Ano Lectivo de **2009/2010**
Licenciatura em Engenharia Informática
Sob orientação do **Prof. Manuel Costa Leite**

Projecto Final de Curso

Xadrezística

Alunos: Clara Marquês – 20076347
Gonçalo Lourenço – 20070588

Índice

Resumo

O trabalho consiste num Jogo, mais precisamente, na extensão conceptual e algorítmica de um Jogo clássico, o Xadrez. Pretendia-se, como ideia inicial, que cada peça clássica fosse assumida e representada por um *robot*, atribuindo-se-lhes uma progressiva autonomia de decisão sobre as regras convencionais do Xadrez, estendendo assim o conceito e as regras do Jogo-Base.

O trabalho junta várias matérias aprendidas durante o curso:

- Uso de algoritmos;
- Existência de inteligência e vida artificial;
- Interfaces gráficas.

Conseguiu-se um Jogo híbrido, como se pretendia, com um enriquecimento na autonomia dos agentes, suficiente para o tornar interessante mas não excessivo para que não quebrasse o sentido completo de jogo.

Foi usado o *Java* como linguagem de programação e para a interface gráfica usámos a *Applet SWING* do *Java*.

1. Introdução

1.1. Objectivos

Pretendia-se, como ideia inicial, que cada peça clássica fosse assumida, representada por um *robot*, atribuindo-se-lhes uma progressiva autonomia de decisão sobre as regras convencionais do Xadrez. Procedia-se, assim, a uma extensão do conceito e das regras do Jogo-Base, o Xadrez

Pretendia-se também, a um nível motivacional, fazer algo criativo, com o qual pudéssemos aprender coisas novas e, ao mesmo tempo, consolidar conhecimentos adquiridos durante o curso.

1.2. Motivação

Inicialmente – como já dissemos -, a ideia para este projecto era implementar o jogo de xadrez de modo a ser jogado fisicamente com *robots*. No entanto, depois de termos feito uma avaliação do estado da arte, confrontámo-nos com a constatação de que esta é uma ideia já explorada, e que em si mesma não suscitaria as potencialidades de criatividade que pretendíamos originalmente. Assim estudámos o xadrez e o que este nos poderia oferecer dentro da ideia inicial, ainda que sem a robótica (no sentido físico e "equipamental" do termo), atendo-nos "apenas" a uma dimensão algorítmica.

Analisando como decorre um jogo de xadrez jogado com *robots*, chegámos à conclusão que a parte mais interessante seria quando, por exemplo, no começo do jogo, se um cavalo se quisesse deslocar para a frente de um peão, o peão *robot* teria que se mover para dar passagem ao cavalo *robot*, sendo este um exemplo simples daquilo a que vamos chamar o *problema-tipo*.

Começámos então a estudar este *problema-tipo* e como o resolver, fazer com que um *robot* (peça) consiga chegar de um ponto a outro da maneira mais eficiente, mesmo que envolvendo uma solução *negociada*.

Porém, não queríamos que o nosso projecto se ficasse por ai e fomos construindo, progressivamente reelaborando um Jogo – vamos chamar-lhe *Xadrezística* -, a partir do que tínhamos – o Xadrez.

Esse Jogo, que evoluiu deliberadamente de forma híbrida e se apropriou na junção de algumas características gerais de vários jogos; se bem que o jogo se tenha tornado noutro que não o Xadrez (por isso avançámos o nome de *Xadrezística*) a ideia de o fazer com *robots* manteve-se.

Precisávamos, no entanto, de um mínimo de 12 *robots* diferentes para conseguirmos simular o jogo e um número mínimo de 32 para o fazer na sua totalidade. Assim, devido à falta de meios físicos, objectivos, exteriores, "equipamentais", optámos por fazer o jogo em JAVA simulando o comportamento dos robots fisicamente.

1.3. O *Novo Jogo*, a *Xadrezística*

1.3.1. Objectivo do *Novo Jogo*

Este nosso novo Jogo – vamos continuar a designá-lo, à falta de melhor, pelo neologismo *Xadrezística* - tem como principal objectivo fazer o maior número possível de pontos, tomando as peças do adversário. Este objectivo é, em si mesmo, comum a muitos dos Jogos concebíveis.

O jogo pode acabar quando as peças do adversário tiverem sido comidas ou por acordo de ambos os jogadores, ganhando em ambos os casos quem tem maior pontuação. Também este objectivo estratégico é, em si mesmo, comum a muitíssimos tipos de jogos.

1.3.2. Dificuldade do *Novo Jogo*

O jogador pode escolher a dificuldade com que pretende jogar, escolha essa que vai ditar o número de pontos inicial do seu jogo:

- Fácil: 160 pontos;
- Médio: 90 pontos;
- Difícil: 50 pontos.

1.3.3. Peças do *Novo Jogo*

A posição inicial das peças é atribuída de forma aleatória.

As peças e o tabuleiro usados neste jogo são os mesmos que no Jogo-Base, o Xadrez, assim como as movimentações das peças também o são:

- **Torre**: move-se um qualquer número de casas segundo um movimento tanto na horizontal como na vertical;
- **Bispo**: pode-se mover para qualquer casa na diagonal;
- **Cavalo**: move-se duas casas na horizontal ou vertical, seguido de uma casa na perpendicular ao movimento anterior, à semelhança de um L;
- **Rainha**: pode-se mover um qualquer número de casas em qualquer direcção;
- **Rei**: pode mover-se uma casa em qualquer direcção;
- **Peão**: move-se uma ou duas (somente na 1ª vez que se move) casas na vertical, sendo que neste jogo o peão não tem movimentação especial para tomar.

Cada peça possui uma determinada pontuação:

- **Torre**: 3 pontos;
- **Bispo**: 3 pontos;

- **Cavalo**: 3 pontos;
- **Rainha**: 7 pontos;
- **Rei**: 10 pontos;
- **Peão**: 1 ponto.

Estas pontuações determinam o custo de movimentação de cada peça, sendo que este será multiplicado pelo número de casas a percorrer. Por outro lado, também serve para determinar quando se pode tomar uma peça.

1.3.4. Tomar Peças, segundo a nova Semântica

Para tomar peças do adversário será necessário fazer três coisas:

- Somar os pontos da peça que se pretende tomar com as peças da mesma cor que lhe são contíguas;
- Somar os pontos das peças do jogador que quer tomar que se encontram à volta da peça que se pretende tomar;
- Verificar se a segunda soma é superior à primeira.

Se esta verificação for verdadeira então o jogador pode tomar a peça pretendida.

1.3.5. Personalidades, na nova Semântica

Cada peça tem uma certa personalidade que irá influenciar algumas jogadas, as personalidades presentes no jogo são:

- **Desmoralizado**: faz com que a peça perca valor (pontos) ou fique parada durante um turno, poderá agravar-se se a peça continuar desmoralizada;
- **Herói**: faz com que a peça possa jogar duas vezes seguidas;
- **Cobarde**: faz com que a peça não se queira mover para a posição que o jogador mandou e se o jogador insistir a peça poderá ficar desmoralizada.
- **Inspirador**: faz com que a peça inspire as que estão à sua volta aumentando a moral e removendo os efeitos da desmoralização;
- **Companheiro**: peça poderá ganhar +1 ponto nesse turno, na presença do seu companheiro ou desmoralizado na falta deste;
- **Resistente**: se uma peça do adversário se encontrar no seu caminho esta peça pode deixá-la atordoada e consecutivamente não se move durante um turno.

A forma como estas estão distribuídas pelas peças é a seguinte:

- Torre é cobarde;
- Peão é herói;

- Rei e Rainha são companheiros;
- Cavalo é resistente;
- Bispo é inspirador.

1.4. Pré-requisitos do Projecto

Segue-se uma enumeração dos requisitos do projecto:

- Deve permitir que duas pessoas joguem alternadamente, por turnos, em que cada uma joga com as peças da cor que lhe pertence, branco ou preto;
- As peças devem mover-se como um robot se moveria, casa a casa, ao invés de aparecer logo na posição final;
- O jogo deve ser implementado de maneira a seguir as características e regras explicadas no ponto anterior.

Requisitos da interface:

- Pretende-se uma interface gráfica que contenha um tabuleiro como o de xadrez e peças de xadrez;
- Mostrar de forma permanente os pontos de cada jogador no momento;
- As peças devem aparecer inicialmente em posições aleatórias;
- Deve permitir que cada jogador clique na peça que quer mover;
- Deve mostrar que peça foi escolhida, "clicada", pelo utilizador e as casas para a qual esta se pode mover;
- Deve mostrar que peças estão afectadas pela sua personalidade;
- Deve ter uma caixa onde se mantém um histórico das jogadas que cada jogador fez e onde se mostre erros ao utilizador.

2. Interface

Neste ponto irá ser explicado e mostrado como funciona a interface do jogo. Comecemos por mostrar as imagens representativas das peças:

Rei Bispo Cavalo

Torre Peão Rainha

Passemos agora à interface do jogo, podemos ver a janela inicial de jogo na Figura 1, os rectângulos vermelho e verde foram feitos posteriormente, para realçar certos aspectos da interface. Analisando esta imagem, podemos verificar que se trata de um tabuleiro de xadrez tradicional com coordenadas.

Focando o tabuleiro, podemos observar que ambos o rei e a rainha pretos estão contornados a amarelo, o que significa que a personalidade destas peças está activa, ou seja quando a personalidade de qualquer peça fica activa isso é demonstrado pelo contorno a amarelo da peça.

Realçado pelo rectângulo verde podemos ver os pontos de cada jogador, que vão actualizando à medida que o jogo vai decorrendo.

Dentro do rectângulo vermelho podemos ver uma indicação de qual o jogador que deverá jogar, esta caixa de texto irá manter um log de tudo o que se irá passar durante o jogo.

Figura 1 – Inicio do jogo

Figura 2 – Escolha de peça

Na Figura 2 podemos observar o que acontece, quando um jogador escolhe uma peça com cor que difere da cor que deverá jogar no turno. Dentro do rectângulo vermelho podemos ver a mensagem que dá a conhecer ao jogador o seu erro e diz qual é a cor que está a jogar nesse turno.

Podemos também observar o que acontece quando um jogador escolhe uma peça válida. A casa onde está a peça escolhida aparece com um contorno a verde e, todas as casas com um contorno azul são as casas para onde essa peça se pode movimentar.

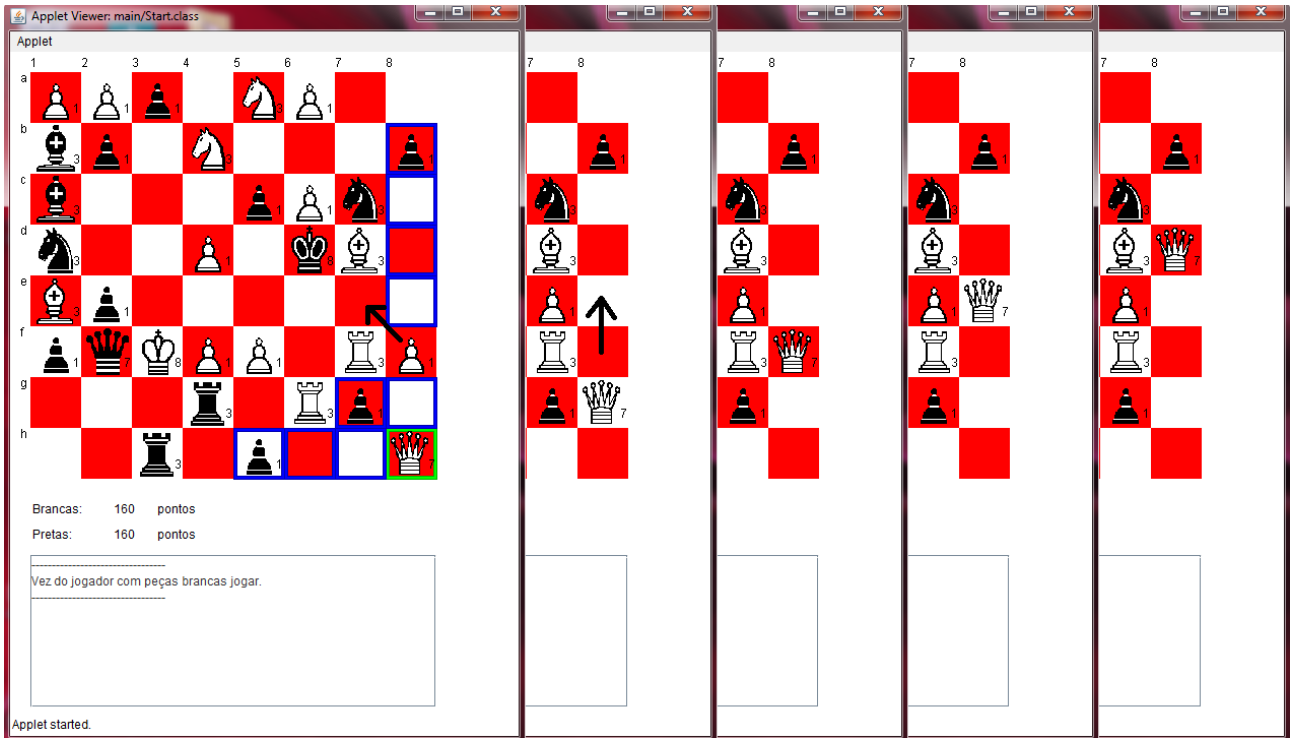


Figura 3 – Movimento da Peça

Depois de se escolher a peça que se pretende movimentar e a casa para onde se quer movimentá-la, o movimento dessa peça está demonstrado na Figura 3 bem como o movimento de peças que se têm que desviar para dar passagem a uma outra (as setas pretas são meramente ilustrativas do movimento que as peças irão fazer).

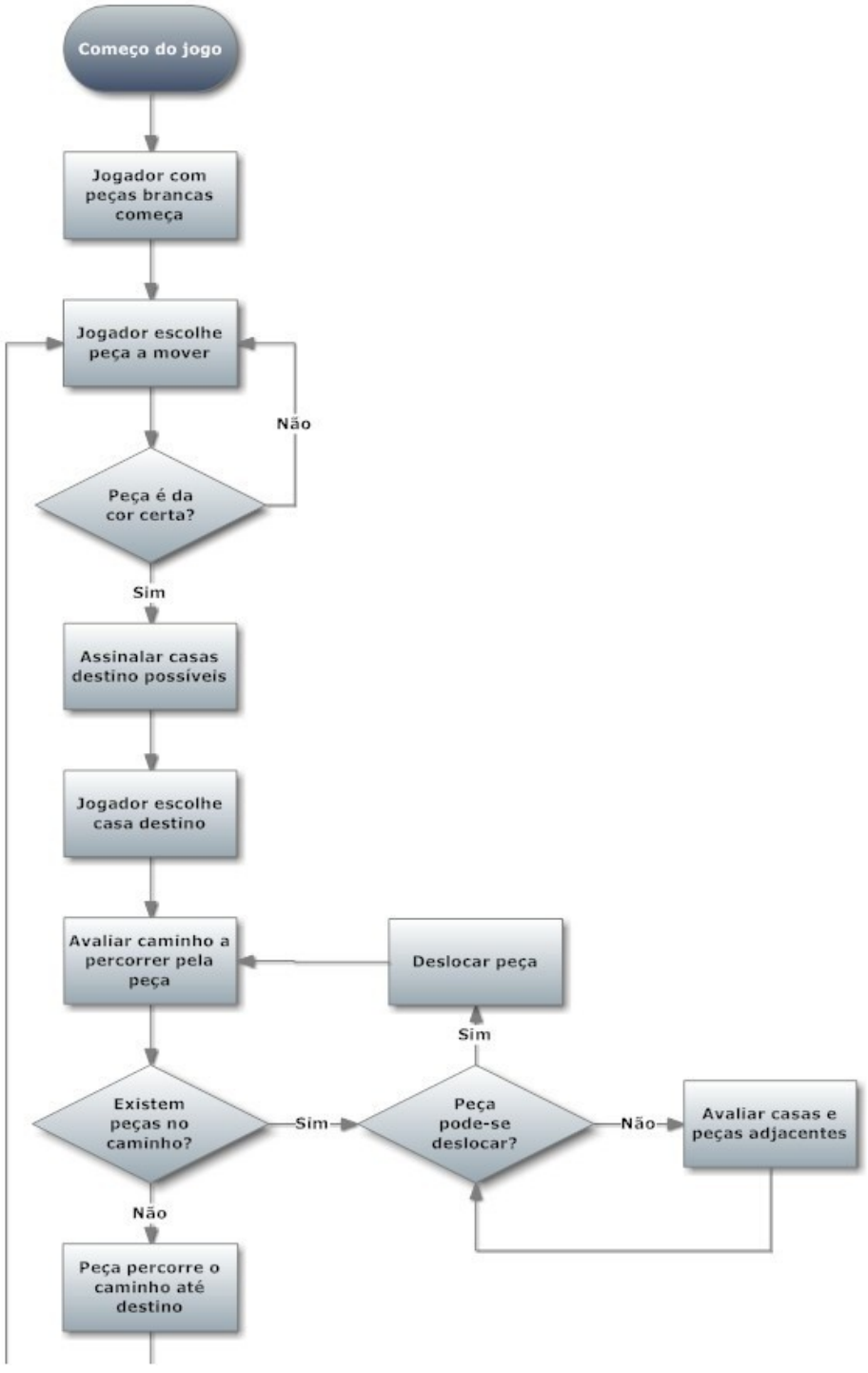
Na Figura 4 podemos ver a última fase do movimento da peça ilustrado na Figura 3, ou seja, quando a peça chega ao destino e as peças que foram desviadas para dar passagem voltam à sua casa inicial.

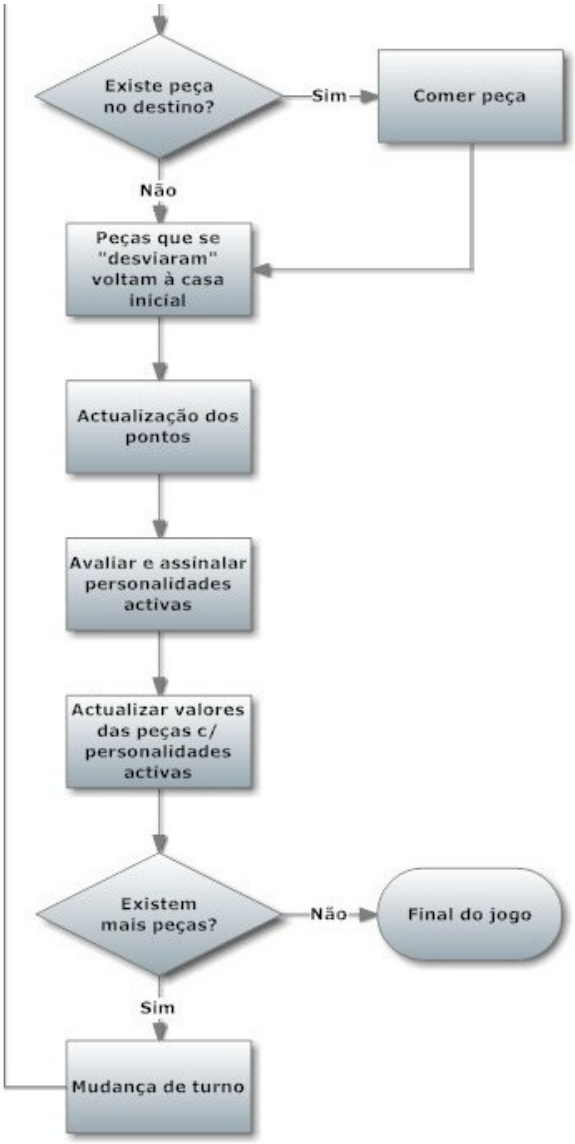
Nesta figura estão realçados 3 acontecimentos. Começando pelo rectângulo amarelo podemos ver que os pontos foram actualizados conforme o movimento e o gasto da peça. De seguida, no rectângulo vermelho podemos ver a informação do movimento feito pelo jogador, tanto da peça que o jogador mexeu como das peças que tiveram que se desviar.

Figura 4 – Final de uma jogada

3. Desenvolvimento

3.1. Fluxo da Aplicação





3.2. Tecnologias utilizadas

A linguagem utilizada na construção deste programa foi *Java* e foram usados as bibliotecas *AWT* e *Swing* para a *GUI* do mesmo.

3.3. Estrutura do Código

O código começa por estar divido em dois pacotes: *Main* e *Pieces*.

Dentro do pacote *Main* encontra-se a classe *Start* que trata de iniciar o jogo e é onde se encontram todas as funções que tratam o comportamento do jogo.

Dentro do pacote *Pieces* encontram-se:

- A superclasse *Piece* que contém tudo o que seja comum a todas as Peças;

- As classes *Pawn* (peão), *Rook* (torre), *Knight* (cavalo), *Bishop* (bispo), *King* (rei), *Queen* (rainha), contém tudo o que pertence a cada tipo de peça individualmente, validação dos movimentos, valores, personalidade, imagem que as representa;
- E, por último contém a classe *Board* que desenha e actualiza o tabuleiro a cada turno.

3.4. Algoritmos

Um dos algoritmos mais complicados de programar foi a função que trata das movimentações das peças, quando uma peça precisa de percorrer o seu caminho.

Para resolver o problema de como é que as peças se iriam mover, e para onde, decidimos utilizar um algoritmo de busca em profundidade limitado com limite 2. O que significa só irá concretizar dois níveis de busca.

Passando a explicar a função em causa.

Esta função recebe como argumentos a peça que se pretende mexer (peça 1), um vector com o caminho que este tem que percorrer e a peça que se encontrou no caminho (peça 2).

A primeira coisa que esta faz é recorrer a outra função, que nos dá um vector com todas as casas que rodeiam a peça 2, exceptuando as casas que coincidem com o caminho que a peça 1 tem que percorrer, já que queremos manter esse caminho livre.

Posto isto, analisamos cada posição desse vector à procura de casas vazias, quando encontrar a primeira casa vazia a peça 2 desloca-se para essa posição, libertando mais casas no caminho da peça 1. Se não existirem casas vazias neste vector escolhemos a peça que se encontra na primeira posição deste vector e esta passa a ser a nossa peça 2, repetindo novamente o processo.

```
    /**
     * Função para resolver o problema do que vai acontecer quando se encontra uma peça no caminho.
     * @param pc1 peça que se encontra no caminho.
     * @param pc peça a mover.
     * @param path caminho que a peça a mover quer percorrer.
     */
    public void resolvePiece(Piece pc1, Piece pc, Vector<int[]> path){

        //vai buscar as casa envolventes à peça que encontrou no caminho e coloca num vector
        Vector<int[]> surTiles = pc1.surroundingTiles();

        debug(surTiles, "ENVOLVENTES");

        Vector<int[]> options = new Vector<int[]>();

        //para todas as casas envolventes
        for(int i = 0; i < surTiles.size(); i++){
            //para todas as casas do caminho
            boolean flag = false;
            for(int j = 0; j < path.size(); j++){
                //verifica quais as casas do primeiro vector que estão no caminho e as q não coincidirem coloca num vector de opções para movimentação
                if(   surTiles.elementAt(i)[0]   ==   path.elementAt(j)[0]   &&
```

```
                    surTiles.elementAt(i)[1] == path.elementAt(j)[1] )
                      //options.addElement(surTiles.elementAt(i));
                      flag = true;
                }
                if(!flag){
                    options.addElement(surTiles.elementAt(i));
                }
            }

            debug(options,"OPTIONS");

            //para todas as opções
            for(int i = 0; i < options.size(); i++){
                //se a opção estiver livre a peça passa para lá
                if(isSpaceEmpty(options.elementAt(i)[0], options.elementAt(i)[1])){
                    rearrangePiece(pc1,options.elementAt(i));
                    return;
                }
            }

            //
            for(int i = 0; i < options.size(); i++){

                Piece         aux         =         findPiece(options.elementAt(i)[0],
options.elementAt(i)[1]);
                Vector<int[]> surTile2 = aux.surroundingTiles();
                for(int[] j: surTile2){
                    if(isSpaceEmpty(j[0],j[1])){
                        rearrangePiece(aux,j);

                        rearrangePiece(pc1,options.elementAt(i));
                        return;

                    }
                }
            }
        }
```

4. Desenvolvimentos Futuros

Em termos de desenvolvimentos futuros, este jogo tem 2 tipos de vertentes que poderá seguir, sendo que o mais interessante seria seguir as duas. Uma delas seria fazer uma simulação com robots para depois passar o jogo para o plano físico. A outra seria seguir a vertente de jogo no computador e, visto que a internet faz parte do dia-a-dia de muita gente, fazer com que este possa ser jogado em rede.

4.1. Código

Desenvolvimentos futuros em termos de código seria apenas terminar e polir o jogo como um jogo de computador. Implementar uma forma de gravar de forma permanente as pontuações mais altas, implementar 3 tipos diferentes de dificuldade para começar o jogo e por fim e mais importante implementar um modo de jogador vs computador.

4.2. Interface

De futuro, a interface poderá ser completada em vários aspectos:

- Inclusão de uma barra menu que teria opções como iniciar novo jogo, ver pontuações mais altas, ajuda (com as regras do jogo), escolher dificuldade em que se pretende jogar (entre jogos);
- Criar uma janela que apareça quando o utilizador corre o programa para que este possa escolher a dificuldade em que pretende jogar, cada vez que termina poderá mudar esta opção na barra de menu;

4.3. Jogo em Rede

Implementar uma classe que permite ao jogador inserir um endereço IP e que depois se tenta ligar a esse mesmo IP, permitindo que duas pessoas em qualquer sítio consigam jogar entre elas.

4.4. Simulação com Robots

Neste ponto, o objectivo seria implementar o jogo numa plataforma 3D, para fazer uma simulação de como o jogo seria em tabuleiro com robots. Mais tarde, eventualmente, criar-se-ia um protótipo do jogo no plano físico.

5. Conclusões

Para concluir, foi verificado que todos os objectivos a que nos propúnhamos foram atingidos.

Queremos com isto dizer que, todos os requisitos que delineámos para a construção do jogo foram conseguidos. Também o objectivo de criar algo que, ao nosso olhar, fosse diferente e criativo julgamos ter sido atingido. Pelo facto de, termos criado um jogo inovador a partir de um já existente e bem conhecido por parte de todos nós.

Com este projecto tivemos oportunidade de consolidar conhecimentos em várias áreas do nosso curso, tais como:

- Inteligência artificial, que foi usada na movimentação e interacção das peças;
- Interface Homem-Máquina, pois tratando-se de um jogo no pc que possui interface, o uso desta matéria é implícito.
- POO (Programação Orientada a Objectos), pois foi este paradigma de programação que usámos na construção do jogo através da linguagem de programação JAVA;
- Construção da interface do jogo usando bibliotecas específicas do JAVA (SWING, AWT).

No que toca à divisão do tempo pelas várias fases do projecto pode-se dizer que, grande parte do mesmo foi dispendido no exercício mental de construção/concepção do jogo.

Este facto fez com que faltasse algum tempo para "limar algumas arestas" no código e na eliminação de alguns bugs que o producto final possa conter.

Este trabalho é assim o resultado de um estudo minucioso que exigiu, no decorrer do mesmo, muita análise e reflexão.

6. Bibliografia

- Russel, Stuart J. and Norvig, Peter, Artificial Intelligence: A Modern Approach, 3th ed. Prentice Hall Publication, 2009. ISBN 0-13-604259-7.

7. Anexos

Índice de Anexos

Pacotes

Packages	
Main	
Pieces	

Main

Class Start

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Panel
              └ java.applet.Applet
                  └ main.Start
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.event.ItemListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible

---

```
public class Start
extends java.applet.Applet
implements java.awt.event.ItemListener, java.awt.event.ActionListener
```

See Also:

Serialized Form

Field Summary

char	**turn**

Constructor Summary

Start()

Method Summary

void	**actionPerformed**(java.awt.event.ActionEvent e)
void	**appendLogPieces**(Piece pc, int fxcoord, int fycoord, int msg) Função para escrever as acções executadas pelo utilizador e erros.
void	**appendLogTurn**(char turn) Função para informar quem joga a seguir.
void	**backToPlace**() Função para voltar a meter as peças que se mexeram para dar passagem a outra no mesmo lugar.
boolean	canEat(Piece eating, Piece eatable) Função para saber se uma peça pode comer outra ou não.
Piece	checkPath(Piece pc, java.util.Vector<int[]> path) Função para verificar se existe alguma peça no caminho de outra.

void	**debug**(java.util.Vector<int[]> vec, java.lang.String name)
void	**depthLimitedSearch**(Piece pc, java.util.Vector<int[]> path, int limit)
void	**eatPiece**(Piece found)
Piece	**findPiece**(int xCoord, int yCoord) Função para encontrar a peça que está numa determinada posição
Piece	**firstClick**(int x, int y)
java.awt.Image	**getPiece**(char color, char piece) Função para ir buscar a imagem de uma peça.
java.lang.String	**getType**(Piece pc) Função para descobrir o tipo de peça: Bispo, Cavalo, Peão, Rainha, Rei, Torre.
void	**init**()
boolean	**isSpaceEmpty**(int xCoord, int yCoord) Função para verificar se uma casa está vaga ou não.
void	**itemStateChanged**(java.awt.event.ItemEvent e)
void	**main**(Piece pc, int xcoord, int ycoord) Função que vai tratar da movimentação das peças.
void	**movePiece**(Piece pc, java.util.Vector<int[]> path)
int[]	**randomCoords**() Função para gerar coordenadas random para cada peça.
void	**rearrangePiece**(Piece pc, int[] newPos) Função para mover uma peça que se pretende tirar do caminho de outra.
void	**resolvePiece**(Piece pc1, Piece pc, java.util.Vector<int[]> path) Função para resolver o problema do que vai acontecer quando se encontra uma peça no caminho.
boolean	**secondClick**(Piece pc, int x, int y)
java.lang.String	**toCoords**(char type, int var) Função para converter um inteiro para as coordenadas que se encontram no tabuleiro.
java.util.Vector<int[]>	**validMoves**(Piece peca) Função que constrói um vector com as jogadas possíveis de uma peça.

Field Detail

turn
```
public char turn
```

Constructor Detail

Start
```
public Start()
```

Method Detail

init
```
public void init()
```
Overrides:
`init` in class `java.applet.Applet`

---

getPiece
```
public java.awt.Image getPiece(char color,
                               char piece)
```
Função para ir buscar a imagem de uma peça.
Parameters:
`color` - Cor da peça.
`piece` - Tipo de peça.
Returns:
Imagem

---

itemStateChanged
```
public void itemStateChanged(java.awt.event.ItemEvent e)
```
Specified by:
`itemStateChanged` in interface `java.awt.event.ItemListener`

---

actionPerformed
```
public void actionPerformed(java.awt.event.ActionEvent e)
```
Specified by:
`actionPerformed` in interface `java.awt.event.ActionListener`

---

findPiece
```
public Piece findPiece(int xCoord,
                       int yCoord)
```
Função para encontrar a peça que está numa determinada posição
Parameters:
`xCoord` - coordenada x.
`yCoord` - coordenada y.
Returns:
A peça encontrada ou null se não encontrou nenhuma.

---

isSpaceEmpty
```
public boolean isSpaceEmpty(int xCoord,
                            int yCoord)
```
Função para verificar se uma casa está vaga ou não.

Parameters:
`xCoord` - coordenada x.
`yCoord` - coordenada y.
Returns:
True se uma casa está livre ou false se uma casa está ocupada.

---

checkPath

```
public Piece checkPath(Piece pc,
                       java.util.Vector<int[]> path)
```

Função para verificar se existe alguma peça no caminho de outra.
Parameters:
`pc` - peça que se pretende mexer.
`path` - caminho que ela percorre até à casa pretendida.
Returns:
Peça que está no caminho.

---

main

```
public void main(Piece pc,
                 int xcoord,
                 int ycoord)
```

Função que vai tratar da movimentação das peças.
Parameters:
`pc` - peça a mover.
`xcoord` - posição x para a qual se pretende mover a peça.
`ycoord` - posição y para a qual se pretende mover a peça.

---

validMoves

```
public java.util.Vector<int[]> validMoves(Piece peca)
```

Função que constrói um vector com as jogadas possíveis de uma peça.
Parameters:
`peca` - peça da qual se pretende saber as jogadas possíveis.
Returns:
Posição para onde a peça se pode mover.

---

canEat

```
public boolean canEat(Piece eating,
                      Piece eatable)
```

Função para saber se uma peça pode comer outra ou não.
Parameters:
`eating` - peça que quer comer.
`eatable` - peça que se quer comer.
Returns:
True se puder comer, false se o contrário.

---

randomCoords

```
public int[] randomCoords()
```

Função para gerar coordenadas random para cada peça. Tendo em conta que não podem ficar peças sobrepostas.
Returns:
Vector de inteiros com 2 coordenadas.

getType

```
public java.lang.String getType(Piece pc)
```

Função para descobrir o tipo de peça: Bispo, Cavalo, Peão, Rainha, Rei, Torre .

Parameters:

`pc` - peça.

Returns:

O tipo: Bispo, Cavalo, Peão, Rainha, Rei ou Torre.

appendLogPieces

```
public void appendLogPieces(Piece pc,
                            int fxcoord,
                            int fycoord,
                            int msg)
```

Função para escrever as acções executadas pelo utilizador e erros.

Parameters:

`pc` - peça que vai fazer parte da mensagem.

`fxcoord` - coordenada x.

`fycoord` - coordenada y.

`msg` - Número da mensagem que se pretende apresentar.

appendLogTurn

```
public void appendLogTurn(char turn)
```

Função para informar quem joga a seguir.

Parameters:

`turn` - char representativo da cor que vai jogar.

toCoords

```
public java.lang.String toCoords(char type,
                                 int var)
```

Função para converter um inteiro para as coordenadas que se encontram no tabuleiro.

Parameters:

`type` - tipo de coordenada, x ou y.

`var` - inteiro a converter.

Returns:

Resultado da conversão como uma String.

resolvePiece

```
public void resolvePiece(Piece pc1,
                         Piece pc,
                         java.util.Vector<int[]> path)
```

Função para resolver o problema do que vai acontecer quando se encontra uma peça no caminho.

Parameters:

`pc1` - peça que se encontra no caminho.

`pc` - peça a mover.

`path` - caminho que a peça a mover quer percorrer.

rearrangePiece

```
public void rearrangePiece(Piece pc,
```

```
                        int[] newPos)
```

Função para mover uma peça que se pretende tirar do caminho de outra.

Parameters:

`pc` - peça que se pretende mudar de posição para dar passagem a outra.

`newPos` - nova posição.

---

backToPlace

```
public void backToPlace()
```

Função para voltar a meter as peças que se mexeram para dar passagem a outra no mesmo lugar.

---

depthLimitedSearch

```
public void depthLimitedSearch(Piece pc,
                               java.util.Vector<int[]> path,
                               int limit)
```

---

debug

```
public void debug(java.util.Vector<int[]> vec,
                  java.lang.String name)
```

---

firstClick

```
public Piece firstClick(int x,
                        int y)
```

---

secondClick

```
public boolean secondClick(Piece pc,
                           int x,
                           int y)
```

---

eatPiece

```
public void eatPiece(Piece found)
```

---

movePiece

```
public void movePiece(Piece pc,
                      java.util.Vector<int[]> path)
```

Pieces

Class Board

java.lang.Object
  └ java.awt.Component
    └ java.awt.Canvas
      └ **pieces.Board**

All Implemented Interfaces:

java.awt.event.MouseListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible

---

```
public class Board
extends java.awt.Canvas
implements java.awt.event.MouseListener
```

See Also:

Serialized Form

Nested Class Summary

Field Summary

java.util.Vector<int[]>	**buffedPieces**
boolean	**firstClick**
boolean	**greenFlag**
Piece	**pc**
Piece	**pcChosen**
java.util.Vector<int[]>	**validMovesVec**

Constructor Summary

Board(java.util.Vector<Piece> piece, Start app)

Method Summary

void	**mouseClicked**(java.awt.event.MouseEvent e)
void	**mouseEntered**(java.awt.event.MouseEvent e)
void	**mouseExited**(java.awt.event.MouseEvent e)

void	**mousePressed**(java.awt.event.MouseEvent e)
void	**mouseReleased**(java.awt.event.MouseEvent e)
void	**paint**(java.awt.Graphics g)
int	**xToCoords**(int x)
int	**yToCoords**(int y)

Field Detail

firstClick
```
public boolean firstClick
```

pcChosen
```
public Piece pcChosen
```

greenFlag
```
public boolean greenFlag
```

pc
```
public Piece pc
```

validMovesVec
```
public java.util.Vector<int[]> validMovesVec
```

buffedPieces
```
public java.util.Vector<int[]> buffedPieces
```

Constructor Detail

Board
```
public Board(java.util.Vector<Piece> piece,
             Start app)
```

Method Detail

paint
```
public void paint(java.awt.Graphics g)
```
Overrides:

`paint` in class `java.awt.Canvas`

mouseClicked
```
public void mouseClicked(java.awt.event.MouseEvent e)
```
Specified by:

`mouseClicked` in interface `java.awt.event.MouseListener`

mousePressed

```
public void mousePressed(java.awt.event.MouseEvent e)
```

Specified by:

`mousePressed` in interface `java.awt.event.MouseListener`

---

mouseReleased

```
public void mouseReleased(java.awt.event.MouseEvent e)
```

Specified by:

`mouseReleased` in interface `java.awt.event.MouseListener`

---

mouseEntered

```
public void mouseEntered(java.awt.event.MouseEvent e)
```

Specified by:

`mouseEntered` in interface `java.awt.event.MouseListener`

---

mouseExited

```
public void mouseExited(java.awt.event.MouseEvent e)
```

Specified by:

`mouseExited` in interface `java.awt.event.MouseListener`

---

xToCoords

```
public int xToCoords(int x)
```

---

yToCoords

```
public int yToCoords(int y)
```

Class Piece

java.lang.Object
  └ **pieces.Piece**

Direct Known Subclasses:

Bishop, King, Knight, Pawn, Queen, Rook

```
public class Piece
extends java.lang.Object
```

Field Summary

int	**auxXCoord**
int	**auxYCoord**
char	**color**
java.awt.Image	**img**
boolean	**moved**
java.lang.String	**personality**
char	**type**
protected int	**vip** Variavél que guarda a prioridade da peça.
protected int	**vip_aux** Variavél que guarda a prioridade da peça.
int	**xCoord**
int	**yCoord**

Constructor Summary

Piece(char color, char type, java.awt.Image img, int xCoord, int yCoord) Constructor da classe Piece.

Method Summary

int	**getVip_aux**()
int	**getVip**()

void	**move**(int fxCoord, int fyCoord) Função para mover uma peça.
java.util.Vector<int[]>	**movesAvailable**() Função para ver quais as opções de movimentação da peça.
java.util.Vector<int[]>	**path**(int fxCoord, int fyCoord) Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
int[]	**posAsVec**()
void	**setVip_aux**(int vip_aux)
java.util.Vector<int[]>	**surroundingTiles**() Função para sabermos as coordenadas das casas à volta de uma dada peça.
boolean	**validateMove**(int fxCoord, int fyCoord) Função para validar o movimento da peça.

Field Detail

moved
```
public boolean moved
```

color
```
public char color
```

type
```
public char type
```

img
```
public java.awt.Image img
```

xCoord
```
public int xCoord
```

yCoord
```
public int yCoord
```

auxXCoord
```
public int auxXCoord
```

auxYCoord
```
public int auxYCoord
```

personality
```
public java.lang.String personality
```

vip
```
protected int vip
```

Variavél que guarda a prioridade da peça.

---

vip_aux

```
protected int vip_aux
```

Variavél que guarda a prioridade da peça.

Constructor Detail

Piece

```
public Piece(char color,
             char type,
             java.awt.Image img,
             int xCoord,
             int yCoord)
```

Constructor da classe Piece.

Parameters:

`color` - guarda a cor da peça.

`type` - guarda o tipo de peça: Cavalo, Bispo, Rei, Rainha, Peão, Torre.

`img` - guarda a imagem da peça.

`xCoord` - coordenada x da posição onde a peça se encontra.

`yCoord` - coordenada y da posição onde a peça se encontra.

Method Detail

move

```
public void move(int fxCoord,
                 int fyCoord)
```

Função para mover uma peça.

Parameters:

`fxCoord` - coordenada x final da peça.

`fyCoord` - coordenada y final da peça.

---

validateMove

```
public boolean validateMove(int fxCoord,
                            int fyCoord)
```

Função para validar o movimento da peça.

Parameters:

`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.

`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.

Returns:

True ou false.

---

path

```
public java.util.Vector<int[]> path(int fxCoord,
                                    int fyCoord)
```

Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.

Parameters:

`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.

`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.

Returns:

Um vector de vectores de 2 posições com as coordenadas de cada casa do caminho.

---

movesAvailable

```
public java.util.Vector<int[]> movesAvailable()
```

Função para ver quais as opções de movimentação da peça.

Returns:

Um vector com as coordenadas, x e y, possíveis.

---

surroundingTiles

```
public java.util.Vector<int[]> surroundingTiles()
```

Função para sabermos as coordenadas das casas à volta de uma dada peça.

Returns:

Retorna todas as casas à volta de uma peça e que façam parte do tabuleiro.

---

getVip

```
public int getVip()
```

---

setVip_aux

```
public void setVip_aux(int vip_aux)
```

---

getVip_aux

```
public int getVip_aux()
```

---

posAsVec

```
public int[] posAsVec()
```

Class Bishop

```
java.lang.Object
  └ pieces.Piece
      └ pieces.Bishop
```

```
public class Bishop
extends Piece
```

O Bispo move-se na diagonal um qualquer número de casas não podendo, no entanto, passar por cima de nenhuma peça.

Field Summary

Fields inherited from class pieces.Piece
auxXCoord, auxYCoord, color, img, moved, personality, type, vip, vip_aux, xCoord, yCoord

Constructor Summary

Bishop(char color, char type, java.awt.Image img, int xCoord, int yCoord)

Method Summary

java.util.Vector<int[]>	**movesAvailable**() Função para ver quais as opções de movimentação da peça.
java.util.Vector<int[]>	**path**(int xcoord, int ycoord) Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
boolean	**validateMove**(int fxCoord, int fyCoord) Função para validar o movimento da peça.

Methods inherited from class pieces.Piece
getVip_aux, getVip, move, posAsVec, setVip_aux, surroundingTiles

Constructor Detail

Bishop

```
public Bishop(char color,
              char type,
              java.awt.Image img,
              int xCoord,
              int yCoord)
```

Method Detail

validateMove

```
public boolean validateMove(int fxCoord,
```

```
                                int fyCoord)
```

Description copied from class: `Piece`

Função para validar o movimento da peça.

Overrides:

`validateMove` in class `Piece`

Parameters:

`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.

`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.

Returns:

True ou false.

---

path

```
public java.util.Vector<int[]> path(int xcoord,
                                    int ycoord)
```

Description copied from class: `Piece`

Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.

Overrides:

`path` in class `Piece`

Parameters:

`xcoord` - coordenada x, da posição final para onde se pretende mover a peça.

`ycoord` - coordenada y, da posição final para onde se pretende mover a peça.

Returns:

Um vector de vectores de 2 posições com as coordenadas de cada casa do caminho.

---

movesAvailable

```
public java.util.Vector<int[]> movesAvailable()
```

Description copied from class: `Piece`

Função para ver quais as opções de movimentação da peça.

Overrides:

`movesAvailable` in class `Piece`

Returns:

Um vector com as coordenadas, x e y, possíveis.

Class King

```
java.lang.Object
  └pieces.Piece
      └pieces.King
```

```
public class King
extends Piece
```

O Rei pode-se mover uma casa em qualquer direcção.

Field Summary

Fields inherited from class pieces.Piece
auxXCoord, auxYCoord, color, img, moved, personality, type, vip, vip_aux, xCoord, yCoord

Constructor Summary

King(char color, char type, java.awt.Image img, int xCoord, int yCoord)

Method Summary

java.util.Vector<int[]>	**path**(int fxCoord, int fyCoord) Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
boolean	**validateMove**(int fxCoord, int fyCoord) Função para validar o movimento da peça.

Methods inherited from class pieces.Piece
getVip_aux, getVip, move, movesAvailable, posAsVec, setVip_aux, surroundingTiles

Constructor Detail

King

```
public King(char color,
            char type,
            java.awt.Image img,
            int xCoord,
            int yCoord)
```

Method Detail

validateMove

```
public boolean validateMove(int fxCoord,
                            int fyCoord)
```

Description copied from class: `Piece`

Função para validar o movimento da peça.

Overrides:
`validateMove` in class `Piece`
Parameters:
`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.
`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.
Returns:
True ou false.

---

path

```
public java.util.Vector<int[]> path(int fxCoord,
                                    int fyCoord)
```

Description copied from class: `Piece`
Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
Overrides:
`path` in class `Piece`
Parameters:
`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.
`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.
Returns:
Um vector de vectores de 2 posições com as coordenadas de cada casa do caminho.

Class Knight

```
java.lang.Object
  └ pieces.Piece
      └ pieces.Knight
```

```
public class Knight
extends Piece
```

O Cavalo pode-se mover duas casas na horizontal ou vertical seguido de uma casa à esquerda ou direita, fazendo um L.

Field Summary

Fields inherited from class pieces.Piece
auxXCoord, auxYCoord, color, img, moved, personality, type, vip, vip_aux, xCoord, yCoord

Constructor Summary

Knight(char color, char type, java.awt.Image img, int xCoord, int yCoord)

Method Summary

java.util.Vector<int[]>	**path**(int xcoord, int ycoord) Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
boolean	**validateMove**(int fxCoord, int fyCoord) Função para validar o movimento da peça.

Methods inherited from class pieces.Piece
getVip_aux, getVip, move, movesAvailable, posAsVec, setVip_aux, surroundingTiles

Constructor Detail

Knight

```
public Knight(char color,
              char type,
              java.awt.Image img,
              int xCoord,
              int yCoord)
```

Method Detail

validateMove

```
public boolean validateMove(int fxCoord,
                            int fyCoord)
```

Description copied from class: `Piece`

Função para validar o movimento da peça.

Overrides:
`validateMove` in class `Piece`
Parameters:
`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.
`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.
Returns:
True ou false.

---

path

```
public java.util.Vector<int[]> path(int xcoord,
                                    int ycoord)
```

Description copied from class: `Piece`
Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
Overrides:
`path` in class `Piece`
Parameters:
`xcoord` - coordenada x, da posição final para onde se pretende mover a peça.
`ycoord` - coordenada y, da posição final para onde se pretende mover a peça.
Returns:
Um vector de vectores de 2 posições com as coordenadas de cada casa do caminho.

Class Pawn

```
java.lang.Object
  └pieces.Piece
      └pieces.Pawn
```

```
public class Pawn
extends Piece
```

O Peão pode andar duas casa para a frente se for a primeira jogada e depois pode mover-se uma em frente.

Field Summary

boolean	**firstMove**

Fields inherited from class pieces.Piece

auxXCoord, auxYCoord, color, img, moved, personality, type, vip, vip_aux, xCoord, yCoord

Constructor Summary

Pawn(char color, char type, java.awt.Image img, int xCoord, int yCoord)

Method Summary

boolean	**heroism**()
java.util.Vector<int[]>	**movesAvailable**() Função para ver quais as opções de movimentação da peça.
java.util.Vector<int[]>	**path**(int fxCoord, int fyCoord) Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.
boolean	**validateMove**(int fxCoord, int fyCoord) Função para validar o movimento da peça.

Methods inherited from class pieces.Piece

getVip_aux, getVip, move, posAsVec, setVip_aux, surroundingTiles

Field Detail

firstMove

```
public boolean firstMove
```

Constructor Detail

Pawn

```
public Pawn(char color,
            char type,
            java.awt.Image img,
            int xCoord,
            int yCoord)
```

Method Detail

heroism

```
public boolean heroism()
```

---

validateMove

```
public boolean validateMove(int fxCoord,
                            int fyCoord)
```

Description copied from class: `Piece`

Função para validar o movimento da peça.

Overrides:

`validateMove` in class `Piece`

Parameters:

`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.

`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.

Returns:

True ou false.

---

path

```
public java.util.Vector<int[]> path(int fxCoord,
                                    int fyCoord)
```

Description copied from class: `Piece`

Função que guarda o caminho que a peça terá que percorrer até à casa pretendida.

Overrides:

`path` in class `Piece`

Parameters:

`fxCoord` - coordenada x, da posição final para onde se pretende mover a peça.

`fyCoord` - coordenada y, da posição final para onde se pretende mover a peça.

Returns:

Um vector de vectores de 2 posições com as coordenadas de cada casa do caminho.

---

movesAvailable

```
public java.util.Vector<int[]> movesAvailable()
```

Description copied from class: `Piece`

Função para ver quais as opções de movimentação da peça.

Overrides:

`movesAvailable` in class `Piece`

Returns:

Um vector com as coordenadas, x e y, possíveis.
