cpp - Jogo::estadoDoJogo

Este método checa o estado do jogo de acordo com o atributo de estado e printa na tela para o jogador de acordo com os estados disponíveis (cheque, jopo em andamento, cheque-mate, render-se, peças capturadas);

- Jogo::interface

```
void Jogo::interface(){
    int opcao = 0;

    cout<<"JOGO DE XADREZ"<<endl;
    cout<<"_____"<<endl;
    cout<<"Escolha uma opÃ§Ã£o!"<<endl;
    cout<<"1 - Novo Jogo"<<endl;
    cout<<"2 - Carregar Jogo"<<endl;
    cout<<"3 - Sair"<<endl;

    cin>>opcao;

    do{

        switch(opcao){
            case 1:
                play();
                return;
                break;
            case 2:
                carregarJogo();
                play();
                return;
                break;
            case 3:
                exit(1);
                return;
            default:
                cout << "OpÃ§Ã£o invÃ¡lida!" << endl;
                cin>>opcao;
        }
    }while(opcao != 3);
}
```

Simplesmente printa um menu com as opções do jogo (novo jogo, carregar jogo e sair), e redireciona para o método apropriado.

- Jogo::play

```
void Jogo::play(){
    istream &operator>> (istream & i, Tabuleiro &tab);


        while(estado != RECEBEU_XEQUE_MATE && estado != PAUSAR && estado != RENDER){
            estadoDoJogo();
            do{

                try{
                    cout<<"Digite sua jogada: ";
                    cin>>*tab;
                    estado = tab->getEstadoTabuleiro();

                }catch(int i){
                    estado = i;
                    if(estado == ENTRADA_INVALIDA){
                        cout<<"JOGADA INVÃ LIDA, DIGITE NOVAMENTE!!"<<endl;
                    }
                }
            }while(estado  == ENTRADA_INVALIDA);
            mudarVezDoJogador();
        }

        if(estado == PAUSAR){
            salvarJogo(tab->getJogadas());
        }
        estadoDoJogo();
        return;


}
```

Aqui é onde fazemos a chamada a sobrecarga feita em Tabuleiro. De acordo com o retorno da sobrecarga, verificamos a entrada do usuário. Se for uma jogada normal, a jogada prossegue; se for um comando (render ou pausar), chamamos os métodos apropriados.

- Jogo::salvarJogo

```
void Jogo::salvarJogo(queue<string> jogadas){
    ofstream arquivo;
    try{
        arquivo.open("jogadasSalvas.txt");
        if(arquivo.is_open() == false) throw "O arquivo nÃ£o foi aberto\n";
    }catch(string erro){
        cout<<erro;
        return;
    }

    while(jogadas.empty() == false){
        arquivo<<jogadas.front()<<"\n";
        jogadas.pop();
    }
    arquivo.close();
}
```

figure Aqui, simplemente abrimos o arquivo "jogadasSalvas.txt"e passamos todo o conteúdo da fila de jogadas para ele.

- Jogo::carregarJogo

```
void Jogo::carregarJogo(){

    int estadoAux;
    string linha;
    string origem;
    string destino;
    string erro = "O arquivo estÃ¡ corrompido, o jogo prosseguirÃ¡ a partir daqui!";
    ifstream arquivo;

    try{
        arquivo.open("jogadasSalvas.txt");
        if(arquivo.is_open() == false) throw "O arquivo nÃ£o existe";
    }catch(string erro){
        cout<<erro<<endl;
        exit(1);
    }
    while(getline(arquivo,linha)){
        try{

            if(linha.find(">>") == -1) throw erro;

            origem = linha.substr(0,linha.find(">>"));
            destino = linha.substr(linha.find(">>")+2,linha.size());
            if(origem.size() != 2 || destino.size() != 2) throw erro;

            if(((estadoAux = tab->checaMovimentoNoTabuleiro(origem,destino,vez)) == -1)){
                throw erro;
            }else {
                estado = estadoAux;
                if(vez == false && estado == 2) estado = 3;
                else if(vez == true && estado == 2) estado = 2;
                tab->guardarJogada(linha);
            }
```

```
        }catch(string err){
            cout<<endl<<err;
            arquivo.close();
            return;
        }

    }
    arquivo.close();
}
```

Neste método, fazemos o contrário de "salvarJogo()"; Aqui, abrimos o arquivo e lemos cada uma de suas jogadas, e aplicamos no tabuleiro. Após o termino da jogada, passamos a vez para o jogador da cor oposta, e o jogo continua. Caso haja alguma entrada inválida, o método lança um throw, indicando que o arquivo está corrompido.

4 O jogo em funcionamento

Aqui é a parte do relatório onde demonstramos o funcionameto do jogo, e como o usuário deve interagir com ele.

1. Começando o Jogo

   Quando o usuário inicia o jogo, esta é a tela que ele entra em contato:

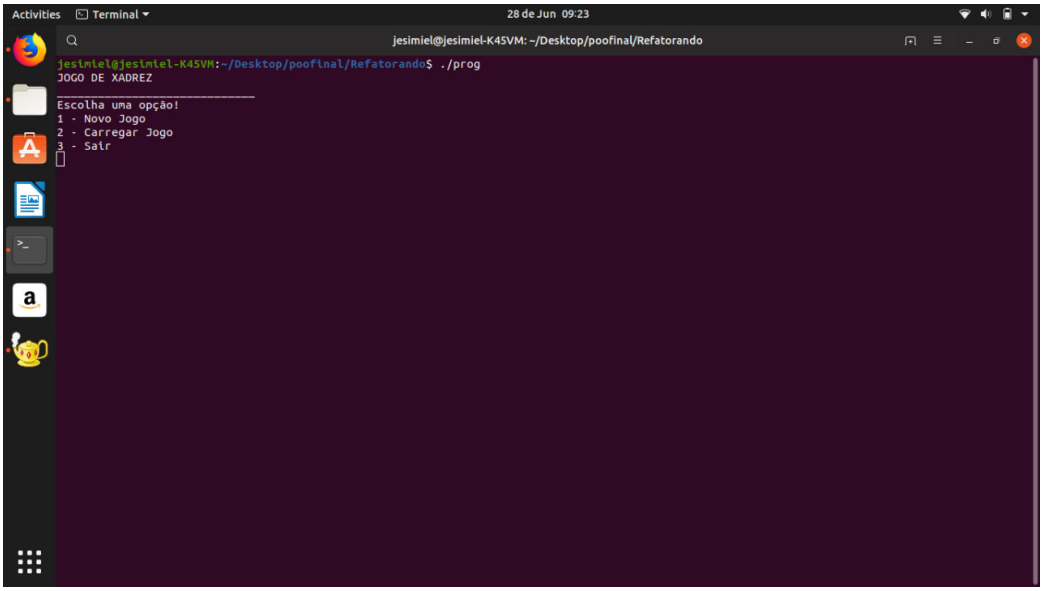

Figura 65: Tela Inicial

Neste menu;

- Caso ele escolha a opção 1, ele se depara com o tabuleiro indicando um "Novo Jogo";
- Se caso escolher 2, o jogo busca o arquivo "jogadasSalvas.txt"e retoma o estado salvo do jogo que está dentro do arquivo;
- Caso escolha 3, o jogo se encerra;

Supondo que o usuário escolheu 1;

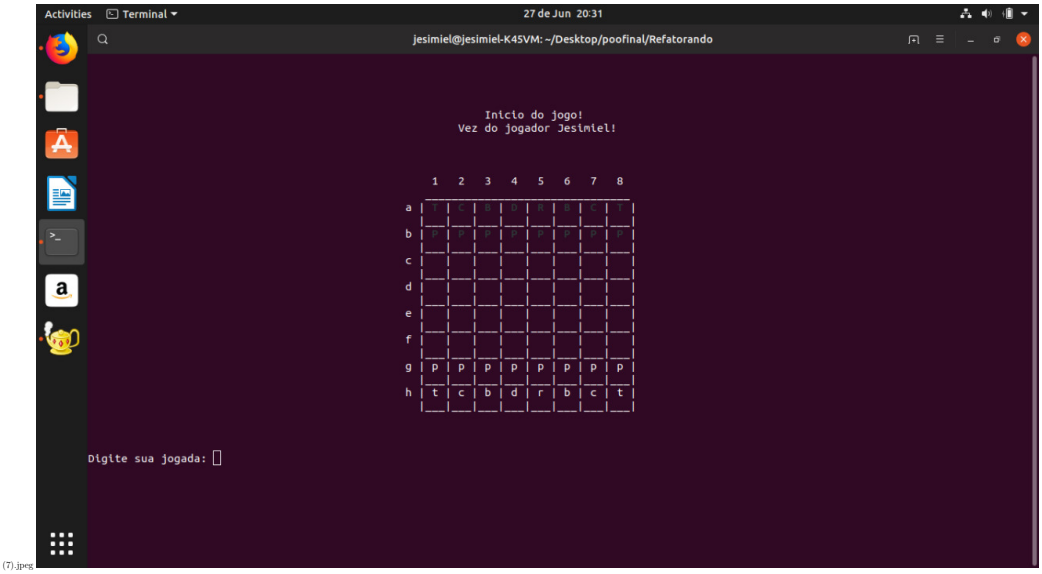

Figura 66: Novo Jogo

A partir daqui, o usuário pode inserir sua jogada na seguinte ordem:

LINHA ORIGEM | COLUNA ORIGEM » LINHA DESTINO | COLUNA DESTINO

B2»C2, por exemplo

Qualquer jogada fora disso, será considerada inválida. Caso o jogador tente movimentar alguma peça que não é dele, o mesmo erro acontece. Assim que a jogada finalizar, o jogo automaticamente notifica a mudança de jogador, e assim o jogo se segue.

2. Cheque e Cheque-Mate

   Conforme o jogo vai se seguindo, quando algum jogador entra em Cheque, o jogo o avisa:

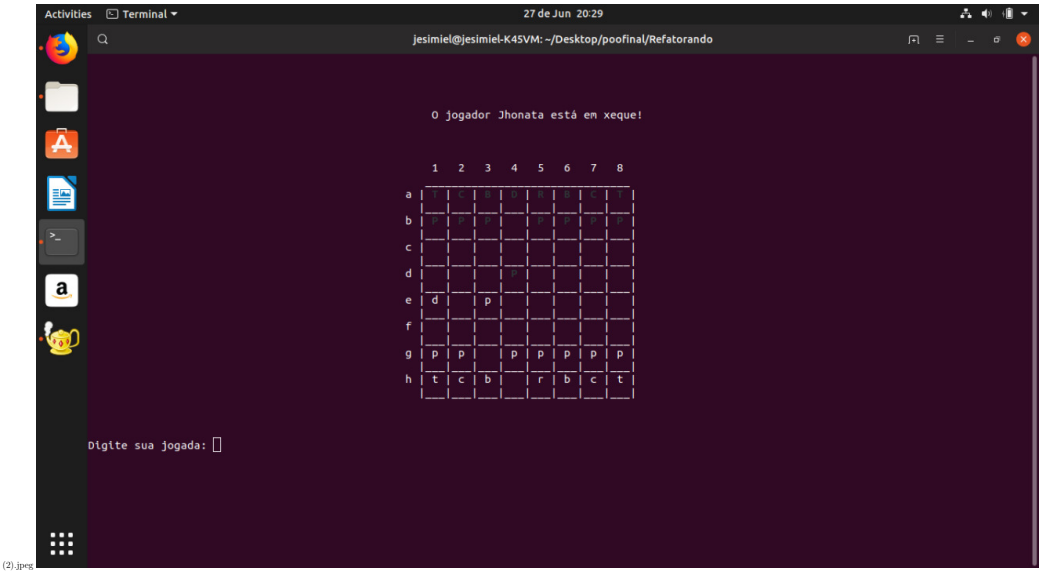

Figura 67: Cheque

O jogador oposto tem apenas uma tentativa de sair do cheque. Se caso ele conseguir sair, o jogo remove a notificação, altera a vez do jogador, e o jogo continua a se seguir normalmente. Se caso o jogador que estava em cheque não for capaz de sair da condição, o jogo anuncia o cheque-mate e se encerra:

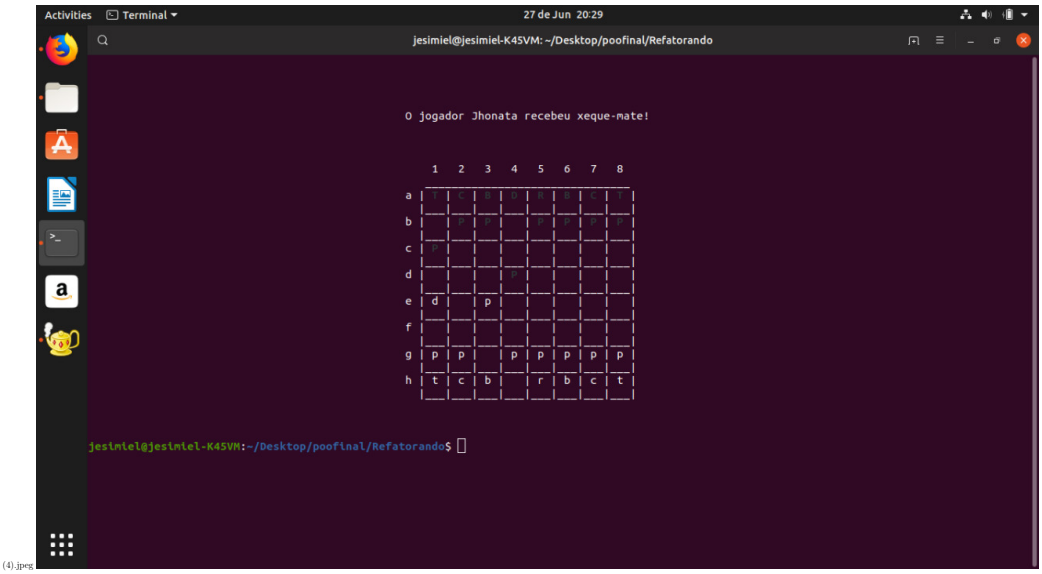

Figura 68: Cheque

3. Comandos especiais

   O jogo possibilita ao jogador digitar alguns comandos no meio do jogo:

   - "PAUSAR"

     Este comando faz com que todas as jogadas executadas sejam salvas num arquivo que é gerado pelo próprio jogo, chamado "jogadasSalvas.txt", que geralmente é gerado na pasta do jogo. Depois as jogadas são salvas, o jogo se encerra, mas ele pode ser restaurado escolhendo-se a opção 2 do menu principal:

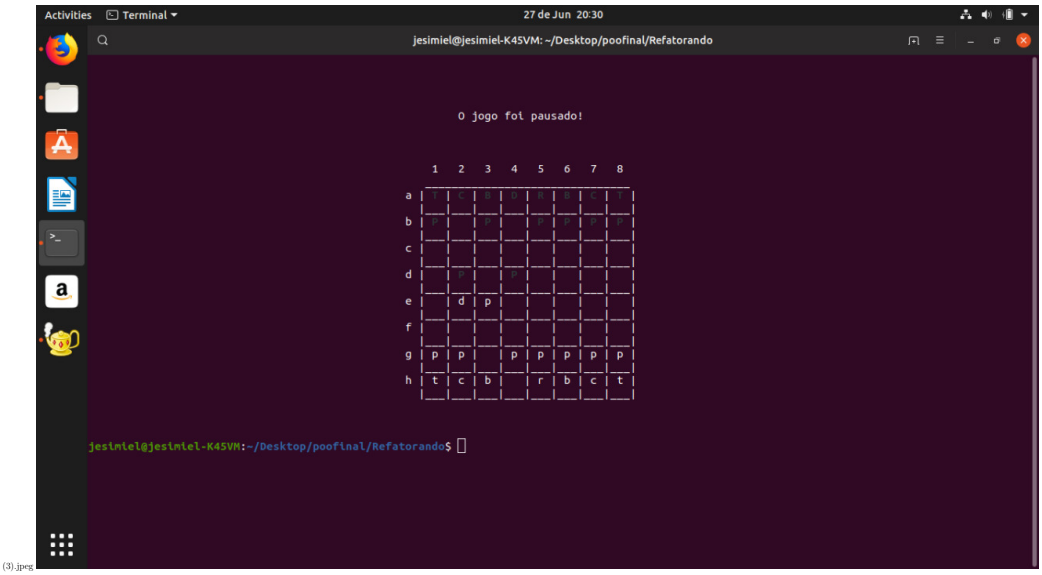

Figura 69: Pausar

   - "RENDER"Caso o jogador queira desistir do jogo, ele pode digitar "RENDER"no jogo, o que dará a vitória ao oponente e encerrará o jogo:

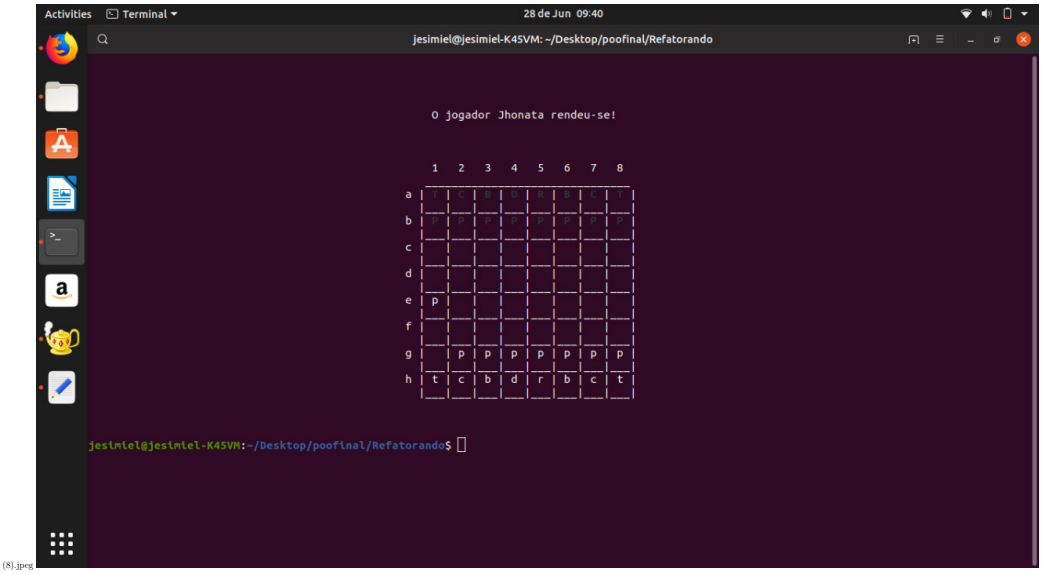

Figura 70: Render

4. Leitura de Arquivo

   O usuário tem a escolha de carregar um arquivo "jogadasSalvas.txt"com as jogadas que ele queira fazer (escolhendo a opção 2 da tela inicial). Se o modelo estiver certo, o jogo vai conseguir replicar todas as jogadas salvas no arquivo e vai voltar a executar a partir da última jogada feita.

   Agora, se caso o arquivo possuir alguma jogada inválida, o programa avisará que o arquivo está corrompido, e prosseguirá a partir da última jogada válida:

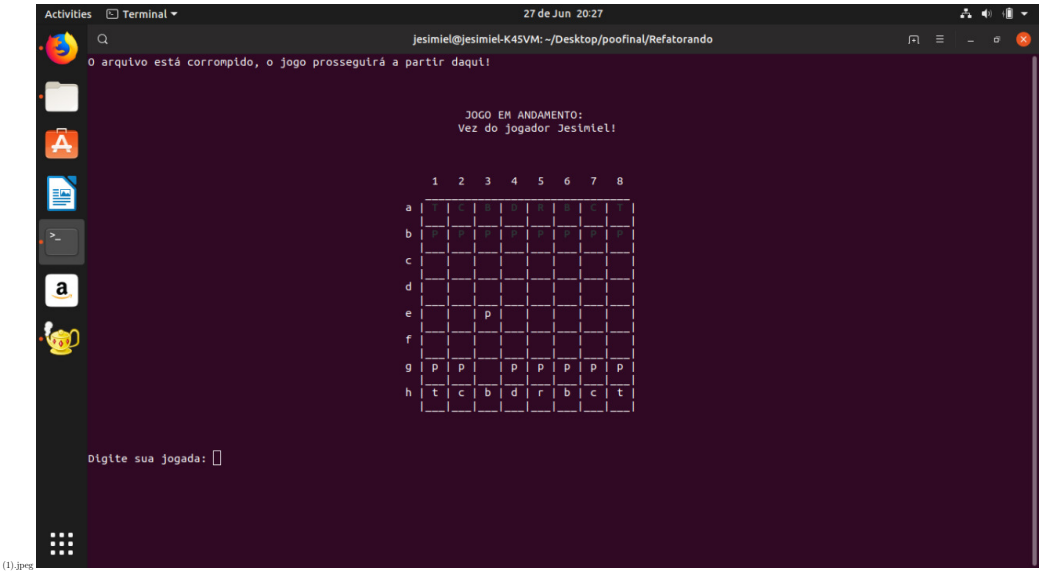

Figura 71: Arquivo Corrompido

5 Resultados

Com todo o esforço necessário, pudemos observar o poder da Programação Orientada a Obejetos, e como um projeto simples como um jogo de Xadrez pode ser complexo de se montar. Os desafios foram grandes, mas o aprendizado foi muito recompensador:

- Dificuldades Encontradas
  - Aplicar os try catch de forma apropriada;
  - Usar virtuais e static nos momentos apropriados;
  - Usar a sobrecarga de operadores. A decisão de aplicar a sobrecarga em Tabuleiro e não em Jogador foi algo que a equipe discutiu bastante;
  - Marnter o encapsulamento. Definitivamente foi algo difícil de fazer, já que o jogo de xadrez é muito dependente de fatores comuns, como peças e posições
  - Montar as regras específicas, como En Passant e cheque-mate;
- Possíveis erros
  - O jogo provavelmente não declara empate, já que não tivemos o tempo necessário para implementar;

- O cheque-mate não está corretamente implementado;
- A cada seis catacteres, o jogo notifica a jogada inválida mais de uma vez;
- O jogo não possui roque, nem promoção de peças;
- No mais, nada a acrescentar.

Agradecemos muito pela oportunidade de fazer este projeto. Foi divertido, e e grande aprendizado. Obrigado a todos que tornaram isso possível.