Universidade Federal de São Carlos – Campus Sorocaba

Gabriel Viana Teixeira - 759465

Guilherme Pereira Fantini - 759468

Tales Baltar Lopes da Silva – 759535

Xadrez

Sorocaba

2019

Sumário

1. Introdução

A primeira fase do projeto tem como objetivo a elaboração inicial de um jogo de xadrez, utilizando os conhecimentos adquiridos na disciplina de Programação Orientada à Objetos.

Para isso, realizamos o desenvolvimento das classes: Jogo, Tabuleiro, Jogador, Posição, Peça, Peão, Bispo, Torre, Cavalo, Rainha e Rei. Dessa forma, conseguimos modelar o xadrez tradicional.

O diagrama a seguir esboça como é a relação entre as classes.

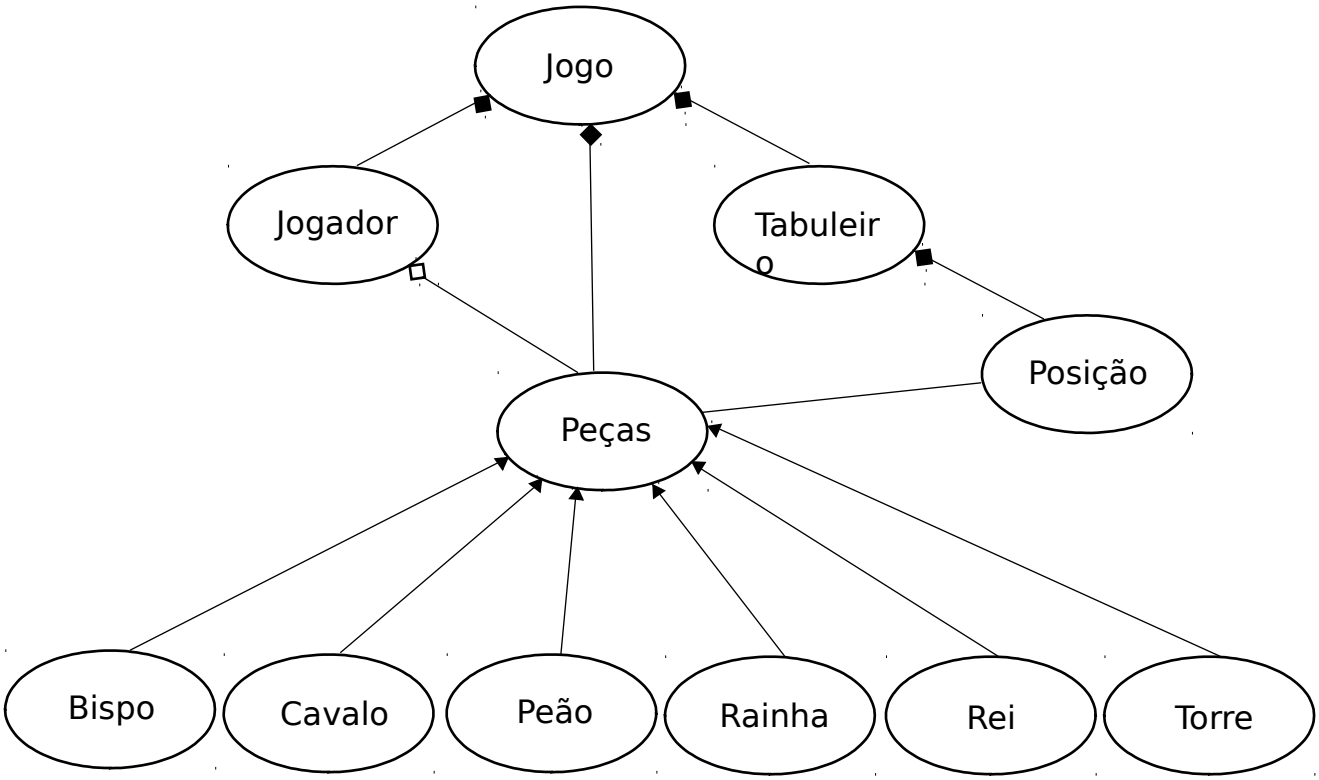


Figura 1 – Relação entre Classes

2. Funcionamento das Classes

Nessa etapa, o nosso código lê quatro inteiros do usuário, que são as coordenadas (linha e coluna) de origem e as coordenadas de destino da peça. Se for um movimento válido, a peça se desloca à posição de destino, senão ela não se desloca e imprime uma mensagem de erro.

2.1. Jogo

```
class Jogo {
    private:
        Peca *p[32];
        Jogador *j[2];
        Tabuleiro *t;
        int vezDeJogar;
        int estado;
    public:
        Jogo();
        ~Jogo();
        bool movimentarPeca(int linhaOrigem, int colunaOrigem, int linhaDestino, int colunaDestino);
        void imprimirTabuleiro();
};
```

Figura 2 – Classe Jogo (jogo.h)

A classe Jogo tem como objetivo gerenciar o jogo e controlar todas as operações necessárias para o funcionamento pleno do jogo.

Para isso, definimos a classe Jogo da Figura 1, no qual possui os atributos: um tabuleiro, o conjunto de 32 peças, os dois jogadores, a variável responsável por guardar o estado do jogo e outra que é responsável por guardar a vez do jogador.

Além disso, também tem os métodos essenciais para o funcionamento do jogo. O construtor inicializa as peças, o tabuleiro e o jogador, também separa as peças brancas das pretas, determina o estado inicial do jogo e o jogador que fará o primeiro movimento.

O método movimentarPeca checa se é um movimento válido, se for verdadeiro muda a vez de jogador. Já o imprimirTabuleiro, chama o método que desenha o tabuleiro.

2.2. Jogador

```
class Jogador {
    private:
        int id;
        Peca *peca[16];
    public:
        Jogador(int jogador, Peca *p[16]);
};
```

Figura 3 – Classe Jogador (jogador.h)

Nessa fase, a classe Jogador é responsável por guardar o id do jogador e o conjunto de peças de cada usuário.

Por conta disso, definimos a classe Jogador, como pode ser visto na figura 2, no qual utilizamos dois atributos, um inteiro que guarda o id do jogador, e um array de 16 posições que cada elemento é um endereço de um objetivo tipo Peca.

Juntamente com os atributos, temos o construtor da classe, que atribui ao id o valor do jogador e, também, atribui uma peça para cada elemento do vetor de peças.

2.3. Tabuleiro

```
class Tabuleiro {
    private:
        Posicao *posicao[8][8];
        bool movimentoPermitido(int linhaOrigem, int colunaOrigem,
                                int linhaDestino, int colunaDestino, int vezDeJogar);
        void desenhaEspacoB();
    public:
        Tabuleiro(Peca *pecas[32]);
        ~Tabuleiro();
        void desenharTabuleiro(); // Desenhar tabuleiro
        bool movimentarPeca(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestino, int vezDeJogar);
};
```

Figura 4 – Classe Tabuleiro (tabuleiro.h)

A classe Tabuleiro é responsável pela inicialização, impressão e manutenção do tabuleiro, também tem o papel de verificar os movimentos solicitados pelos jogadores.

Essa classe é formada por um atributo, posicao, que é um vetor bidimensional de 64 endereços de objetos do tipo Posicao.

Paralelamente, também temos seis métodos.

- movimentoPermitido, que verifica se o movimento não ultrapassa os limites do tabuleiro, também verifica se a movimentação é condizente com a peça da posição;
- desenhaEspacoB, que é um método auxiliar para imprimir espaços em branco em posições que não possuem peças;
- desenharTabuleiro, responsável pela impressão do tabuleiro na tela;
- movimentarPeca, método que, após a resposta do movimentoPermitido, atualiza a posição da peça, caso seja um movimento válido;
- Construtor, responsável por alocar um objeto do tipo posição para cada elemento do vetor posicao, já colocando cada peça em seu respectivo local inicial;
- Destrutor, responsável por desalocar cada um dos objetos do vetor posicao.

2.4. Posição

```
class Posicao {
    private:
        int linha;
        int coluna;
        Peca *peca;

    public:
        Posicao(int l, char c, Peca *p);
        void atualizarPos(Peca *p); // Coloca uma nova peça na posição
        Peca* consultarPos(); // Retornar a peça que esta na posicão
        bool temPeca(); // Retornar se tem peça na posiçao
        int getLinha();
        int getColuna();
};
```

Figura 5 – Classe Posicao (posicao.h)

A classe Posicao tem como objetivo gerenciar cada posição do tabuleiro, ou seja, ela verifica se tem peça na posição, atualiza a peça que está na posição e retorna qual peça está na posição.

Essa classe é formada por três atributos, duas variáveis do tipo int que guardam a linha e a coluna da posição e também o endereço da peça que está ocupando a posição.

Paralelamente, temos seis métodos:

- Construtor, que ao receber os parâmetros, atribui para cada atributo o seu valor respectivo;
- atualizarPos, que é um método auxiliar para imprimir espaços em branco em posições que não possuem peças;
- consultarPos, responsável por retornar qual peça está na posição;
- temPeca, método que retorna se tem peça na posição;
- getLinha, responsável por retornar qual é a linha da posição;
- getColuna, método que retorna qual é a coluna da posição.

2.5. Peça

```
class Peca {
    private:
        int jogador;
        char peca;
        Rei *k;
        Rainha *r;
        Bispo *b;
        Cavalo *c;
        Torre *t;
        Peao *p;

    public:
        Peca(char tipo);
        ~Peca();
        void desenharPeca();
        bool movimentoPermitidoPeca(int linhaOrigem, int colunaOrigem,
                                    int linhaDestino, int colunaDestino);
        void alterarJogador(int id);
        int consultarJogador();
};
```

Figura 6 – Classe Peca (peca.h)

A classe peça foi feita como uma adaptação para que as diferentes peças do jogo (rei, rainha, bispo, cavalo, torre, peão) possam ser usadas e conectar o código. Ela é responsável por alocar qualquer peça que o jogo requisite e desalocá-las no final, também é responsável por chamar os métodos particulares da peça que for requisitada.

Os métodos utilizados são:

- movimentoPermitidoPeca, verifica o tipo da peça, dessa forma é chamado o método da peça para efetuar a checagem do movimento.
- alterarJogador, um método que recebe o id do jogador e atribui ao atributo jogador;
- desenharPeca, responsável por verificar o tipo da peça e chama o método desenhar daquela peça, se for encontrado um endereço válido;
- consultarJogador, método que retorna o atributo jogador;
- Construtor, utilizado para alocar dinamicamente as peças de acordo com o tipo da peça;
- Destrutor, utilizado para desalocar as peças.

Ela contém um atributo que é o endereço da peça que ela representa e um atributo para identificar qual jogador ela pertence (se ela é branca ou preta), há também um método que retorna a qual jogador a peça pertence.

As classes Rei, Rainha, Bispo, Cavalo, Torre e Peao funcionam de forma similar, todas são encarregadas de desenhar a peça, checar a adequação do movimento e, também, verificar se a peça foi capturada. Portanto, elas têm atributos e métodos

semelhantes, algumas podem ter métodos ou atributos a mais, ou seja, elas diferem somente em como manipulam os atributos e operam os métodos.

O método desenha imprime a letra correspondente de cada peça (veja Tabela 1), sendo em maiúsculo para o jogador 1 e em minúsculo para o jogador 2. Para checar se a peça foi capturada, temos dois métodos, *capturar*, que transforma o atributo *capturado* em *true* e, *foiCapturado*, que retorna o valor do atributo *capturado*.

Representação das Peças	
Rei	K ou k
Rainha	R ou r
Bispo	B ou b
Cavalo	C ou c
Torre	T ou t
Peão	P ou p

Tabela 1 – Representação das Peças

2.5.1. Rei

```
class Rei {
    private:
        bool capturado;
        bool mesmaPosicao(int linhaOrigem, int colunaOrigem,
                          int linhaDestino, int colunaDestino);
    public:
        Rei();
        void desenha(int jogador);
        void capturar();
        bool foiCapturado();
        bool checaMovimento(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestino);
};
```

Figura 7 – Classe Rei (rei.h)

A classe Rei é responsável por gerenciar o comportamento da peça Rei. Ela funciona de forma semelhante às outras peças, difere na forma como checa o movimento. Para verificar o movimento, pegamos o módulo da diferença entre a linha destino e linha origem e, também, o módulo da diferença entre a coluna destino e coluna origem, o resultado desses módulos tem que ser menor ou igual a 1.

Matematicamente,

| linhaDestino – linhaOrigem | < = 1 e | colunaDestino – colunaOrigem | <= 1.

2.5.2. Cavalo

```
class Cavalo {
    private:
        bool capturado;
        bool mesmaPosicao(int linhaOrigem, int colunaOrigem,
                          int linhaDestino, int colunaDestino);
    public:
        Cavalo();
        void desenha(int jogador);
        void capturar();
        bool foiCapturado();
        bool checaMovimento(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestino);
};
```

Figura 8 -Classe Cavalo (cavalo.h)

A classe Cavalo é responsável por gerenciar o comportamento da peça Cavalo. Ela funciona de forma similar às outras peças, diferencia-se na forma como examina o movimento.

O cavalo se movimenta em L, ou seja, temos duas possibilidades de verificação, primeiro, se o módulo da diferença entre as colunas destino e origem for igual a 2, então o módulo da diferença entre as linhas destino e origem tem que ser igual a 1. Além dessa possibilidade, também se o módulo da diferença entre as colunas destino e origem for igual a 1, então o módulo da diferença entre as linhas destino e origem tem que ser igual a 2. Se uma dessas possibilidades for verdadeira, então é um movimento válido.

Matematicamente,

| linhaDestino – linhaOrigem | = 2 e | colunaDestino – colunaOrigem | = 1 ou

| linhaDestino – linhaOrigem | = 1 e | colunaDestino – colunaOrigem | = 2

.

2.5.3. Bispo

```
class Bispo {
    private:
        bool capturado;
        bool mesmaPosicao(int linhaOrigem, int colunaOrigem,
                          int linhaDestino, int colunaDestino);
    public:
        Bispo();
        void desenha(int jogador);
        void capturar();
        bool foiCapturado();
        bool checaMovimento(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestino);
};
```

Figura 9 - Classe Bispo (bispo.h)

A classe Bispo é responsável por gerenciar o comportamento da peça Bispo. Ela funciona de forma similar às outras peças, diferencia-se na forma como examina o movimento.

O bispo se move diagonalmente pelo tabuleiro. Para isso, o módulo da diferença entre as linhas destino e origem tem que ser igual ao módulo da diferença entre as colunas destino e origem.

Matematicamente,

| linhaDestino – linhaOrigem | = | colunaDestino – colunaOrigem |.

Se a fórmula anterior for verdadeiro, então o movimento é válido, senão é um movimento inválido.

2.5.4. Torre

```
class Torre {
    private:
        bool capturado;
        bool mesmaPosicao(int linhaOrigem, int colunaOrigem,
                          int linhaDestino, int colunaDestino);
    public:
        Torre();
        void desenha(int jogador);
        void capturar();
        bool foiCapturado();
        bool checaMovimento(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestino);
};
```

Figura 10 – Classe Torre (torre.h)

A classe Torre é responsável por gerenciar o comportamento da peça Torre. Ela funciona de forma similar às outras peças, diferencia-se na forma como examina o movimento.

A torre se move em linhas retas pelas linhas e colunas do tabuleiro, dessa forma sempre uma das coordenadas (linha ou coluna) da posição destino tem que ser igual a uma das coordenadas da posição de origem, ou seja.

LinhaOrigem = LinhaDestino ou ColunaOrigem = ColunaDestino.

2.5.5. Rainha

```
class Rainha {
    private:
        bool capturado;
        bool mesmaPosicao(int linhaOrigem, int colunaOrigem,
                          int linhaDestino, int colunaDestinono);
    public:
        Rainha();
        void desenha(int jogador);
        void capturar();
        bool foiCapturado();
        bool checaMovimento(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestinono);
};
```

Figura 11 - Classe Rainha (rainha.h)

A classe Rainha é responsável por gerenciar o comportamento da peça Rainha. Ela funciona de forma similar às outras peças, diferencia-se na forma como examina o movimento.

Como a rainha tem o movimento do bispo e da torre combinados, então a forma de verificação do movimento utiliza o raciocínio dessas outras peças, ou seja, a Rainha tem que respeitar a fórmula a seguir.

| linhaDestino – linhaOrigem | = | colunaDestino – colunaOrigem | ou

LinhaOrigem = LinhaDestino ou ColunaOrigem = ColunaDestino.

2.5.6. Peão

```
class Peao {
    private:
        int movimentacoes;
        bool capturado;
        bool mesmaPosicao(int linhaOrigem, int colunaOrigem,
                          int linhaDestino, int colunaDestino);
    public:
        Peao();
        void desenha(int jogador);
        void capturar();
        bool foiCapturado();
        bool checaMovimento(int linhaOrigem, int colunaOrigem,
                            int linhaDestino, int colunaDestino,
                            bool temPeca=false);
};
```

Figura 12 - Classe Peao (peao.h)

A classe Peao é responsável por gerenciar o comportamento da peça Peao. Ela funciona de forma similar às outras peças, diferencia-se na forma como examina o

movimento. Além disso, como a primeira movimentação do peão é diferente da movimentação durante o jogo, adicionamos um atributo que conta a quantidade de movimentos feitos pelo peão.

O peão tem três movimentos, deslocar uma posição para frente, deslocar duas posições para frente, se for o movimento inicial e, também, capturar pela diagonal frontal. Para isso, verificamos se a coluna de origem for igual a coluna destino e se a linha de destino é igual a linha de origem acrescida de 1, em movimentos simples, ou se a linha de destino é igual a linha de origem acrescida de 2, na possibilidade do movimento inicial do peão.

Além disso, para movimentos diagonais, se o módulo da diferença entre a coluna de origem e a coluna de destino for igual a 1, e a linha destino for igual a linha origem acrescida de 1, então também retorna como um movimento válido. Caso não respeite essas condições, então é um movimento inválido.

3. Testes

Para verificar o funcionamento das classes do projeto, realizamos testes com as possibilidades de escolha do usuário.

De modo geral, para movimentar cada peça, Jogo chama Tabuleiro pelo método *movimentarPeca*, que utiliza o método privado *movimentoPermitido*, que faz a chamada de dois métodos, *consultarPos* da Posicao e *consultaJogador* da Peca, que retornam a peça que está na posição e qual jogador fez a jogada. Assim, podemos verificar se é a vez do jogador e qual peça está na posição de origem, se não tiver peça, já não é um movimento permitido.

Além disso, é chamado o método *movimentoPermitidoPeca* da classe Peca que, de acordo com o tipo da peça, chama o método *checaMovimento* da respectiva peça, se for um movimento válido, retorna *TRUE*, senão retorna *FALSE*. Voltando ao *movimentoPermitido*, se após todas as verificações não acusar nada inválido, o movimento é considerado válido e retorna *TRUE*, senão retorna *FALSE*.

Após toda a verificação, se for considerado um movimento permitido, é chamado o método *atualizaPos*, dessa forma colocamos a peça no destino e a retiramos da origem.

3.1. Tabuleiro

O teste tem como objetivo verificar se o tabuleiro é desenhado com todas as peças nos locais de início.

Nesse caso, é usado todas as classes do jogo, pois a classe Jogo chama o método *desenhaTabuleiro* da classe Tabuleiro, que utiliza as classes Posicao (método *consultarPos*) e Peca (método *desenharPeca*) para verificar em quais posições tem peça e desenha as peças juntamente com o resto do tabuleiro.

Para desenhar as peças, a classe Peca chama o método *desenha* de cada peça para elas serem impressas na tela. O resultado pode ser visto na figura a seguir.

3.2. Rei

Os testes a seguir são para verificar o funcionamento da movimentação do rei, a forma como essa operação está sendo executada está descrita no começo desse capítulo.

Os primeiros testes da movimentação do rei que realizamos foram três movimentos válidos, a figura 14 demonstra o teste do rei indo uma casa para frente, na figura 15, ele está indo uma casa para trás, já na figura 16, o rei está se movimentando diagonalmente, como são movimentos condizentes com a peça, eles foram executados.

Por outro lado, na figura 17, o rei está tentando ir a uma posição maior do que uma casa, portanto foi retornado a mensagem de movimento inválido.

3.3. Rainha

Os testes a seguir são para verificar o funcionamento da movimentação da rainha, a forma como essa operação está sendo executada está descrita no começo desse capítulo.

Os primeiros testes da movimentação da rainha que realizamos foram três movimentos válidos, a figura 18 demonstra o teste da rainha indo para frente, na figura 19, ele está indo para a diagonal, já na figura 20, o rei está se movimentando para o lado, como são movimentos condizentes com a peça, eles foram executados.

Por outro lado, na figura 21, o rei está tentando realizar o movimento do cavalo, portanto foi retornado a mensagem de movimento inválido.

3.4. Torre

Os testes a seguir são para verificar o funcionamento da movimentação da torre, a forma como essa operação está sendo executada está descrita no começo desse capítulo.

Os primeiros testes da movimentação da torre que realizamos foram dois movimentos válidos, a figura 22 demonstra o teste da torre indo para frente, já na figura

23, a torre está se movimentando para o lado, como são movimentos condizentes com a peça, eles foram executados.

Por outro lado, na figura 24, a torre está tentando andar diagonalmente, portanto foi retornado a mensagem de movimento inválido.

3.5. Bispo

Os testes a seguir são para verificar o funcionamento da movimentação do bispo, a forma como essa operação está sendo executada está descrita no começo desse capítulo.

Os primeiros testes da movimentação do bispo que realizamos foram um movimento válido, a figura 25 demonstra o teste do bispo indo para a diagonal, já na figura 26, o bispo está tentando andar em linha reta, portanto foi retornado a mensagem de movimento inválido.

3.6. Cavalo

Os testes a seguir são para verificar o funcionamento da movimentação do cavalo, a forma como essa operação está sendo executada está descrita no começo desse capítulo.

Os primeiros testes da movimentação do cavalo que realizamos foram dois movimentos válidos, a figura 27 demonstra o teste do cavalo fazendo o movimento em L, deslocando duas linhas para baixo e uma coluna para o lado, já na figura 28, o cavalo está realizando o outro modo do movimento, deslocando uma linha para cima e duas colunas para o lado, como são movimentos condizentes com a peça, eles foram executados.

Por outro lado, na figura 29, o cavalo está tentando andar em linha reta, portanto foi retornado a mensagem de movimento inválido.

3.7. Peão

Os testes a seguir são para verificar o funcionamento da movimentação do peão, a forma como essa operação está sendo executada está descrita no começo desse capítulo.

Os primeiros testes da movimentação do peão que realizamos foram dois movimentos válidos, a figura 30 demonstra o peão indo uma casa para frente, já na figura

31, o peão está se movimentando duas casas para frente, como são movimentos condizentes com a peça, eles foram executados.

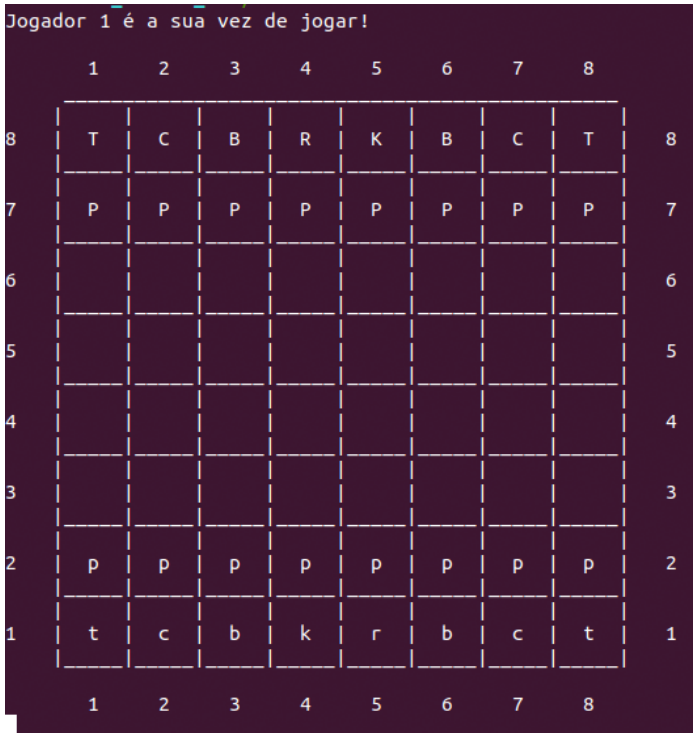
Por outro lado, na figura 32, o peão está tentando andar duas casas, mas não é o primeiro movimento dele, então foi retornado a mensagem de movimento inválido.

4. Conclusão

Durante a elaboração dessa fase, tivermos dificuldades em implementar a classe Posicao sem a classe Peca, portanto nós desenvolvemos uma classe Peca para que o xadrez ficasse funcional. Em relação ao funcionamento das peças, os principais itens restantes que precisamos desenvolver é a captura de peças e, como pode ser visto na imagem 18, fazer com que a peça não se desloque se tiver alguma peça no caminho.

Em suma na primeira fase, percebemos como é iniciado, elaborado, desenvolvido e testado um programa mais extenso, e como a orientação à objetos facilitou na maneira como modelamos um jogo real com diversas possibilidades, como é o xadrez.

Lista de Figuras



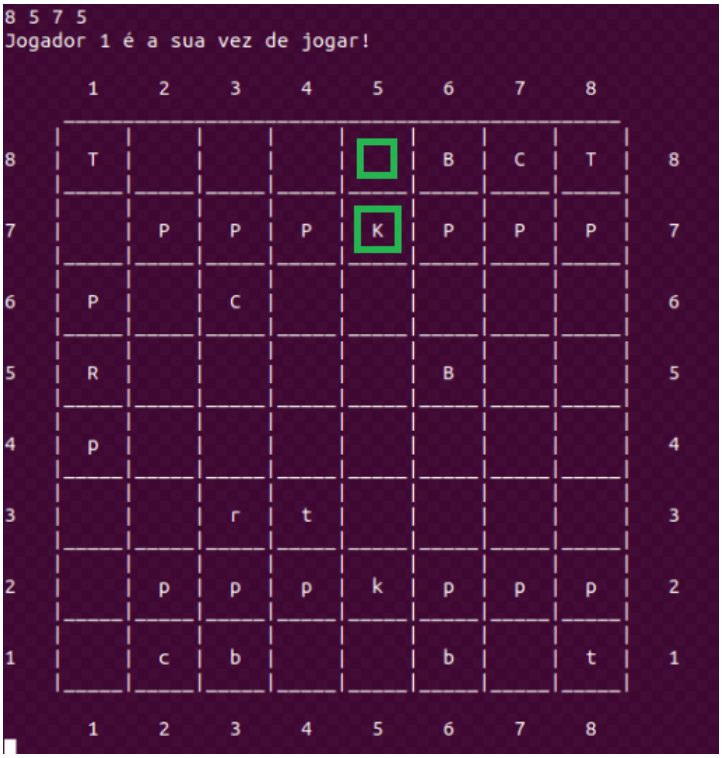


Figura 15 – Movimento Rei Trás

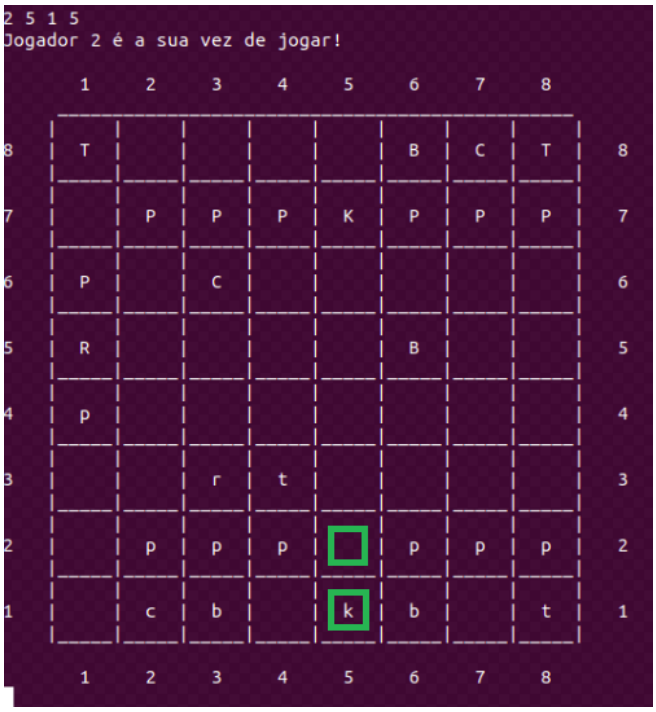


Figura 16 – Movimento Rei Diagonal

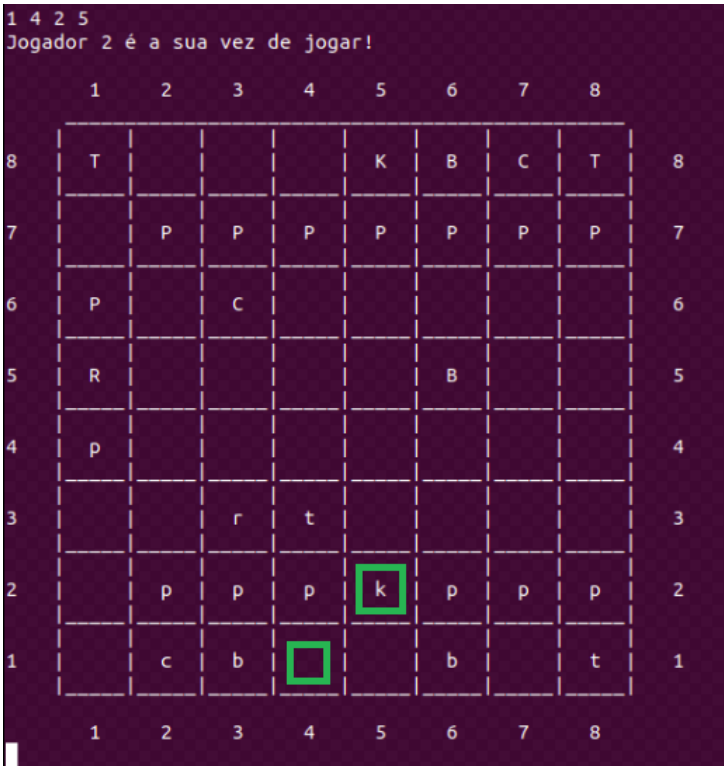


Figura 17 – Movimento Inválido do Rei

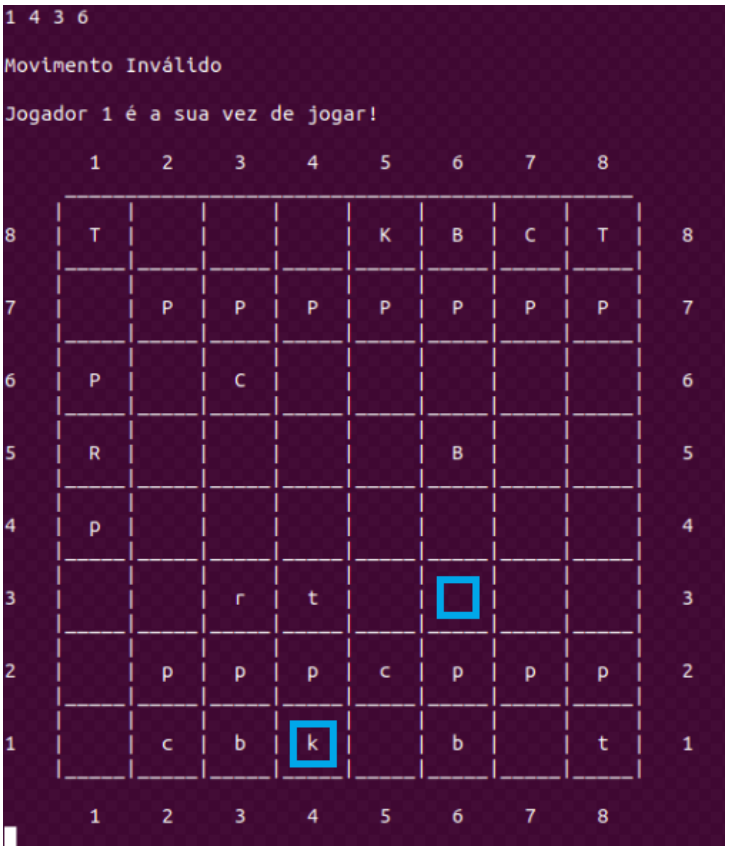


Figura 18 – Movimento Rainha Frente

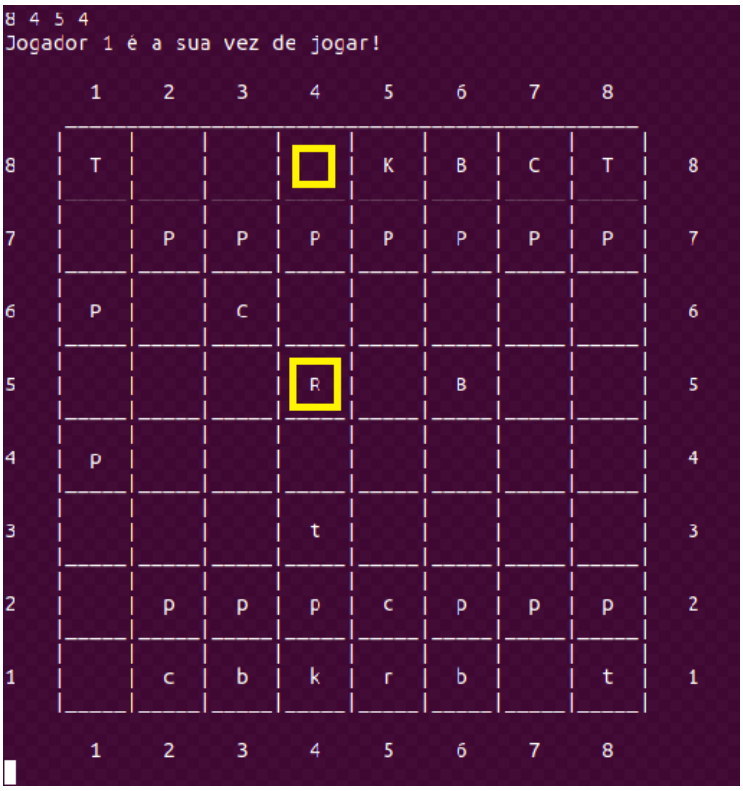


Figura 19 – Movimento Rainha Diagonal

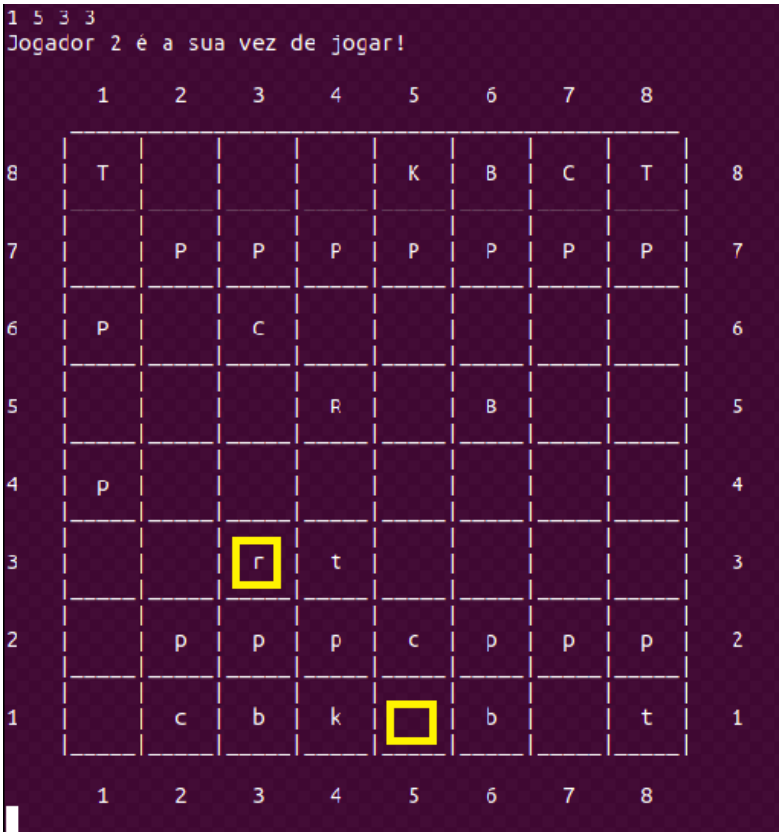


Figura 20 – Movimento Rainha Lado

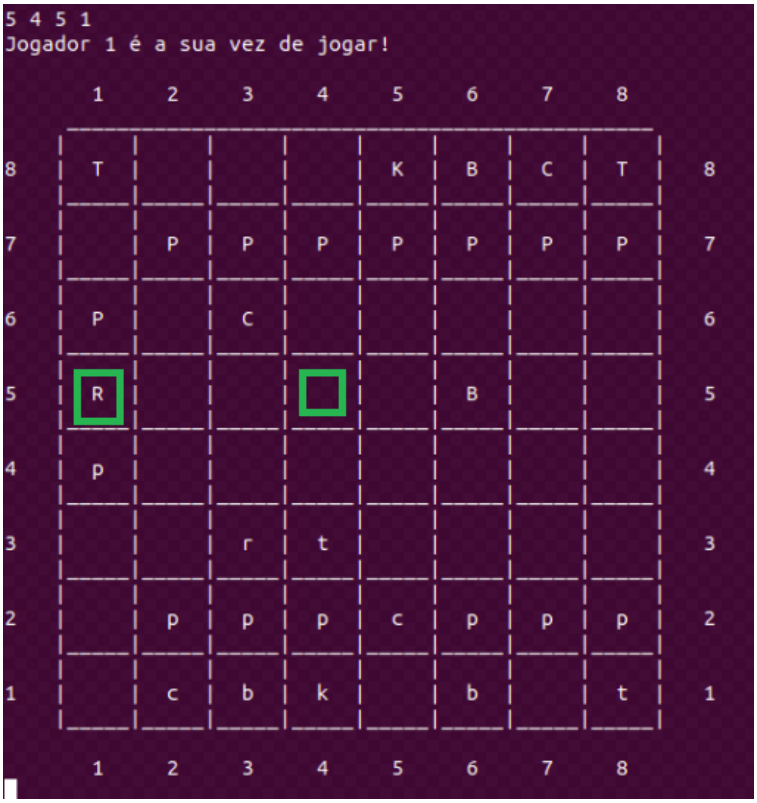


Figura 21 – Movimento Inválido da Rainha

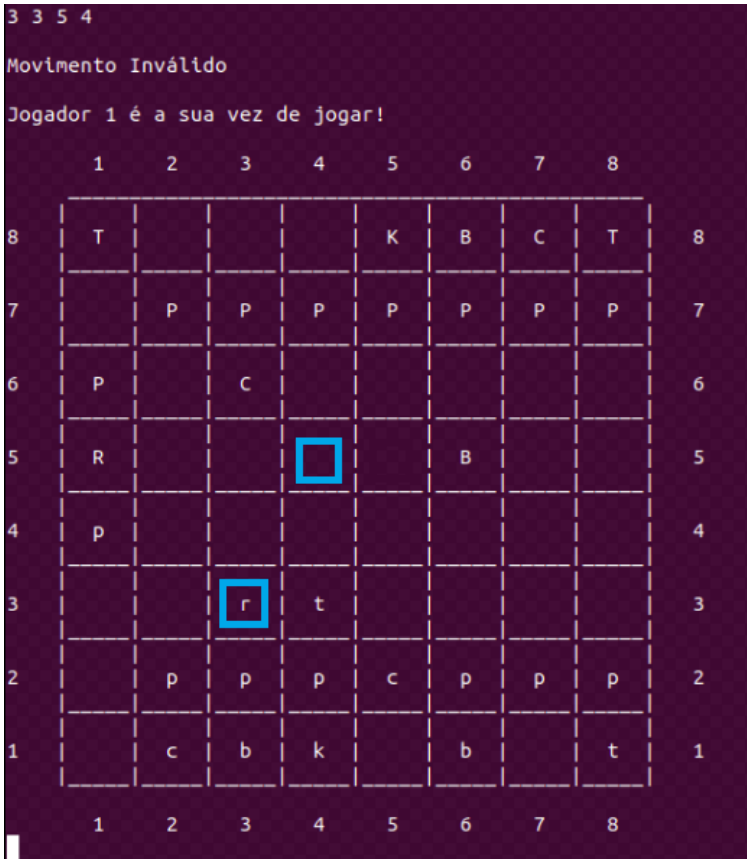


Figura 22 – Movimento Torre Frente

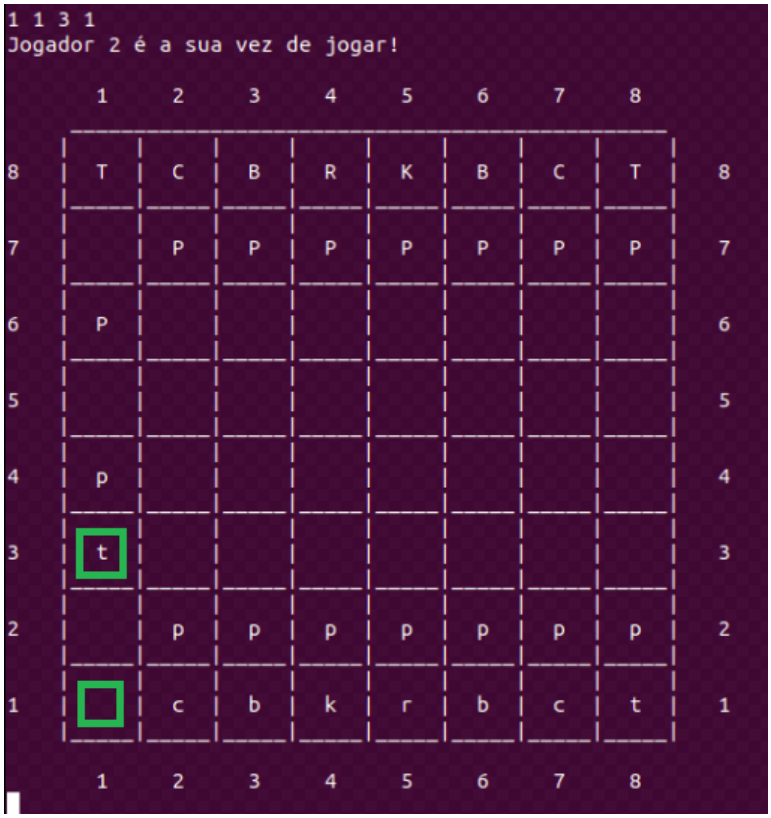


Figura 23 – Movimento Torre Lado

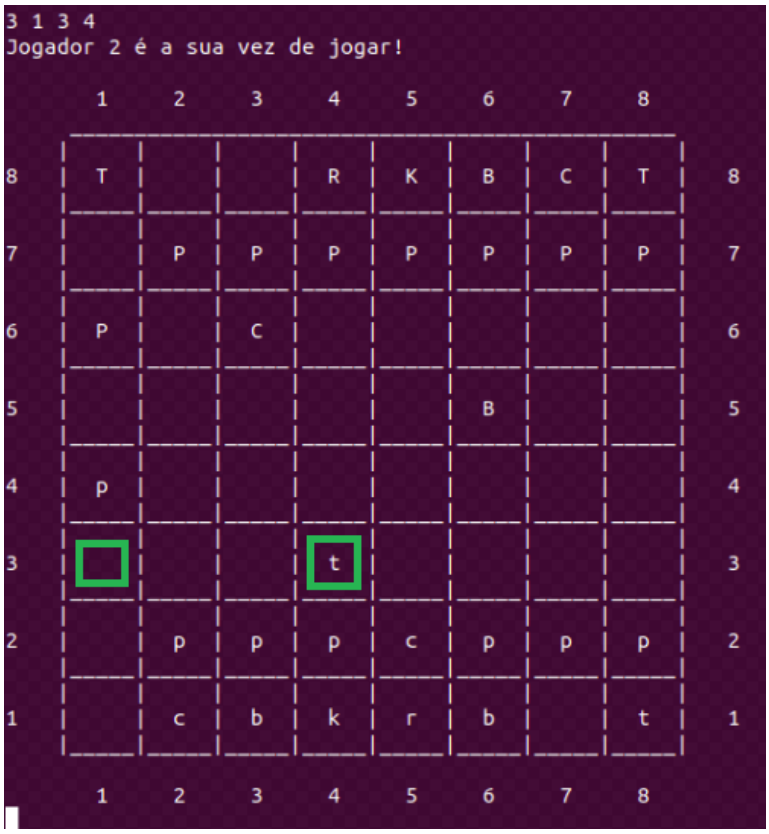


Figura 24 – Movimento Inválido da Torre

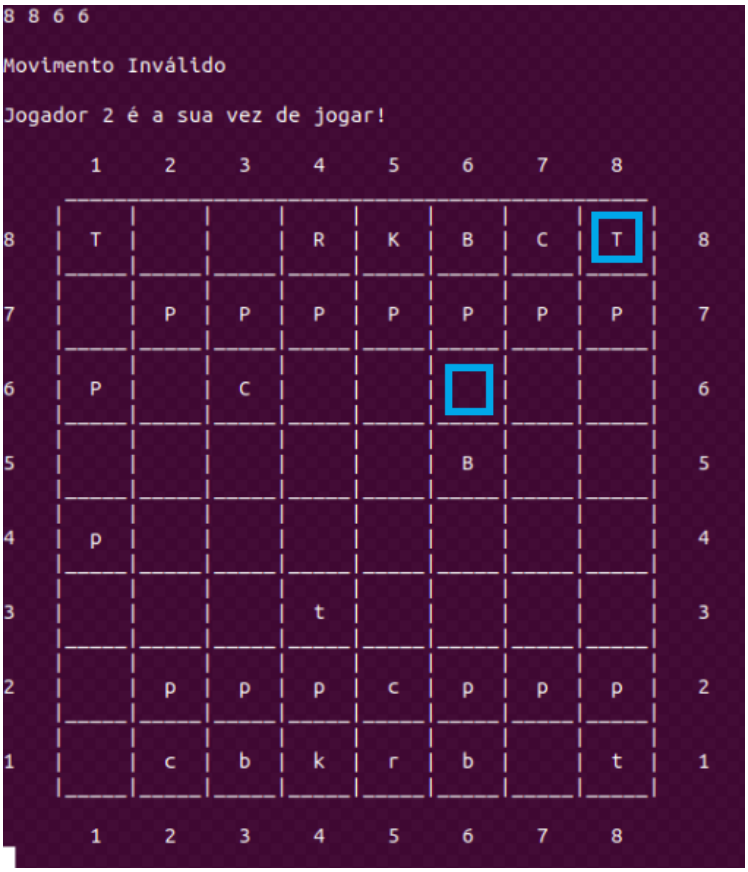


Figura 25 – Movimento Bispo Diagonal

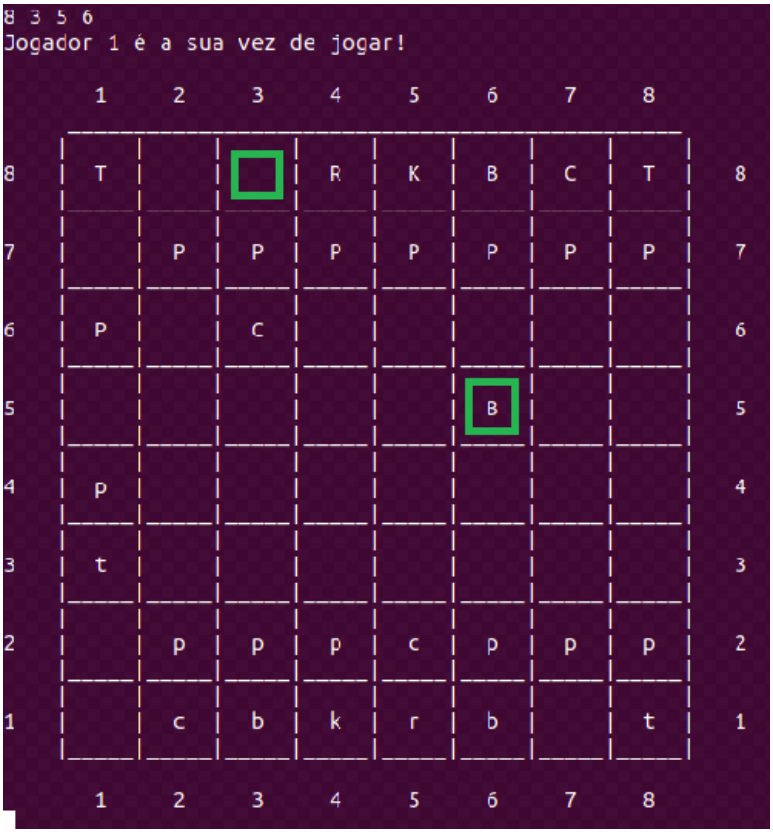


Figura 26 – Movimento Inválido do Bispo

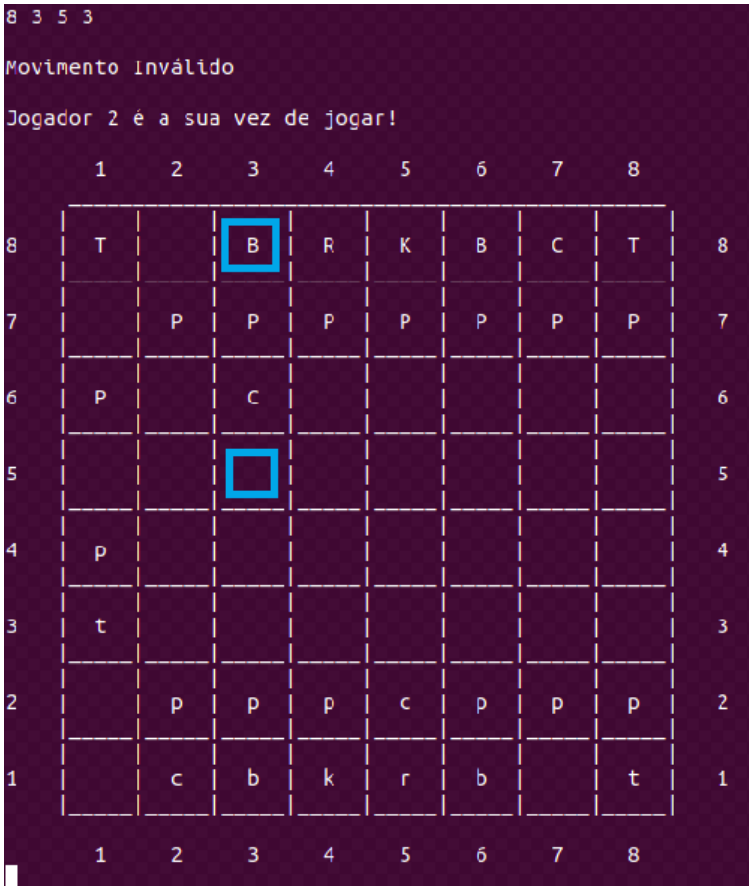


Figura 27 – Movimento do Cavalo Modo 1

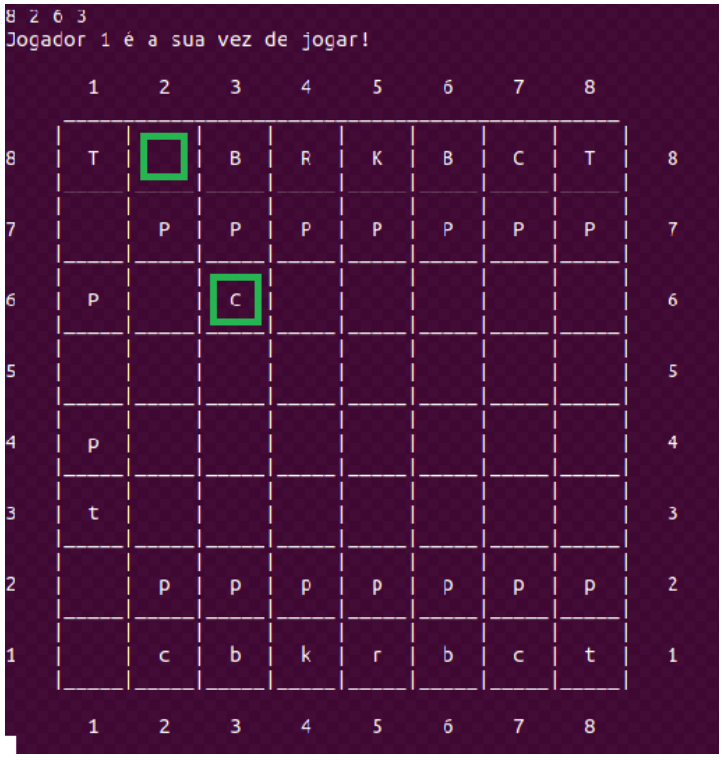


Figura 28 – Movimento do Cavalo Modo 2

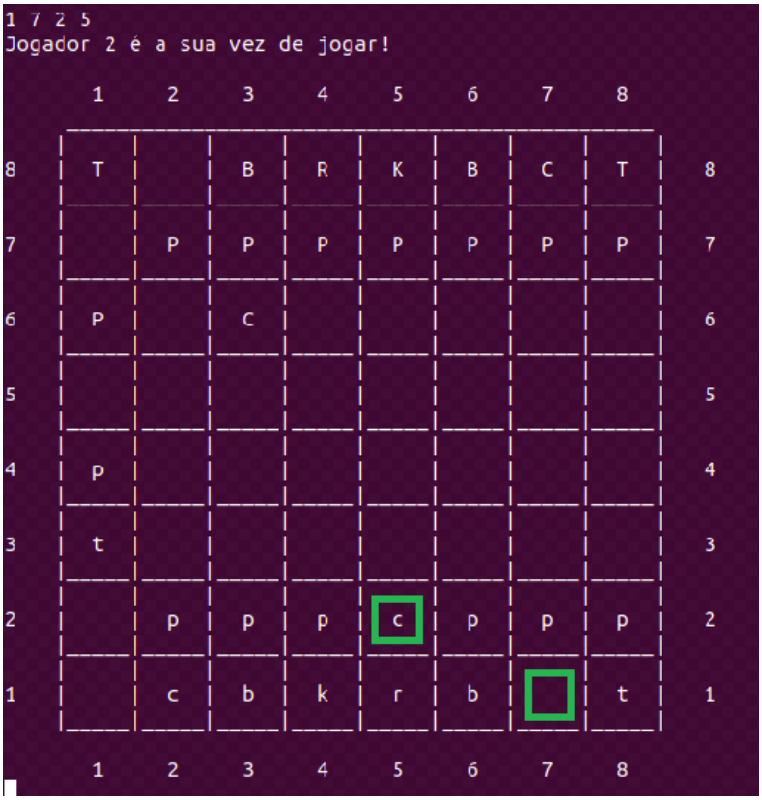


Figura 29 – Movimento Inválido do Cavalo

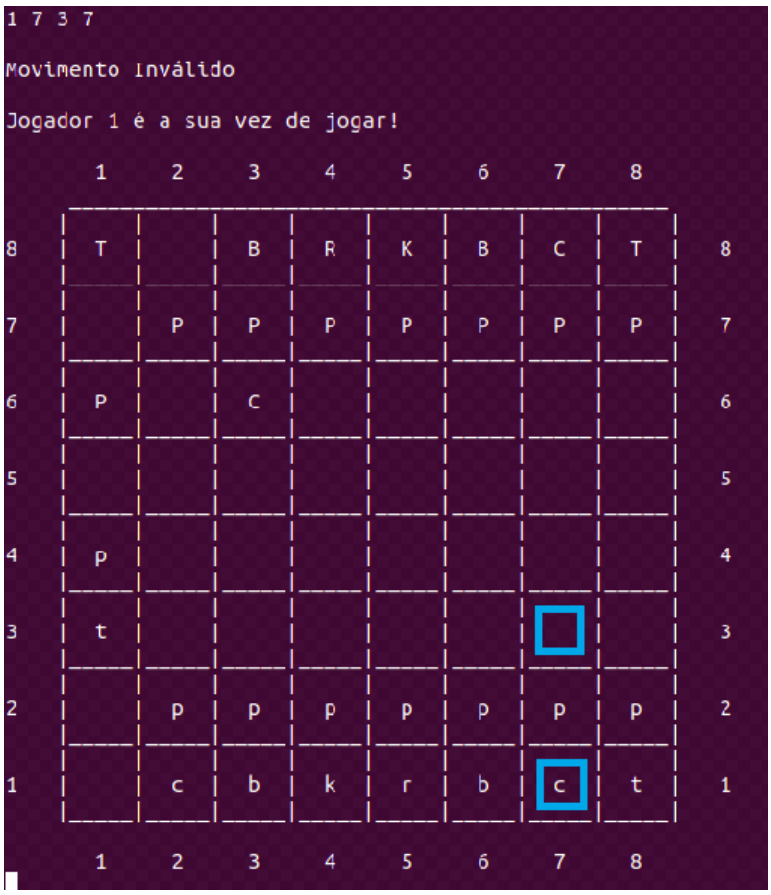


Figura 30 – Movimento Peão Uma Casa

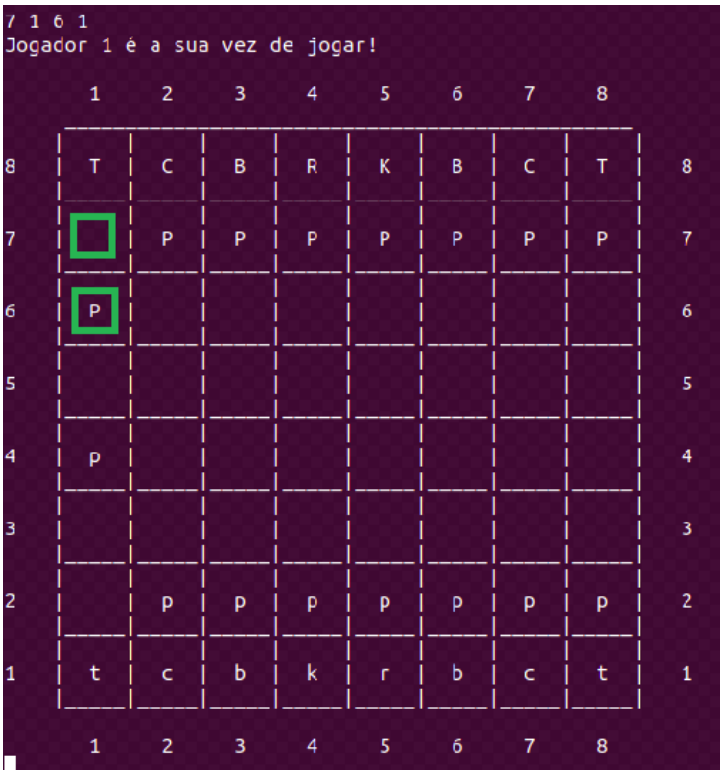


Figura 31 – Movimento Peão Duas Casas

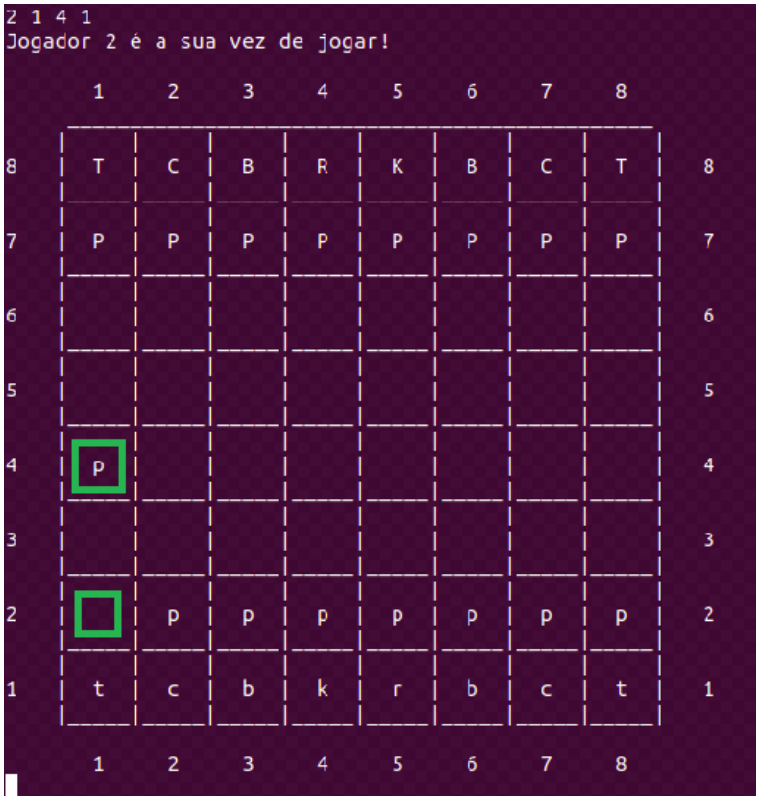


Figura 32 – Movimento Inválido do Peão

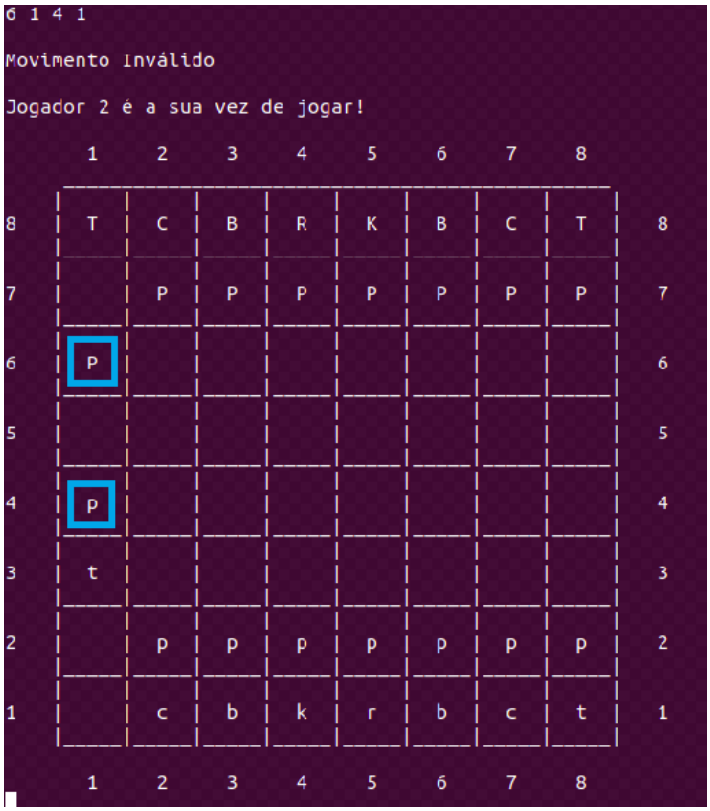
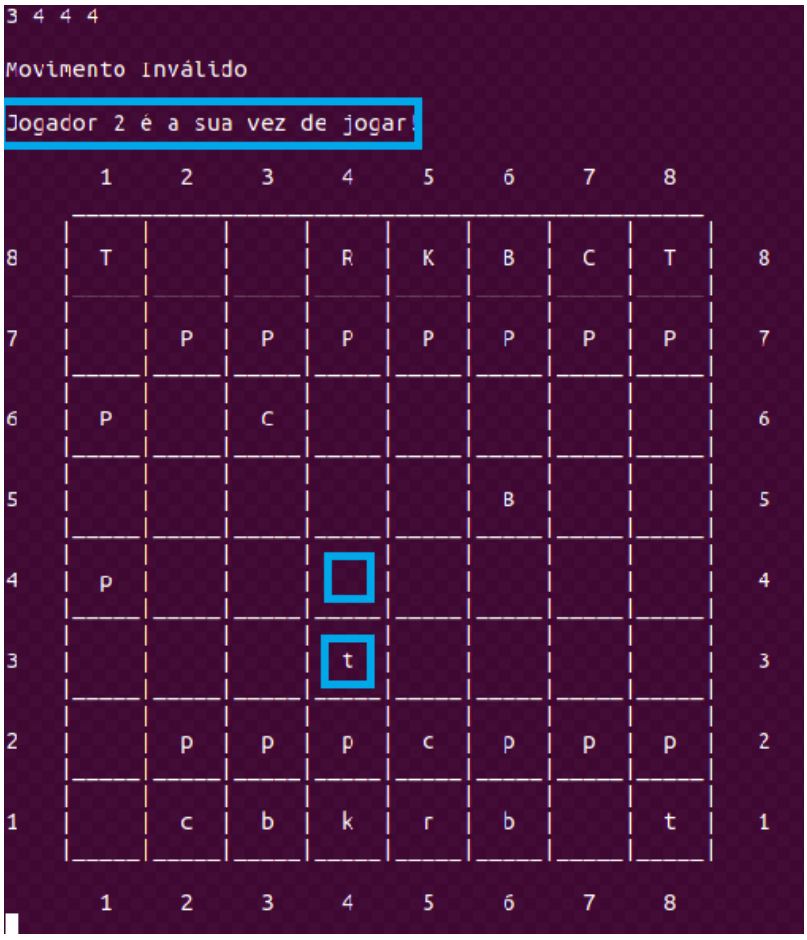


Figura 33 – Jogador 1 jogando fora da vez


3 4 4 4
Movimento inválido
Jogador 2 é a sua vez de jogar
1 2 3 4 5 6 7 8
8 T R K B C T 8
7 P P P P P P P 7
6 P C 6
5 B 5
4 p 4
3 t 3
2 p p p c p p p 2
1 c b k r b t 1
1 2 3 4 5 6 7 8