



Universidade do Porto  
Faculdade de Engenharia  
**FEUP**

# Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Programação 2  
2008

JOGO DE TABULEIRO

**QUARTO**

Implementação em C++

Relatório Final

**Turma 12. Grupo 5.**  
Gonçalo Bernardo  
Ismael Miranda  
Nuno Oliveira

**Resumo.** Neste trabalho apresenta-se uma implementação em C++ do Quarto, um jogo de tabuleiro para duas pessoas. A partir do código base fornecido e seguindo a mesma estrutura foi desenvolvido o programa que permite a dois jogadores defrontarem-se segundo as regras do jogo e visualizarem o estado actual do tabuleiro através de uma representação em modo texto. Neste relatório são apresentados os aspectos internos da mecânica de jogo e da arquitectura do sistema. É também fornecido o código fonte completo.

# Índice

1. Introdução.....	4
2. Descrição do Problema.....	4
2.1 Peças.....	4
2.2 Dinâmica de jogo.....	4
2.3 Terminação de jogo.....	4
3. Arquitectura do Sistema.....	4
4. Módulo de Lógica do Jogo.....	5
4.1 Representação do Estado do Jogo.....	5
Classe: Peca.....	5
Classe: Tabuleiro.....	6
4.2 Visualização do Estado do Jogo.....	7
4.3 Validação de Jogadas.....	7
Classe: Jogador.....	8
Classe: MyException.....	8
Classe: Jogo.....	8
4.4 Execução de Jogadas.....	9
4.5 Avaliação do Tabuleiro e Final do Jogo.....	9
5. Conclusões e Perspectivas de Desenvolvimento.....	9
A. Bibliografia.....	10
B. Anexos - Código Fonte.....	10

## 1. Introdução

No âmbito da disciplina de Programação 2 (1º ano, 2º semestre) foi proposta a realização de um trabalho de grupo envolvendo a implementação de um jogo de tabuleiro em C++ para dois jogadores que não dependesse do factor sorte.

Esperava-se obter uma aplicação para execução em ambiente de linha de comandos fiável e amigável do utilizador que permitisse a dois jogadores defrontarem-se segundo as regras de jogo. Para tal foi preciso desenvolver o modo de representação interna da mecânica de jogo e o modo de visualização do tabuleiro que permitisse fazer a comunicação com os utilizadores.

Ao longo deste relatório apresenta-se primeiro o jogo e as suas regras, depois mostra-se a forma como o problema foi abordado e finalmente descrevem-se as classes implementadas.

## 2. Descrição do Problema

O jogo a implementar é o Quarto. Este jogo, inventado por Blaise Müller, desenvolve-se num tabuleiro 4x4, possui 16 peças e só permite a participação de dois jogadores.

### 2.1 Peças

Cada peça tem quatro atributos: grande ou pequena, branca ou preta, redonda ou quadrada e maciça ou oca. Não existem peças iguais.

### 2.2 Dinâmica de jogo

Cada jogador lança um dado e o que obtiver maior pontuação será o primeiro a colocar uma peça no tabuleiro.

À vez os jogadores escolhem uma das peças que ainda não está colocada no tabuleiro para o seu adversário colocar na posição que entender.



*Tabuleiro e Peças do Quarto. Situação de vitória: numa das linhas todas as peças são grandes*

### 2.3 Terminação de jogo

Um jogador é declarado vencedor se ao colocar a peça consegue formar uma linha, coluna, diagonal ou quadrado 2x2 em que as quatro peças são representativas de um mesmo atributo (todas grandes, todas brancas, etc). O jogo termina em empate se todas as posições do tabuleiro tiverem sido ocupadas e não se verificar nenhuma situação de vitória.

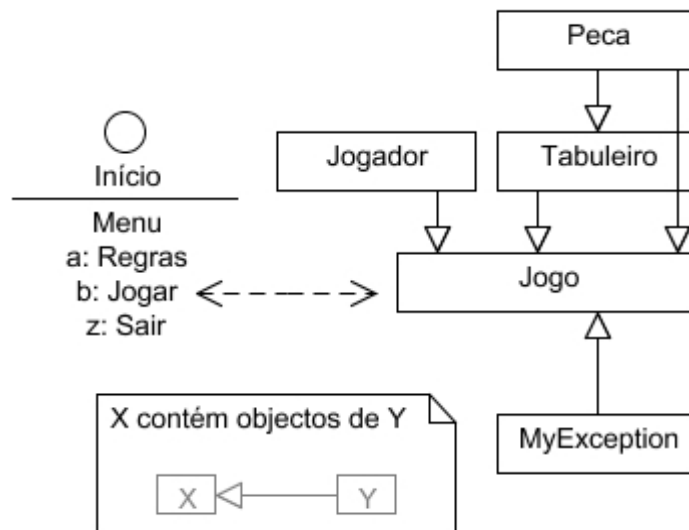
Uma das características invulgares do Quarto é o facto de existir um mesmo conjunto de peças para os dois jogadores e não dois conjuntos distintos para cada um deles. Este é considerado um jogo totalmente imparcial onde apenas é testada a inteligência e perícia bem como a capacidade de antecipação dos jogadores.

## 3. Arquitectura do Sistema

Na implementação da aplicação foi seguido o ideal de programação orientada a objectos. O que se fez foi criar uma representação da realidade numa versão computável. Assim distinguiram-se três agentes físicos a representar: os jogadores (classe *Jogador*), o tabuleiro (classe *Tabuleiro*) e as

peças (classe *Peca*). Um quarto elemento não físico diz respeito às regras de jogo e à dinâmica deste (classe *Jogo*). Para tornar a aplicação robusta tornou-se necessário a introdução de um último elemento que fosse capaz de tratar erros cometidos pelos utilizadores na introdução de dados (classe *MyException*).

No esquema seguinte mostram-se as relações de interdependência entre as várias classes.



Visão geral da arquitectura do sistema

Pode-se acrescentar que:

- O menu recebe os utilizadores e desencadeia o jogo;
- As peças fazem parte do jogo antes de serem jogadas e do tabuleiro depois;
- Os jogadores são os intervenientes na dinâmica de jogo e as suas opções reflectem-se no tabuleiro;
- Se ocorrerem problemas na comunicação com os utilizadores são lançadas excepções que fazem com que seja interrompida a lógica normal de jogo e se volte atrás na execução.

Partindo do princípio que toda a representação interna dos vários elementos físicos é coerente a codificação do jogo resume-se a traduzir em linguagem C++ a sua ordem natural. No Quarto, de uma forma abstracta, esta ordem será: um jogador escolhe a peça a ser jogada, o seu adversário coloca-a numa casa vazia do tabuleiro e o jogo continua invertendo os papéis dos jogadores a cada ciclo a menos que seja detectada uma situação vencedora ou o tabuleiro esteja já totalmente preenchido.

## 4. Módulo de Lógica do Jogo

### 4.1 Representação do Estado do Jogo

A representação interna do estado do jogo depende das classes *Peca* e *Tabuleiro*.

- Classe: *Peca*

Os objectos desta classe pretendem ser uma representação das peças reais de jogo. Como tal têm obviamente quatro membros-dado que definem a peça representada. Cada característica da peça é dada por um valor booleano. Assim, as variáveis *bool branca*, *bool grande*, *bool redonda* e *bool macica* tomam valores *true* ou *false* conforme a peça em questão. Por exemplo, se a peça for preta,

grande, redonda e oca a sua representação interna na forma de objecto da classe *Peca* será feita da seguinte forma: *branca = false; grande = true; redonda = true; e macica = false*.

A estas quatro variáveis foi acrescentado um quinto membro-dado *bool naoPeca*. Para peças reais esta variável terá de assumir obrigatoriamente o valor *false*. A variável *naoPeca* assumirá o valor *true* quando for necessário “preencher” a matriz tabuleiro com casas vazias. O que se faz é preenchê-lo com peças que não têm um paralelo na realidade (não-peças) e que servem apenas para reservar a posição sem permitir a proliferação de valores desconhecidos.

O construtor sem argumentos da classe cria não-peças em que a variável *naoPeca* toma o valor *true* e às demais variáveis é atribuído por defeito o valor *false*. O construtor com argumentos recebe por valor os atributos da peça a criar e por defeito atribui a *naoPeca* o valor *false*.

O método *string representacaoPeca() const* retorna a informação do objecto do tipo *Peca*. Se o objecto for uma não-peça são retornados três espaços. Caso o objecto seja a representação de uma peça real são concatenados três caracteres de acordo com o seguinte modelo:

- 1º caracter (Forma): '(' redonda, '[' quadrada;
- 2º caracter (Tamanho e Cor): 'b' branca, 'p' preta; <maiúscula> grande, <minúscula> pequena;
- 3º caracter (Preenchimento): '!' macica, ';' oca.

Para finalizar a definição da classe foram implementados os operadores `==` e `<<` para realizar as tarefas normalmente a eles atribuídas. O operador `<<` imprime o resultado da invocação de *string representacaoPeca() const*.

- Classe: *Tabuleiro*

Internamente o tabuleiro é visto como uma matriz 4x4 de objectos da classe *Peca*. De notar que os índices da matriz estão totalmente encapsulados devendo as classes exteriores usar a numeração natural começada em 1. Às classes exteriores é também exigido que passem apenas valores válidos aos métodos de *Tabuleiro* uma vez que estes não procedem a qualquer tipo de verificação explícita de coerência dos dados.

Na definição da classe incluíram-se dois membros-dado. Um *const Peca naoPeca* que guarda uma representação de não-peças (útil para comparações) e a própria matriz como vector de vectores de objectos da classe *Peca*, *vector< vector<Peca> > matriz*.

Os métodos definidos nesta classe destinam-se essencialmente a servir a classe *Jogo* actualizando e verificando o estado da matriz tabuleiro:

- Construtor *Tabuleiro(const Peca&)*: gera a matriz preenchida com copias da peça referenciada como parâmetro (em situação normais, uma não-peça);
- *void imprimeTabuleiro() const*: imprime tabuleiro como discutido na secção 4.2;
- *bool estaLivre(int, int) const*: testa se posição do tabuleiro está livre;
- *void inserePeca(int, int, const Peca&)*: insere peça na posição indicada;
- *bool vencedor() const*: detecta situações vencedoras;
- *bool estaCheio() const*: testa se tabuleiro está cheio percorrendo a matriz e chamando *bool estaLivre(int, int) const* para cada célula.

Todos os métodos têm uma implementação simples e intuitiva. Pensa-se que apenas o método *bool vencedor() const* merece uma explicação mais aprofundada. A detecção de situações vencedoras aproveita a representação booleana das características das peças. O que se faz é percorrer as linhas, colunas, diagonais e quadrados 2x2 com cinco variáveis auxiliares que vão sendo incrementadas ou mantidas conforme a peça existente na célula em questão. Tomemos como

exemplo a variável *int grd* (grande) que é colocada a 0 a cada leitura de um novo bloco. Em cada célula a variável é incrementada se a peça aí existente for grande (equivalente a *grande = true*) e mantida se a peça em questão for pequena (equivalente a *grande = false*). Considerem-se três situações:

- *grd*, depois de ser lido o bloco, apresenta um valor diferente de 0 ou 4. Ficamos a saber que as peças nem são todas grandes nem são todas pequenas;
- *grd*, depois de ser lido o bloco, apresenta um valor igual a 0 ou 4. Ficamos a saber que uma das características, grande ou pequena, se manteve e estamos portanto numa situação de vitória;
- Salvarde-se a situação em que a variável *int npc* (não-peça) apresente valor não nulo. Daí concluímos que o bloco não está totalmente preenchido e invalida-se a situação anterior.

Esta análise é feita também para as restantes variáveis: *int brc* (branca), *int red* (redonda) e *int mac* (maciça). O método usa a variável *bool win* para aglomerar toda esta informação e retorna *true* se encontrar uma situação vencedora em algum dos blocos – correspondente a alguma das variáveis *grd*, *brc*, *red*, *mac* apresentar um valor igual a 0 ou 4 e a variável *npc* se manter nula (*win = true*).

## 4.2 Visualização do Estado do Jogo

A materialização do tabuleiro está implementada em *void Tabuleiro::imprimeTabuleiro()* *const* que gera uma grelha em modo texto como a apresentada. A materialização das peças nele contidas é o resultado da simples utilização do operador << sobre objectos da classe *Peca*.

```

# _ A _ # _ B _ # _ C _ # _ D _ #
|     |     |     |     |
1     |     |     |     | 1
|     |     |     |     |
#-----#-----#-----#-----#
|     |     | (B; |     | 2
|     |     |     |     |
#-----#-----#-----#
| (P; |     |     |     | 3
|     |     |     |     |
#-----#-----#-----#
|     |     | [p; |     | 4
|     |     |     |     |
# _ A _ # _ B _ # _ C _ # _ D _ #

```

Como seria de esperar a nomenclatura utilizada no tabuleiro, para linhas de 1 a 4 e para colunas de A a D, é a que é pedida ao utilizador aquando da introdução dos dados relativos à posição em que quer jogar a peça escolhida.

## 4.3 Validação de Jogadas

A forma como o jogo se desenvolve internamente é descrita na classe *Jogo* que depende das duas já apresentadas e de outras duas, *Jogador* e *MyException*. A classe *Jogo* é ainda responsável pela interface que permite a comunicação entre jogadores e máquina, bem como pelo tratamento de

erros cometidos por estes na introdução dos dados. Far-se-á primeiro uma breve descrição destas classes e de seguida serão descritos os procedimentos para validação de jogadas.

- Classe: *Jogador*

Esta é uma classe bastante simples cujos objectos pretendem representar os jogadores. No Quarto as peças são independentes dos jogadores e por isso como membro-dado apenas se definiu *string nome*. Para operar com estes objectos codificaram-se métodos básicos. Um construtor sem parâmetros que cria jogadores com nome vazio, *gets* e *sets*. Definiu-se também o operador << .

- Classe: *MyException*

Para auxiliar a classe *Jogo* no tratamento de excepções foi criada a classe *MyException* cujos objectos são lançados aquando da ocorrência de alguma situação anormal. Estes objectos têm como membro-dado *const string mensagem* que é utilizado para comunicar ao utilizador o tipo de erro quando a excepção for interceptada.

- Classe: *Jogo*

Se as três primeiras classes apresentadas representavam os elementos físicos presentes no jogo a classe *Jogo* agrupa estes elementos e estabelece a dinâmica segundo a qual se relacionam.

Como membros-dado da classe definiram-se:

- *Tabuleiro Tab*;
- *Jogador jogadores[2]*;
- *int jogadorActivo* que guarda o índice (0 ou 1) do jogador que está a executar a jogada (relativamente ao array destes);
- *vector<Peca> pecas* que guarda as peças ainda não jogadas.

Os métodos aqui implementados gerem a dinâmica de jogo:

- Construtor *Jogo()*: cria um objecto para um novo jogo. Inicializa o tabuleiro com não-peças, pede os nomes dos jogadores e actualiza os elementos de *jogadores*, atribui um valor aleatório (0 ou 1) a *jogadorActivo* e coloca as 16 peças iniciais do jogo em *pecas*;
- *void jogar()*: desencadeia a acção do jogo propriamente dita;
- *void trocaJogador()*: dá a vez ao jogador seguinte alterando o membro-dado *jogadorActivo*;
- *void listarPecas()*: imprime os elementos de *pecas* para que os jogadores saibam que peças ainda podem jogar;
- *bool executaJogada()*: executa uma jogada pedindo a um dos jogadores a peça a ser jogada e ao seu adversário a posição do tabuleiro onde esta deve ser colocada.

Existem ainda 3 membros-função - *int interChar1(char)*, *int interChar2(char)*, *int interChar3(char)* – que se destinam a interpretar caracteres introduzidos pelo utilizador em várias situações: caracter lido para peça, caracter lido para linha e caracter lido para coluna, respectivamente. Se esse caracter for válido as funções retornam um valor significativo. Se não, lançam uma excepção com a mensagem de erro apropriada. Optou-se por receber todos os dados do utilizador em forma de *string* uma vez que assim é possível ao programador controlar mais eficazmente a ocorrência de erros sem que a execução do programa seja afectada.



É possível agora apresentar o mecanismo de validação de jogadas implementado em *bool executaJogada()*. Este método recebe os dados introduzidos pelos jogadores e desencadeia os processos que levam à actualização do estado do jogo. Por ser aqui que é pedida a intervenção dos utilizadores este é o ponto mais sensível da aplicação e no qual é fundamental antever situações anómalas. Para isso foi implementado um mecanismo de detecção de dados inválidos que impede que o programa os assimile, avisa o utilizador e volta a pedir os dados. Basicamente esta rotina intercepta as excepções lançadas pelos métodos discutidos no parágrafo anterior e não deixa o programa avançar enquanto não conseguir informação dentro do âmbito definido. Este mecanismo impede também que uma peça seja jogada numa casa já ocupada.

#### 4.4 Execução de Jogadas

Assumindo jogadores ideais que não cometem erros e que introduzem sempre a informação adequada podemos simplificar a descrição de *bool Jogo::executaJogada()* da seguinte forma:

- Lista peças por jogar invocando *void Jogo::listarPecas()*;
- Pede a um dos jogadores a peça a ser jogada aproveitando o facto de na listagem das peças ser atribuído a cada uma delas um carácter identificativo;
- Calcula índice da peça no vector *pecas*;
- Pede ao outro jogador a posição do tabuleiro em que quer jogar a peça;
- Insere a peça escolhida na posição indicada do tabuleiro;
- Retira a peça do vector *pecas* (relembra-se que este vector apenas contém peças que ainda não foram jogadas).

#### 4.5 Avaliação do Tabuleiro e Final do Jogo

O decorrer do jogo processa-se em *void Jogo::jogar()* com um laço incondicional que é quebrado a partir de dentro em situações de final de jogo conforme a lógica seguinte:

- Executa jogada (*bool Jogo::executaJogada()*);
- Imprime tabuleiro resultante (*void Tabuleiro::imprimirTabuleiro()*);
- Avalia se o estado do tabuleiro representa uma vitória (*bool Tabuleiro::vencedor()*). Se sim declara o jogador actual vencedor e quebra;
- Avalia se o estado do tabuleiro representa um empate (*bool Tabuleiro::estaCheio()*). Se sim anuncia empate e quebra;
- Dá a vez ao jogador seguinte (*void Jogo::trocaJogador()*).

### 5. Conclusões e Perspectivas de Desenvolvimento

Fica assim descrita toda a lógica de jogo implementada. Pensa-se que o módulo de jogo propriamente dito se encontra já num estágio de maturação bastante elevado. No entanto, apesar de se considerar que o desenvolvimento da representação interna está terminado, reconhece-se que esta não é uma aplicação apresentável ao público e que carece de uma boa interface bem como de mais opções a nível do menu como a possibilidade de guardar o jogo para continuar mais tarde, etc.

## A. Bibliografia

Harvey & Paul Deitel. C++ How to Program (5th Edition). Alexandria, VA: Prentice Hall, 2005.

“C++ Reference” 15 Maio. 2008

<<http://www.cppreference.com>>

“Quarto (board game) - Wikipedia, the free encyclopedia.” 15 Maio. 2008

<[http://en.wikipedia.org/wiki/Quarto\\_\(board\\_game\)](http://en.wikipedia.org/wiki/Quarto_(board_game))>.

“Programação 2 Wiki” 15 Maio. 2008

<<http://gnomo.fe.up.pt/~pmcm/pmwiki.php/Disciplinas/PROG2>>

## B. Anexos – Código Fonte

```
// Peca.h
// Definição da classe Peca

#include <iostream>
#include <string>
using namespace std;

#ifndef PECA_H_
#define PECA_H_

class Peca {
public:
    Peca(); // Cria não-peças que preenchem o tabuleiro inicial
    Peca(bool, bool, bool, bool);
        // Cria peças reais segundo descrição passada por valor
    ~Peca();
    string representacaoPeca() const;
        // Retorna representação (3 caracteres)
    bool operator==(const Peca&) const;
        // Compara descrição de duas peças

        // Dados:
    bool grande, // Descrição total da peça
        branca,
        redonda,
        macica,
        naoPeca; // true apenas para não-peças
};

ostream& operator<<(ostream &, Peca); // Declaração

#endif /*PECA_H_*/

/* =====
Representação de peças reais de jogo;
Permite também criação de não-peças para preenchimento do tabuleiro;

Cada peça tem uma de duas característica relativa a quatro qualidades:
- Tamanho: Grande ou Pequena;
- Cor: Branca ou Preta;
- Forma: Redonda ou Quadrada;
```

- Preenchimento: Maciça ou Oca;

Acrescentou-se uma 5ª qualidade:

- Tipo: não-peça ou Peça
  - O objectivo é poder usar peças que não o são no preenchimento inicial da matriz tabuleiro;

Nota relativa ao construtor de não-peças: os valores dados a grande, branca, redonda e macica servem apenas para simplificar situações de comparação não tendo nenhum significado real;

Modo de representação:

- 1º Character (Forma): '(' redonda, '[' quadrada
- 2º Character (Tamanho + Cor): 'b' branca, 'p' preta
  - : <maiúscula> grande, <minúscula> pequena
- 3º Character (Preenchimento): '!' macica, ';' oca

Implementação permite utilização de "<<" para imprimir representação da peça e "==" para comparar peças.

Nota relativa à política de privacidade: Gets e Sets foram considerados desnecessários uma vez que dificultariam o uso da classe e se verificou que não é possível atribuir valores inválidos aos membros-dado da classe. É possível atribuir valores que não queremos mas não é possível atribuir valores inválidos.

===== \*/

```
// Peca.cpp
// Implementação dos métodos da classe Peca

#include "Peca.h"

Peca::Peca() { // Cria não-peças
    grande = false; branca = false; redonda = false;
    macica = false; naoPeca = true;
}

Peca::Peca(bool grd, bool brc, bool red, bool mac) {
    grande = grd; branca = brc; redonda = red;
    macica = mac; naoPeca = false;
}

Peca::~Peca() { }

string Peca::representacaoPeca() const {
    string ret; // Variável a retornar

    if(naoPeca) return " ";
    else {
        if(redonda) ret += "(";
        else ret += "[";

        if(branca) {
            if(grande) ret += "B";
            else ret += "b";
        } else {
            if(grande) ret += "P";
            else ret += "p";
        }

        if(macica) ret += "!";
        else ret += ";";
    }
}
```

```

    }
    return ret;
}

bool Peca::operator==(const Peca &piece) const {
    if ( grande == piece.grande &&
        branca == piece.branca &&
        redonda == piece.redonda &&
        macica == piece.macica &&
        naoPeca == piece.naoPeca )
        return true;
    else
        return false;
}

ostream& operator<<(ostream& os, Peca p) {
    os << p.representacaoPeca();
    return os;
}

```

---

```

// Tabuleiro.h
// Definição da classe Tabuleiro

#include <vector>
#include "Peca.h"
using namespace std;

#ifndef TABULEIRO_H_
#define TABULEIRO_H_
#define TABULEIRO_DIM 4          // Tabuleiro 4x4

class Tabuleiro {
public:
    Tabuleiro(const Peca &);
    // Cria tabuleiro preenchido com cópias da peça passada por referência
    ~Tabuleiro();
    void imprimeTabuleiro() const;          // Imprime tabuleiro
    bool estaLivre(int row, int col) const;
        // Testa se posição (row-1,col-1) está vazia
    void inserePeca(int row, int col, const Peca &p);
        // Insere peça p na posição (row-1,col-1)
    bool vencedor() const;
        // Determina se tabuleiro se encontra em estado vencedor
    bool estaCheio() const;          // Testa se tabuleiro está cheio
private:
        // Dados:
    const Peca naoPeca;          // Guarda a descrição de não-peças
    vector< vector<Peca> > matriz;
        // Matriz tabuleiro como vector de vectores de peças
};

#endif /*TABULEIRO_H_*/

/* =====
Representação do tabuleiro real de jogo;

```

O tabuleiro é visto como uma matriz de peças. Ele deve ser inicialmente preenchido com não-peças (abstracção adoptada para impedir valores desconhecidos). Os índices da matriz estão totalmente encapsulados devendo as classes exteriores usar a numeração natural começada em

1 e não em 0. As classes exteriores só podem passar valores válidos;

Antes de se colocar uma peça numa determinada posição devemos assegurar-mo-nos que ela não está já preenchida com uma peça real.

A detecção de situações vencedoras aproveita a representação booleana das características das peças. O que se faz é percorrer linhas, colunas, diagonais e quadrados 2x2 com 5 variáveis auxiliares que vão sendo incrementadas ou mantidas conforme a peça existente na célula em questão. Tomemos como exemplo a variável `grd` (grande). Em cada célula a variável é incrementada se a peça aí existente for grande e mantida se a peça em questão for pequena (equivalente a `grande = false`). Consideremos duas situações:

- A variável, depois de ser lido o bloco apresenta um valor diferente de 0 ou 4. Ficamos a saber que as peças nem são todas grandes nem são todas pequenas;
- A variável, depois de ser lido o bloco apresenta um valor igual a 0 ou 4. Ficamos a saber que uma das características, grande ou pequena, se manteve ao longo de todas as peças do bloco e estamos portanto numa situação de vitória.
- Salvar-se a situação em que a variável `npc` (`naoPeca`) apresente valor positivo. Daí concluímos que o bloco não está totalmente preenchido e invalida-se a afirmação do ponto anterior.

As situações de empate estão cobertas por `estaCheio` cuja função é determinar se o tabuleiro se encontra totalmente preenchido com peças reais.

===== \*/

```
// Tabuleiro.cpp
// Implementação dos métodos da classe Tabuleiro

#include "Tabuleiro.h"

Tabuleiro::Tabuleiro(const Peca &my_naoPeca)
    : naoPeca(my_naoPeca)
{
    matriz.resize(TABULEIRO_DIM); // Dimensiona linhas da matriz

    for(int row = 0; row < TABULEIRO_DIM; row++) {
        matriz[row].resize(TABULEIRO_DIM);
        // Dimensiona colunas para cada linha da matriz

        for(int col = 0; col < TABULEIRO_DIM; col++) {
            matriz[row][col] = naoPeca; // Preenche com naoPeca
        }
    }
}

Tabuleiro::~Tabuleiro() {}

void Tabuleiro::imprimeTabuleiro() const {
    cout << "  #";
    for(int i = 0; i < TABULEIRO_DIM; i++) {
        cout << "  " << (char)(i+(int)'A') << "  #";
        /* "  #_A_#_B_#_C_#_D_#" */
    }
    cout << endl;

    for(int i = 1; i <= TABULEIRO_DIM; i++) {
```

```

cout << " |";
for(int j = 1; j <= TABULEIRO_DIM; j++) {
    cout << " |";
    /* " | | | | " */
}
cout << endl;

cout << " " << i;
for (int j = 1; j <= TABULEIRO_DIM; j++) {
    cout << " " << matriz[i-1][j-1];
    /* " 1 [p; | | [B! | 1" */
    if(j < TABULEIRO_DIM) cout << " |";
}
cout << " " << i << endl;

cout << " |";
for(int j = 1; j <= TABULEIRO_DIM; j++) {
    cout << " |";
    /* " | | | | " */
}
cout << endl;

if(i < TABULEIRO_DIM) {
    cout << " -#";
    for(int j = 1; j <= TABULEIRO_DIM; j++) {
        if(j < TABULEIRO_DIM) cout << "-----|";
        else cout << "-----#";
    }
    /* " #-----|-----|-----|-----#" */
}
cout << endl;
}

cout << " #";
for (int i = 0; i < TABULEIRO_DIM; i++) {
    cout << "___" << (char)(i+(int)'A') << "___#";
    /* " #_A_#_B_#_C_#_D_#" */
}
cout << endl;
}

bool Tabuleiro::estaLivre(int row, int col) const {
    return matriz[row-1][col-1] == naoPeca;
    // Posição livre se peça aí existente for naoPeca
}

void Tabuleiro::inserePeca(int row, int col, const Peca &p) {
    matriz[row-1][col-1] = p;
}

bool Tabuleiro::vencedor() const {
    // 1: Testar linhas, colunas, diagonais e quadrados 2x2
    // 2: Verificar no fim de cada leitura se variáveis auxiliares
    // estão em estado vencedor
    // 3: Retornar se tabuleiro está em estado vencedor
    int row, col;
    bool win = false; // Ainda não se verificou situação vencedora

    int grd = 0, brc = 0, red = 0, mac = 0, npc = 0;
}

```

```

// Variáveis auxiliares a 0

// Testa linhas
// Reinicializa auxiliares a cada nova linha
for(row = 0; row < TABULEIRO_DIM; grd = 0, brc = 0, red = 0, mac = 0, npc
=
0, row++) {

    for(col = 0; col < TABULEIRO_DIM; col++) {
        // Incrementa ou mantém auxiliares em cada posição
        grd += matriz[row][col].grande;
        brc += matriz[row][col].branca;
        red += matriz[row][col].redonda;
        mac += matriz[row][col].macica;
        npc += matriz[row][col].naoPeca;
    }

    // Testa se auxiliares estão em estado vencedor
    win = (!grd % TABULEIRO_DIM) ||
        !(brc % TABULEIRO_DIM) ||
        !(red % TABULEIRO_DIM) ||
        !(mac % TABULEIRO_DIM)
        && (!npc);

    if(win) return true; // Se sim retorna true
} // Linha seguinte

grd = 0; brc = 0; red = 0; mac = 0; npc = 0; // Variáveis auxiliares a 0

// Testa colunas
// Mesma estrutura que teste anterior
for(col = 0; col < TABULEIRO_DIM; grd = 0, brc = 0, red = 0, mac = 0, npc
=
0, col++) {

    for(row = 0; row < TABULEIRO_DIM; row++) {
        grd += matriz[row][col].grande;
        brc += matriz[row][col].branca;
        red += matriz[row][col].redonda;
        mac += matriz[row][col].macica;
        npc += matriz[row][col].naoPeca;
    }

    win = (!grd % TABULEIRO_DIM) ||
        !(brc % TABULEIRO_DIM) ||
        !(red % TABULEIRO_DIM) ||
        !(mac % TABULEIRO_DIM)
        && (!npc);

    if(win) return true;
} // Coluna seguinte

grd = 0; brc = 0; red = 0; mac = 0; npc = 0; // Variáveis auxiliares a 0

// Testa diagonal principal
for(row = 0, col = 0; col < TABULEIRO_DIM; row++, col++) {
    grd += matriz[row][col].grande;
    brc += matriz[row][col].branca;
    red += matriz[row][col].redonda;
    mac += matriz[row][col].macica;
    npc += matriz[row][col].naoPeca;
}

win = (!grd % TABULEIRO_DIM) ||

```

```

        !(brc % TABULEIRO_DIM) ||
        !(red % TABULEIRO_DIM) ||
        !(mac % TABULEIRO_DIM)
        && (!npc);

if(win) return true;

grd = 0; brc = 0; red = 0; mac = 0; npc = 0;    // Variáveis auxiliares a 0

                                // Testa diagonal não principal
for(row = 0, col = TABULEIRO_DIM - 1; col >= 0; row++, col--) {
    grd += matriz[row][col].grande;
    brc += matriz[row][col].branca;
    red += matriz[row][col].redonda;
    mac += matriz[row][col].macica;
    npc += matriz[row][col].naoPeca;
}

win =  (!(grd % TABULEIRO_DIM) ||
        !(brc % TABULEIRO_DIM) ||
        !(red % TABULEIRO_DIM) ||
        !(mac % TABULEIRO_DIM)
        && (!npc);

if(win) return true;

grd = 0; brc = 0; red = 0; mac = 0; npc = 0;    // Variáveis auxiliares a 0

                                // Testa quadrados 2x2
                                // Percorre posições com índice de linha e de coluna
                                // entre 0 e 2. Assume cada posição como vértice
superior
                                // esquerdo do quadrado 2x2
                                // Reinicializa auxiliares a cada novo quadrado
for(row = 0; row < TABULEIRO_DIM-1; row++) {
    for(col = 0; col < TABULEIRO_DIM-1; grd = 0, brc = 0, red = 0, mac =
                                                0, npc = 0, col++) {

        grd = matriz[row][col].grande + matriz[row][col+1].grande +
[col+1].grande;
                matriz[row+1][col].grande + matriz[row+1]

        brc = matriz[row][col].branca + matriz[row][col+1].branca +
[col+1].branca;
                matriz[row+1][col].branca + matriz[row+1]

        red = matriz[row][col].redonda + matriz[row][col+1].redonda +
                matriz[row+1][col].redonda + matriz[row+1][col+1].redonda;

        mac = matriz[row][col].macica + matriz[row][col+1].macica +
[col+1].macica;
                matriz[row+1][col].macica + matriz[row+1]

        npc = matriz[row][col].naoPeca + matriz[row][col+1].naoPeca +
                matriz[row+1][col].naoPeca + matriz[row+1][col+1].naoPeca;

        win =  (!(grd % TABULEIRO_DIM) ||
                !(brc % TABULEIRO_DIM) ||
                !(red % TABULEIRO_DIM) ||
                !(mac % TABULEIRO_DIM)
                && (!npc);

```



```

        if(win) return true;
    }
}
return false;
// Se chega a esta ponto então o tabuleiro não está num estado vencedor
}

```

```

bool Tabuleiro::estaCheio() const {
    for (int row = 1; row <= TABULEIRO_DIM; row++)
        for (int col = 1; col <= TABULEIRO_DIM; col++)
            if(estaLivre(row, col))
                // Invoca estaLivre para cada posição do tabuleiro

                return false;

    return true;
}

```

---

```

// Jogador.h
// Definição da classe Jogador

```

```

#include <iostream>
#include <string>
using namespace std;

```

```

#ifndef JOGADOR_H_
#define JOGADOR_H_

```

```

class Jogador {
public:
    Jogador();
    ~Jogador();
    void setNome(string);
    string getNome() const;
private:
    string nome; // Dados: // Nome do jogador
};

```

```

ostream& operator<<(ostream&,Jogador); // Declaração

```

```

#endif /*JOGADOR_H_*/

```

```

/* =====
Representação do jogador real;
Única característica relevante é o nome;
Implementação permite utilização de "<<" para imprimir nome do
jogador.
===== */

```

```

// Jogador.cpp
// Implementação dos métodos da classe Jogador

```

```

#include "Jogador.h"

```

```

Jogador::Jogador() {}

```

```

Jogador::~~Jogador() {}

```

```

void Jogador::setNome(string n) {
    nome = n;
}

string Jogador::getNome() const {
    return nome;
}

ostream& operator<<(ostream& os, Jogador j) {
    os << j.getNome();
    return os;
}

```

---

```

// MyException.h
// Definição da classe MyException

```

```

#include <string>
#include <iostream>
using namespace std;

```

```

#ifndef MYEXCEPTION_H_
#define MYEXCEPTION_H_

```

```

class MyException {
public:
    MyException(string);
    ~MyException();
    string getMensagem() const;
private:
    // Dado:
    const string mensagem; // Mensagem a ser apresentada ao utilizador
};

```

```

ostream& operator<<(ostream&, MyException); // Declaração

```

```

#endif /*MYEXCEPTION_H_*/

```

```

/* =====
    Suporte para o lançamento de exceções
    Implementação permite utilização de "<<" para imprimir mensagens
    de erro.
    ===== */

```

```

// MyException.cpp
// Implementação dos métodos da classe MyException

```

```

#include "MyException.h"

```

```

MyException::MyException(string m)
    : mensagem(m) {}

```

```

MyException::~MyException() {}

```

```

string MyException::getMensagem() const {
    return mensagem;
}

```

```

ostream& operator<<(ostream& os, MyException e) {
    os << e.getMensagem();
    return os;
}

```

```
}
```

---

```
// Jogo.h
// Definição da classe Jogo

#include <iostream>
#include <ctime>
#include <cstdlib>
#include <vector>
#include <string>
#include <fstream>
#include "Peca.h"
#include "Jogador.h"
#include "Tabuleiro.h"
#include "MyException.h"
using namespace std;

#ifndef JOGO_H_
#define JOGO_H_

class Jogo {
public:
    Jogo();          // Cria novo jogo
    ~Jogo();
    void jogar();   // Desenvolvimento do jogo
private:
                                // Dados:
    Tabuleiro Tab;          // Tabuleiro
    Jogador jogadores[2];  // Array de jogadores
    int jogadorActivo;
                        // Jogador actual (geração aleatória de primeiro valor)
    vector<Peca> pecas;    // Vector de peças que ainda não foram jogadas

                                // Membros-função específicos:
    void trocaJogador();   // Dá a vez ao jogador seguinte
    void listarPecas();   // Lista peças não jogadas
    int interChar1(char);
                        // Interpreta significado de caracter lido para peça
    int interChar2(char);
                        // Interpreta significado de caracter lido para linha
    int interChar3(char);
                        // Interpreta significado de caracter lido para coluna
    bool executaJogada(); // Executa jogada: pergunta peça e posição
};

#endif /*JOGO_H_*/

/* =====
    Representação da dinâmica de jogo;

    A geração de um objecto do tipo Jogo implica a criação de um tabuleiro
    preenchido com não-peças. É criado também um array de jogadores.
    A variável jogadorActivo é de extrema importância
    uma vez que simplifica a maneira como se troca de jogador.
    A esta variável é dado no início um valor aleatório simulando o
    lançamento do dado para ver quem joga primeiro. O vector pecas guarda
    as peças ainda não jogadas e é inicializado com as 16 peças do jogo;

    Procedimento para cada jogada:
    - O adversário escolhe uma peça
```

- Jogador actual escolhe uma posição (verificar se posição já está ocupada)
- A peça é colocada no tabuleiro
- A peça é retirada do vector pecas
- Percorre-se o tabuleiro para saber se o jogador actual realizou jogada vencedora
- Percorrer-se o tabuleiro para determinar se ainda existem casas livres. Se tal não acontecer é anunciado empate.
- Se nenhum dos dois últimos pontos se verificar é dada a vez ao jogador seguinte

===== \*/

```
// Jogo.cpp
// Implementação dos métodos da classe Jogo

#include "Jogo.h"

Jogo::Jogo()
    : Tab(Peca())
{
    string nomeTmp;
    cout << "Nome do Jogador 1: ";
    getline(cin, nomeTmp);
    jogadores[0].setNome(nomeTmp);
    cout << "Nome do Jogador 2: ";
    getline(cin, nomeTmp);
    jogadores[1].setNome(nomeTmp);

    srand(time(NULL)); // Geração aleatória
    jogadorActivo = rand()%2;

    pecas.resize(17); // Não é utilizado pecas[0]

    pecas[1] = Peca(1,1,1,1); pecas[2] = Peca(1,1,1,0);
    pecas[3] = Peca(1,1,0,1); pecas[4] = Peca(1,1,0,0);
    pecas[5] = Peca(1,0,1,1); pecas[6] = Peca(1,0,1,0);
    pecas[7] = Peca(1,0,0,1); pecas[8] = Peca(1,0,0,0);
    pecas[9] = Peca(0,1,1,1); pecas[10] = Peca(0,1,1,0);
    pecas[11] = Peca(0,1,0,1); pecas[12] = Peca(0,1,0,0);
    pecas[13] = Peca(0,0,1,1); pecas[14] = Peca(0,0,1,0);
    pecas[15] = Peca(0,0,0,1); pecas[16] = Peca(0,0,0,0);
    // Preenchido segundo tabela de valores binários
}

Jogo::~~Jogo() { }

void Jogo::trocaJogador() {
    jogadorActivo = ((jogadorActivo + 1) % 2); // Alterna entre 0 e 1
}

void Jogo::listarPecas() {
    cout << endl << "\tPecas por jogar:";
    int ind; vector<Peca>::iterator it;
    for(ind = 0, it = pecas.begin() + 1; it != pecas.end(); ind++, it++) {
        // pecas[0] não tem significado
        if( !(ind % 4) ) {cout << endl << "\t\t\t";} // Quebra de linha
        cout << (char)('a'+ind) << ":" << (*it) << " ";
    }
    cout << endl << endl;
}
}
```

```

int Jogo::interChar1(char c) {
    // Interpreta caracter lido relativo à peça a jogar
    // Retorna índice da peça no vector pecas
    // Lança exceção se o caracter estiver fora de âmbito
    char max = (char)((pecas.size()-1) + 'a' - 1);
    // Calcula letra da última peça da lista apresentada ao utilizador
    if(c >= 'a' && c <= max) // Se valor válido
        return (c % 'a') + 1; // retorna índice da peça no vector pecas
    else if(c >= 'A' && c <= ('A' + (max % 'a'))
        // Tornar a aplicação mais permissiva
        return (c % 'A') + 1;
    else { // Se valor inválido lança exceção com a mensagem
adequada
        string msg = "para indicar a peça use letras de 'a' a "; msg += "'";
        msg += max; msg += "'";
        throw MyException(msg);
    }
}

int Jogo::interChar2(char c) {
    // Interpreta caracter lido relativo a linha
    // Retorna índice de linha da matriz
    // Lança exceção se o caracter estiver fora de âmbito
    if(c >= '1' && c <= '4')
        return c % '0';
    else throw MyException("para indicar a linha use 1,2,3 ou 4");
}

int Jogo::interChar3(char c) {
    // Interpreta caracter lido relativo a coluna
    // Retorna índice de coluna da matriz
    // Lança exceção se o caracter estiver fora de âmbito
    if(c >= 'A' && c <='D')
        return (c % 'A') + 1;
    else if(c >= 'a' && c <= 'd') // Tornar a aplicação mais permissiva
        return (c % 'a') + 1;
    else throw MyException("para indicar a coluna use A,B,C ou D");
}

bool Jogo::executaJogada() {
    // 1: Peça a jogar?
    // 2: Onde jogar?
    // 3: Actualizar tabuleiro e vector de peças não jogadas
    // Tratamento exaustivo de exceções
    vector<Peca>::iterator indPeca; string charPeca;
    int row, col; string charPos;
    bool erro; // Permite repetir laços enquanto ocorrerem exceções

    cout << jogadores[jogadorActivo] << " joga!" << endl;
    listarPecas();

    do { // Repete enquanto houver erros nos dados introduzidos
        cout << jogadores[(jogadorActivo + 1)%2] << " escolhe a peça" <<
endl;
        erro = false; // Ainda não se verificaram erros
        try {
            cout << "letra da peça?(z para sair):"; getline(cin, charPeca);
            // Usado getline uma vez que "cin>>" traz problemas à manutenção
            // da estabilidade da aplicação
            if( charPeca.length() != 1) throw MyException("deve

```

```

especificar                                     exatamente um
caracter");

        // Queremos 1 e 1 só caracter
        if( charPeca[0] == 'z' || charPeca[0] == 'Z') return false;
// Retorna booleano para que intenção de sair possa ser transmitida
// às duas funções que se encontram acima na ordem de chamada
        indPeca = pecas.begin() + interChar1(charPeca[0]);
        // Coloca o iterador na posição da peça escolhida
    }
    catch(MyException err) { // Avisar o utilizador
        cout << endl << "\tAtenção: " << err << endl << endl;
        erro = true;        // Para que ciclo se repita
    }
} while(erro);

do { // Repete enquanto houver erros nos dados introduzidos
    cout << jogadores[jogadorActivo] << " onde jogar " << (*indPeca) <<
        " ?" << endl;

    erro = false;        // Ainda não se verificaram erros
    try {
        cout << "Posição?:"; getline(cin, charPos);
        if( charPos.length() != 2) throw MyException("para especificar
            posição deve digitar, por exemplo, A1");
        // Queremos 2 e só 2 caracteres
        if( charPos[0] >= '0' && charPos[0] <= '9' && (charPos[1] <
            || charPos[1] > '9')) {
            row = interChar2(charPos[0]); col = interChar3(charPos[1]);
        } // Se primeiro caracter é numérico e o segundo não,
passar valores aos respectivos interChars
        else if( charPos[1] >= '0' && charPos[1] <= '9' && (charPos[0]
            < '0' || charPos[0] > '9')) {
            row = interChar2(charPos[1]); col = interChar3(charPos[0]);
        } // Se segundo caracter é numérico e o primeiro não,
passar valores aos respectivos interChars
        else throw MyException("para especificar posição deve
            digitar,                                     por
            exemplo, A1");
        // Se nenhuma das situações anteriores ocorreu lançamos excepção

        if (!Tab.estaLivre(row,col)) throw MyException("posição
            escolhida já esta ocupada!");
        // Se posição escolhida já está ocupada lançamos excepção
    }
    catch(MyException err) {
        // Mesma estrutura que a sua homóloga anterior
        cout << endl << "\tAtenção: " << err << endl << endl;
        erro = true;
    }
} while(erro);

Tab.inserePeca(row, col, *indPeca); // Coloca a peça no tabuleiro
pecas.erase(indPeca);                // Retira a peça de pecas
return true;                          // O utilizador não pediu para
sair
}

void Jogo::jogar() {
    // 1: Executa jogada
    // 2: Testa se vencedor
    // 3: Testa se empate
    // 4: Dá a vez ao jogador seguinte

```

```

Tab.imprimeTabuleiro(); // Imprime aspecto inicial do tabuleiro

while(1) { // Quebrado a partir de dentro
    if (!executaJogada()) return;
        // Executa jogada
        // Se jogador pediu para sair voltamos ao menu
    Tab.imprimeTabuleiro();
        // Imprime tabuleiro resultante da última jogada

    if(Tab.vencedor()) {
        // Se estado do tabuleiro é vencedor anuncia vitória
        cout << jogadores[jogadorActivo] << " Ganhou!" << endl;
        break;
    }
    if(Tab.estaCheio()) {
        // Se tabuleiro totalmente preenchido anuncia empate
        cout << "Empate..." << endl;
        break;
    }

    trocaJogador(); // Dá a vez ao jogador seguinte
} // Repete até termos situação de fim de jogo
cout << "A terminar..." << endl;
}

// Fim de implementação da classe

// Implementação de funções auxiliares:

void regras() {
    // Imprime regras de jogo a partir de ficheiro
    regras.txt
    try {
        char c;
        ifstream fich("regras.txt");
        if (fich == NULL) throw MyException("impossível mostrar as regras de
jogo");
        // Se ocorreu erro a abrir ficheiro lança excepção
        while(fich.get(c)) cout << c;
    }
    catch(MyException err) { // Avisar utilizador
        cout << endl << "\tAtenção: " << err << endl << endl;
    }
}

void menu() {
    bool erro;
    do { // Repete enquanto houver erros nos dados introduzidos
        try {
            erro = false; // Ainda não se verificaram erros
            string o;
            cout << "\t\tQuarto!" << endl
            << "\ta: Regras" << endl
            << "\tb: Jogar" << endl
            << "\tz: Sair" << endl << endl;

            cout << "Opção?:"; getline(cin,o);
            if( o.length() != 1) throw MyException("digite apenas a, b ou
z");

            switch( o[0] ) {

```

```

        case 'a': case 'A': regras(); break;
        case 'b': case 'B': {Jogo novoJogo; novoJogo.jogar();}
                               break;
        case 'z': case 'Z': cout << endl << "\tAté breve!\n";
                               exit(0);
        default: throw MyException("digite apenas a, b ou z");
    }
}
catch(MyException err) {
    erro = true; // Para que ciclo se repita
    cout << endl << "\tAtenção: " << err << endl << endl;
}
} while(erro);
}

int main() {
    while(1) menu();
}

```