Recibo do Trabalho Final de Introdução à Programação, 1º ano IGE 2002/2003

Nomes: Rui Filipe Borges Varela
Números: 22175

Assinatura do Docente: _____

Introdução à Programação
ISCTE, 2002/2003
Trabalho Final

Grupo nº: 11
Nome: Rui Filipe Borges Varela Nº: 22175

- ## Relatório

De forma muito sucinta posso estruturar o programa em quatro classes que se relacionam entre si de forma hierárquica, no topo temos a classe jogo que tem um membro privado do tipo Tabuleiro, que de forma evidente constitui o tabuleiro de jogo, por seu lado este tabuleiro contém um vector de 8x8 células, e cada célula contém uma peça. Desta forma tem,os no tabuleiro varias peças do tipo sem_peça, que foi a maneira utilizada para representar um célula desocupada. Fora do encapsulamento das Classes foi incluída um única função (n tendo em conta o main)

Com o intuito de tornar o código relativamente dinâmico foram utilizados entre outros, enumerações para estados (i.e. EstadoDePosicao, EstadoDeJogo), nesse mesmo sentido tentamos afastar o controlo do fluxo do programa da classe tabuleiro, que funciona como um suporta para o jogo, não realizando só por si o jogo, pensando no futuro é mais fácil reescrever outra classe jogo que, de forma mais inovadora controle a classe Tabuleiro, para que não fiquem a pensar que é texto em vão, uma solução vantajosa seria arranjar um sistema de selecção de posição que utilizasse o rato.

- ## Extras

• Redimensionar

O primeiro extra é o facto da janela poder redimensionada, sendo que é actualizada de foma eficiente, neste sentido forma estabelecidas medidas base (3x7) como medidas mínima para o tamanho de uma célula. No seguimento desta funcionalidade, foi criada uma variável (aspecto),na classe Tabuleiro, que serve de mediador para calcular as medidas necessárias á função que exigem desenho, do tabuleio

O segundo é o facto das damas realizarem jogadas tal como num jogo de damas real

• Damas Normais

O segundo extra é o facto das damas se comportarem como damas regulares, isto é podem dar saltos maiore que os peõs. Porém esta funcionalidade exigiu que fosse devidamente implementado o seu suporte no que diz respeito a cumprimento de um requisito do programa, dado um posição saber se é possível chegar a outra.

Sendo que é a função Tabuleiro::simulacaoDePercurso(..) é mais complexa do jogo passo a explicar:

Esta função funcioana em cooperação com Tabuleiro::podePercorrer (), a primeira prepara a execução da outra, no caso de se dama é necessária uma atenção especial neste contexto visto que a peça pode executar um salto(onde toma uma peça) e só depois começa a comer em sequência, como tal foi necessário verificar as possibilidades desse salto. A primeira função inicia um especie de simulação que vai percorrendo as diagonais á dama até encontrar uma posição onde posso haver seqüência, queno encontra passa controlo para a função Tabuleiro::podePercorrer (), esta indiace se é possível através de capturas chegar á posição desejada. O termo simulação n é usado em vão, visto que á medida que se vai testanto o

percurso as pseudo-peças comidas vão sendo marcadas como sem_peça, e quando é necessário voltam a ser marcadas com o verdadeiro estado.

- **IPC++**

Com este código é possível jogar contra outra máquina (embora dividindo processos), através de ipc++.

O protocolo é muito simples, de um lado é criado um vector de posições relativas à jogada, esse vector é posteriormente enviado. Este protocolo envia apenas a posições por onde a peça "em marcha" efectivamente passou.

Do outro lado o vector é recebido construído recebendo (linha/coluna) referentas ás posições recebidas, deposi estas posições são tratadas e executadas

-

- **Listagem**

damas_entrega_final.C

- **Programa principal (`damas_entrega_final.C`)**

```
/*********************************************************
                          ISCTE
           Informática e Gestão de Empresas

              Intodução à Programação

             Jogo Das Damas (2ª Fase)

             Trabalho realizado por:
             Rui Varela nº 22175 IA2
             Grupo 11

            the_true_ryven@hotmail.com
 *********************************************************/
#include <iostream>
#include <vector>
#include <Slang++/slang.H>
#include <IPC++/mensageiro.H>
#include <cassert>
#include <cmath>

using namespace std;
using namespace Slang;
using namespace IPC;

Ecra::ObjectoCor const cor_azul(azul,azul); // tabuleiro preto
Ecra::ObjectoCor const cor_ciano(ciano,ciano);// tabuleiro branco
Ecra::ObjectoCor const cor_branco(branco,branco); //peoes
Ecra::ObjectoCor const cor_preto(preto,preto); //peoes
Ecra::ObjectoCor const cor_damas_pretas(branco,preto);
Ecra::ObjectoCor const cor_damas_brancas(preto,branco);

Ecra::ObjectoCor const cor_texto_brancas(preto,ciano);
Ecra::ObjectoCor const cor_texto_pretas(branco,azul);
Ecra::ObjectoCor const cor_texto(branco,preto);

Ecra::ObjectoCor const cor_selecao(vermelho,vermelho);
Ecra::ObjectoCor const cor_selecao_origem(magenta,magenta);

enum CorDeQuadrado {
    quadrado_preto,
    quadrado_branco,
    quadrado_azul,
    quadrado_ciano,
    quadrado_seleccao_origem,
    quadrado_seleccao_destino
};
/** Procedimento que desenha quadrados
    @pre altura >= 0, largura >= 0, posicionamento >= Posicao(0,0)
    @post quadrado kom as dimendoes altura largura, desenhado a partir
de
```

```
    posicionamenteo e de cor cor. */
void desenhaQuadrado(Posicao const &posicionamento, CorDeQuadrado const
cor do quadrado,int const altura, int const largura){
    assert((altura >= 0) and (largura >= 0) and
(posicionamento.linha() >= 0) and (posicionamento.coluna() >= 0));
    ecra << cursor(posicionamento + Dimensao(-1,largura));
    for(int linha_actual = 0; linha_actual != altura; ++linha_actual)
    {
        ecra << ecra.posicaoDoCursor() + Dimensao(1,-largura);
        for(int coluna actual =0; coluna actual != largura;
++coluna actual)
        {
            switch (cor_do_quadrado)
            {
              case quadrado_preto:
                ecra << cor_preto << ' ';
                break;
              case quadrado_branco:
                ecra << cor_branco << ' ';
                break;
              case quadrado_azul:
                ecra << cor_azul << ' ';
                break;
             case quadrado_ciano:
               ecra << cor_ciano << ' ';
               break;
              case quadrado_seleccao_destino:
                ecra << cor_selecao << ' ';
                break;
              case quadrado_seleccao_origem:
                ecra << cor_selecao_origem << ' ';
                break;
            }
        }
    }
}
/** Representa um peça.
    @invariant V. */
class Peca {
public:
    enum Tipo {
        sem_peca,
        dama,
        peao
    };
    enum Jogador {
        brancas,
        pretas
    };
    /** Constrói uma nova peça segunto tipo_da_peca e a sua cor
        @pre V.
        @post Peca = tipo da peca/cor*/
    Peca(Tipo const tipo_da_peca,Jogador jogador);
    /** Simula Peça eliminada,(permitindo simulações de jogadas)
        @pre V.
        @post (this*).tipo() = sem_peca */
    inline void simulaPecaEliminada();
    /** Para a simulação de peça eliminada
        @pre V.
```

```
        @post (this*).tipo() = tipo da peca */
    inline void paraSimulacao();
    /** Devolve o tipo desta peça.
        @pre V.
        @post tipo = tipo desta peça */
    inline Tipo tipo() const;
    /** Devolve o jogador a que esta peça pertence.
        @pre V.
        @post jogador = jogador desta peça */
    inline Jogador jogador() const;
    /** Devolve a cor, que é usada para desenhar desta peça.
        @pre V.
        @post corDestaPeca = cor(a pintar) da peça */
    inline CorDeQuadrado corDestaPeca() const;
private:
    bool simulacao a decorrer;
    CorDeQuadrado cor da peca;
    Tipo tipo da peca;
    Jogador jogador da peca;
};
/** Representa um celula.
    @invariant (posicao no ecra.linha() >= 0) e
(posicao no ecra.coluna() >= 0)
    e (altura da celula >= 3) e (largura da celula >= 7) */
class Celula {
public:
    /** Constrói uma nova celula
        @pre V.
        @post Celula /cor */
    inline Celula(CorDeQuadrado const cor a usar,Peca::Tipo const tipo,
Peca::Jogador const jogador);
    /** Desenha a celula
        @pre V.
        @post Celula desenhada*/
    inline void desenha() const;
    /** Desenha a peca agregada à celula
        @pre V.
        @post peca desenhada */
    void desenhaPeca() const;
    /** Desenha a selecao de destino (caracter central)
        @pre V.
        @post desenhaSeleccaodestino =  1 caracter no centro da celula
*/
    inline void desenhaSeleccaoDestino() const;
    /** Desenha a selecao de Origem (pinta por cima da celula)
        @pre V.
        @post desenhaSeleccaoOrigem =  celula pintada com a cor de
seleccao */
    inline void desenhaSeleccaoOrigem() const;
    /** Actualiza os elementso de referencia da celula
        @pre (0 <= linha <= 7) and (0 <= coluna <= 7) and (altura >= 3)
and (largura >= 7).
        @post actualiza as referencias da celula
(posicao,altura,largura) */
    inline void actualizaMedidas(int const linha,int const coluna,int
const altura, int const largura);
    /** Substitui a peça da celula
        @pre V.
        @post peca da celula = nova peca */
```

```
    inline void substituiPeca(Peca const& nova peca);
    /** Eliminia a peça agrgada á celula
        @pre V.
        @post peca da celula.tipo() = sem peca */
    inline void eliminaPeca();
    /** Simula a eliminação da peça da celula
        @pre V.
        @post peca da celula.tipo() = sem peca */
    inline void simulaEliminaPeca();
    /** Pára a simulaão de eliminação da peça da celula
        @pre V.
        @post peca_da_celula = peca_da_celula */
    inline void paraSimulacao();
    /** Devolve a peça da celula ->inspector
        @pre V.
        @post peca() = peca da celula */
    inline Peca peca() const;
    /** Indica se a celula está ocupada
        @pre V.
        @post estaOcupada = estad da celula (ocupada ou desocupada)  */
    inline bool estaOcupada() const;
    /** Devolve o tipo da peça agregada á celula
        @pre V.
        @post tipoDaPecaQueOcupaCelula() = tipo da peça refernete a esta
celula  */
    inline Peca::Tipo tipoDaPecaQueOcupaCelula() const;
    /** Devolve o jogador da peça agregada á celula
        @pre V.
        @post jogadorDaPecaQueOcupaCelula() = jogador da peça rferente a
esta celula  */
    inline Peca::Jogador jogadorDaPecaQueOcupaCelula() const;
private:
    Peca peca da celula;
    Posicao posicao no ecra;
    CorDeQuadrado cor da celula;
    int altura da celula,largura da celula;
    /** Indica se a instância cumpre a condição invariante de classe.
        @pre V.
        @post cumpreInvariante = (posicao no ecra.linha() >= 0) e
(posicao no ecra.coluna() >= 0)
        e (altura da celula >= 3) e (largura da celula >= 7) */
    inline bool cumpreInvariante() const;
};
/** Representa um tabuleiro de damas.
    @invariant celulas do tabuleiro forma um vector 8x8, aspecto >=0*/
class Tabuleiro {
public:
    enum EstadoDePosicao {
        possivel,
        pode_comer,
        impossivel,
        origem sem peca,
        destino ocupado,
        jogador diferente da seleccao,
        mais do que um movimento
    };
    /** Constrói uma novo tabuleiro de damas
        @pre V.
        @post tabuleiro criado*/
```

```
    Tabuleiro();
    /** Desenha o tabuleiro
        @pre V.
        @post tabuleiro desenhado*/
    void desenha() const;
    /** Desenha as peças
        @pre V.
        @post peças desenhado*/
    void desenhaPecas() const;
    /** Desenha as novas referencias de medida do tabuleiro e das
celulas
        @pre V.
        @post aspecto = novo aspecto, celulas kom medidas actualizadas*/
    void actualizaMedidas();
    /** Desenha a seleccao de destino na celula posicao do tabuleiro
        @pre cumpreLimitesDePosicao(posicao do tabuleiro)
        @post SeleccaoDestino() = desenhada em posicao do tabuleiro */
    inline void SeleccaoDestino(Posicao const &posicao_do_tabuleiro)
const;
    /** Activa a seleccao de origem em posicao do tabuleiro
        @pre cumpreLimitesDePosicao(posicao do tabuleiro)
        @post origem selecionada = posicao do tabuleiro,
estado da seleccao de origem = true */
    inline void activaSeleccaoOrigem(Posicao const
&posicao_do_tabuleiro);
    /** Desactiva a seleccao de origem
        @pre V.
        @post estado da seleccao de origem = false */
    inline void desactivaSeleccaoOrigem();
    /** Devolve o estado da seleccao de origem
        @pre V.
        @post haSeleccaoDeOrigem() = estado_da_seleccao_de_origem */
    inline bool haSeleccaoDeOrigem() const; //inspector
    /** Indica se há nova dama, alteradndo de seguida ha nova dama para
false
        @pre V.
        @post haNovaDama() = ha_nova_dama, ha_nova_dama = false */
    inline bool haNovaDama();
    /** Devolve o valor d aspecto do tabuleiro
        @pre V.
        @post valorDoAspecto() = aspecto*/
    inline int valorDoAspecto() const; //inspector
    /** Exibe um aviso de erro konsoante o estado
        @pre V.
        @post mensagem de erro */
    void processaErroDeEstadoDePosicao(EstadoDePosicao estado) const;
    /** Indica se alguma peça de 'jogador' pode ser selccionada
        @pre V.
        @post podeSeleccionarAlgumaPosicao() = pode selecional alguma
casa */
    bool podeSeleccionarAlgumaPosicao(Peca::Jogador jogador) const;
    /** Indica se posicao do tabuleiro pode ser origem de acordo com
jogador actual
        @pre cumpreLimitesDePosicao(posicao do tabuleiro)
        @post podeSerOrigem() = estado da posicao */
    EstadoDePosicao podeSerOrigem(Posicao const
&posicao_do_tabuleiro,Peca::Jogador const jogador_actual,
                                  Peca::Tipo const tipo_da_peca) const;
    /** Indica se destino pode ser destino do movimento de acordo com
```

```
        jogador actual e origem
        @pre cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino)
        @post podeSerOrigem() = estado da posicao */
    EstadoDePosicao podeSerDestino(Posicao const &origem,Posicao const
&destino,Peca::Jogador const jogador_actual
                                    ,bool const e_sequencia);
    /** Efectua a jogada
        @pre podeserOrigem(origem) and podeSerDestino(destino)
        @post jogada efectuada */
    Peca::Tipo executaJogada(Posicao const &origem,Posicao const
&destino);
private:
    /** Simula a execução de um percurso, indicando s é possivel atingir
o destino
        @pre V.
        @post indica se pode chegar ao destino */
    bool simulacaoDePercurso(Posicao const &origem,Posicao const
&destino,Peca::Jogador const jogador_actual);
    /** Informa se através de movimentos de captura é possível chegar ao
destino
        @pre V.
        @post indica se pode chegar ao destino */
    bool podePercorrer(Posicao const &origem,Posicao const
&destino,Peca::Jogador const jogador_actual,Peca::Tipo tipo_de_peca);
    /** Termina a simulaçãod e perurso
        @pre V.
        @post todas as peças voltam a reaparecer com os tipo correcto */
    void terminaSimulacao();
    /** Calcula o numero de peças entre 2 pontos d uma diagonal
        @pre cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino)
        @post numeroDePecasEntreDiagonal() = numero total de peças entre
a origem e o destino */
    int numeroDePecasEntreDiagonal(Posicao const &origem,Posicao const
&destino) const;
   /** Indica se a instância cumpre a condição invariante de classe.
        @pre V.
        @post celulas do tabuleiro == matrix de [8]x[8], aspecto >= 0
*/
    bool cumpreInvariante() const;
    /** Indica se posicao a verificar esta dentro dos limites suportados
pelo celulas do tabuleiro
        @pre V.
        @post posicao a verificar = Posicao(0->7,0->7) */
    inline bool cumpreLimitesDePosicao(Posicao const
&posicao a verificar) const ;
    vector< vector<Celula> > celulas do tabuleiro;
    bool estado da seleccao de origem;
    Posicao origem_selecionada;
    int aspecto;
    bool ha nova dama;
};
/** Representa um jogo de damas.
    @invariant posição destno e origem com valores de vector válidos,
    número de peças no tabuleiro válido*/
class Jogo {
public:
    enum EstadoDeJogo {
```

```
        brancas ganharam,
        pretas ganharam,
        brancas desistiram,
        pretas desistiram,
        empate,
        empate_limite_jogadas
    };
    /** Constrói um novo jogo de damas
        @pre V.
        @post jogo de damas criado */
    Jogo();
    /** Corre o jogo de damas
        @pre V.
        @post jogo terminado*/
    EstadoDeJogo motorDeJogo();
    /** Exibe um mensagem que informa o resultado do jogo
        @pre V.
        @post mensagem com o resulatdo do jogo */
    void processaRsultadoDeJogo(Jogo::EstadoDeJogo const
resultado do jogo) const;
private:
    /** Selecciona uma posicao do tabuleiro
        @pre V.
        @post posicao = posicao de vector escolhida */
    Tecla selecciona(Posicao &posicao);
    /** Selecciona a origem e destino konsoante as regras das damas
        @pre V.
        @post origem e destino escolhidos de forma válida */
    bool seleccaoInteractiva();
    /** Desenha as informações de jogo
        @pre V.
        @post informações e jogo desenhadas */
    void mostraInformacoes() const;
    /** Verifica se o jogador actual ganhou
        @pre V.
        @post jogadorActualGanhou() = estado do jogador (se ganhou ou
n ) */
    inline bool jogadorActualGanhou() const;
    /** Verifica se a jogada gerou novas damas e actualiza a
        estatistica de jogo
        @pre V.
        @post informações de peças actualizadas */
    inline void verificaNovasDamas();
    /** Actualiza as informações de jogo (estatistica), consoante o
        tipo de peca tomada
        @pre V.
        @post informações de peças actualizadas */
    inline void calculaNovasInformacoesDeJogo(Peca::Tipo const
tipo peca tomada, bool const peca tomada);
    /** Indica se a instância cumpre a condição invariante de classe.
        @pre V.
        @post posição destno e origem com valores de vector válidos,
    número de peças no tabuleiro válido */
    inline bool cumpreInvariante() const;
    /** Envia a pilha de posições usando IPC++
        @pre V.
        @post pilha de posições enviada*/
    void despachaPilha();
    /** Executa os movimentos inerentes ás posições da
```

```
pilha de poisições
        @pre pilha de posições devidamente recebida.
        @post jogadas executadas*/
    void executaJogadasDaPilha();
    Peca::Jogador jogador actual;
    Tabuleiro tabuleiro de jogo;
    bool e sequencia;
    Posicao destino,origem;
    int peoes brancas,damas brancas,peoes pretas,damas pretas;
    int contador de jogadas;
    Peca::Jogador meu jogador;
    Mensageiro mensageiro;
    vector<Posicao> pilha de posicoes;
};
Peca::Peca(Tipo const tipo da peca,Jogador cor)
    :simulacao a decorrer(false),tipo da peca(tipo da peca),
jogador da peca(cor)
{
    if (cor == brancas)
        cor da peca = quadrado branco;
    else
        cor da peca = quadrado preto; //sem peca n vai ser desenhado por
issu é indiferente
}
void Peca::simulaPecaEliminada()
{
    simulacao a decorrer = true;
}
void Peca::paraSimulacao()
{
    simulacao a decorrer = false;
}
Peca::Tipo Peca::tipo() const
{
    return simulacao a decorrer ? sem peca : tipo da peca;
}
CorDeQuadrado Peca::corDestaPeca() const
{
    return cor da peca;
}
Peca::Jogador Peca::jogador() const
{
    return jogador da peca;
}
//---------------------------------------CLASSE
CELULA---------------------------------------

Celula::Celula(CorDeQuadrado const cor a usar,Peca::Tipo const tipo,
Peca::Jogador const jogador)
    : peca da celula(tipo,jogador),cor da celula(cor a usar)
{
//apóe contrução é necessário actuaçizar as medidas ->
actualizaMedidas();
}
bool Celula::cumpreInvariante() const
{
    return ((posicao no ecra.linha() >= 0) and
(posicao no ecra.coluna() >= 0)
            and (altura da celula >= 3) and largura da celula >= 7);
```

```
}
void Celula::simulaEliminaPeca()
{
    assert(cumpreInvariante());
    peca_da_celula.simulaPecaEliminada();
    assert(cumpreInvariante());
}
void Celula::paraSimulacao()
{
    assert(cumpreInvariante());
    peca_da_celula.paraSimulacao();
    assert(cumpreInvariante());
}
Peca Celula::peca() const
{
    assert(cumpreInvariante());
    return peca_da_celula;
}
void Celula::substituiPeca(Peca const& nova_peca)
{
    assert(cumpreInvariante());
    peca_da_celula = nova_peca;
    assert(cumpreInvariante());
}
void Celula::eliminaPeca()
{
    assert(cumpreInvariante());
    peca_da_celula = Peca(Peca::sem_peca,Peca::brancas);
    assert(cumpreInvariante());
}
bool Celula::estaOcupada() const
{
    assert(cumpreInvariante());
    return (peca_da_celula.tipo() != Peca::sem_peca);
}
void Celula::actualizaMedidas(int const linha,int const coluna,int const
altura, int const largura)
{
    assert((0 <= linha <= 7) and (0 <= coluna <= 7) and (altura >= 3)
and (largura >= 7));
    altura_da_celula = altura;
    largura_da_celula = largura;
    posicao_no_ecra = Posicao(2 + (linha * altura_da_celula), 2+ (coluna
* largura_da_celula));
    assert(cumpreInvariante());
}
void Celula::desenha() const
{
    assert(cumpreInvariante());

desenhaQuadrado(posicao_no_ecra,cor_da_celula,altura_da_celula,largura_d
a_celula);
}
void Celula::desenhaPeca() const
{
    assert(cumpreInvariante());
    Posicao posicao_da_peca = posicao_no_ecra + Dimensao(1,2);
    int altura = altura_da_celula - 2;
    int largura = largura_da_celula - 4;
```

```
    Peca::Jogador jogador da peca = peca da celula.jogador();
    Peca::Tipo tipo da peca = peca da celula.tipo();
    CorDeQuadrado cor da peca = peca da celula.corDestaPeca();
    if (tipo da peca != Peca::sem peca)
    {
        desenhaQuadrado(posicao_da_peca,cor_da_peca,altura,largura);
        if (tipo_da_peca == Peca::dama) {
            if (jogador da peca == Peca::brancas)
                ecra << cor damas brancas
<< cursor((posicao da peca.linha() + (altura / 2)),

(posicao_da_peca.coluna() + (largura / 2)))
                        << '*';
            if (jogador da peca == Peca::pretas)
                ecra << cor damas pretas
<< cursor((posicao da peca.linha() + (altura / 2)),

(posicao_da_peca.coluna() + (largura / 2)))
                        << '*';
        }
    }
}
void Celula::desenhaSeleccaoDestino() const
{
    assert(cumpreInvariante());
    Posicao posicao da seleccao = posicao no ecra +
Dimensao(altura da celula / 2,largura da celula / 2);
    desenhaQuadrado(posicao da seleccao,quadrado seleccao destino,1,1);
}
void Celula::desenhaSeleccaoOrigem() const
{
    assert(cumpreInvariante());

desenhaQuadrado(posicao no ecra,quadrado seleccao origem,altura da celul
a,largura da celula);
}
Peca::Tipo Celula::tipoDaPecaQueOcupaCelula() const
{
    assert(cumpreInvariante());
    return peca da celula.tipo();
}
Peca::Jogador Celula::jogadorDaPecaQueOcupaCelula() const
{
    assert(cumpreInvariante());
    return peca da celula.jogador();
}
//---------------------------------------CLASSE
TABULEIRO---------------------------------------

Tabuleiro::Tabuleiro()
    :estado da seleccao de origem(false),origem selecionada(Posicao(0,0)
),aspecto(0),ha nova dama(false)
{
    celulas do tabuleiro.resize(8);
    for(vector< vector<Celula> >::size type linha = 0; linha !=
celulas_do_tabuleiro.size(); ++linha)
    {
        for(int coluna  = 0; coluna != 8; ++coluna)
        {
```

```
            Peca::Tipo tipo peca da celula = Peca::peao;
            Peca::Jogador jogador peca da celula;
            bool valor e impar = ((linha + coluna) % 2) != 0;
            if ((linha <= 2) and valor e impar)
                jogador peca da celula = Peca::pretas;
            else if (((linha >= 5) and (linha <= 7)) and valor e impar)
                jogador peca da celula = Peca::brancas;
            else
                tipo peca da celula = Peca::sem peca;
            Celula celula a acrescentar((((linha + coluna) % 2) == 0) ?
quadrado ciano : quadrado azul
                                        ,
tipo peca da celula,jogador peca da celula );
            celulas do tabuleiro[linha].push back(celula a acrescentar);
        }
    }
    actualizaMedidas();
    assert(cumpreInvariante());
}
bool Tabuleiro::cumpreLimitesDePosicao(Posicao const
&posicao a verificar) const
{
    return (((posicao a verificar.linha() >= 0) and
(posicao a verificar.linha() <= 7)) and
            ((posicao a verificar.coluna() >= 0) and
(posicao a verificar.coluna() <= 7)));
}
bool Tabuleiro::cumpreInvariante() const
{
    if (celulas do tabuleiro.size() != 8)
        return false;
    for(vector< vector<Celula> >::size type linha = 0; linha !=
celulas do tabuleiro.size(); ++linha)
    {
        if(celulas do tabuleiro[linha].size() != 8)
            return false;
    }
    return (aspecto >= 0);
}
void Tabuleiro::desenha() const
{
    assert(cumpreInvariante());
    for(vector< vector<Celula> >::size type linha = 0; linha !=
celulas do tabuleiro.size(); ++linha)
    {
        for(vector< vector<Celula> >::size type coluna  = 0; coluna !=
celulas do tabuleiro[linha].size(); ++coluna)
            celulas do tabuleiro[linha][coluna].desenha();
    } //O cursor está agora no canto inferior direito do tabuleiro
    int altura = 3 + aspecto;
    int largura = 7 + aspecto;
    ecra << cursor(2+int(altura / 2),0); // vertical (numeros)
    for(int Linha = 1; Linha <= 8  ; ++Linha)
    {
        ecra << fundo << Linha;
        ecra << ecra.posicaoDoCursor() + Dimensao(altura,-1);
    }
    ecra << cursor(0,2+int(largura / 2)); // horizontal (letras)
    char ch= 'A';
```

```
    for(int  Coluna= 0; Coluna < 8  ; ++Coluna)
    {
        ecra << fundo << char(ch + Coluna);
        ecra << ecra.posicaoDoCursor() + Dimensao(0,largura - 1);
    }
    ecra << cursor(0,0) << fundo << caixa(1,1,8 * altura + 2, 8 *
largura + 2);//Desenha a borda da Tabuleiro
    if(estado da seleccao de origem) //desenha a seleccao de origem
        celulas do tabuleiro[origem selecionada.linha()]
[origem selecionada.coluna()].desenhaSeleccaoOrigem();
}
void Tabuleiro::actualizaMedidas()
{
    int tamanho vertical = ecra.dimensao().numeroDeLinhas();
    int tamanho horizontal = ecra.dimensao().numeroDeColunas();
    while((tamanho vertical < 27) or (tamanho horizontal < 82))
    {
        Aviso("Erro: Aumente a dimensao da janela para
continuar!").interage();
        tamanho vertical = ecra.dimensao().numeroDeLinhas();
        tamanho horizontal = ecra.dimensao().numeroDeColunas();
    }
    int incremento tamanho vertical = int((tamanho vertical - 27) / 8);
    int incremento tamanho horizontal = int((tamanho horizontal - 82) /
8);
    aspecto = ((incremento tamanho vertical >
incremento tamanho horizontal)?

incremento tamanho horizontal:incremento tamanho vertical);
    int largura da celula = 7 + aspecto;
    int altura da celula  = 3 + aspecto;
    for(vector<vector<Celula> >::size_type linha = 0; linha !=
celulas do tabuleiro.size(); ++linha)
    {
        for(vector< vector<Celula> >::size type coluna  = 0; coluna !=
celulas do tabuleiro[linha].size(); ++coluna)
        {
            celulas do tabuleiro[linha]
[coluna].actualizaMedidas(linha,coluna,altura da celula,largura da celul
a);
        }
    }

    ecra << apaga << refresca tudo;
    assert(cumpreInvariante());
}
void Tabuleiro::desenhaPecas() const
{
    assert(cumpreInvariante());
    for(vector< vector<Celula> >::size_type linha = 0; linha !=
celulas do tabuleiro.size(); ++linha)
    {
        for(vector< vector<Celula> >::size type coluna  = 0; coluna !=
celulas do tabuleiro[linha].size(); ++coluna)
        {
            celulas_do_tabuleiro[linha][coluna].desenhaPeca();
        }
    }
}
```

```
void Tabuleiro::SeleccaoDestino(Posicao const &posicao_do_tabuleiro)
const
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(posicao_do_tabuleiro));
    celulas_do_tabuleiro[posicao_do_tabuleiro.linha()]
[posicao_do_tabuleiro.coluna()].desenhaSeleccaoDestino();
}
void Tabuleiro::activaSeleccaoOrigem(Posicao const
&posicao_do_tabuleiro)
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(posicao_do_tabuleiro));
    estado_da_seleccao_de_origem = true;
    origem_selecionada = posicao_do_tabuleiro;
    desenha();
    desenhaPecas();
    assert(cumpreInvariante());
}
void Tabuleiro::desactivaSeleccaoOrigem()
{
    assert(cumpreInvariante());
    estado_da_seleccao_de_origem = false;
    desenha();
    desenhaPecas();
    assert(cumpreInvariante());
}
bool Tabuleiro::haSeleccaoDeOrigem() const
{
    assert(cumpreInvariante());
    return estado_da_seleccao_de_origem;
}
int Tabuleiro::valorDoAspecto() const
{
    assert(cumpreInvariante());
    return aspecto;
}
bool Tabuleiro::haNovaDama()
{
    bool valor_a_devolver = ha_nova_dama;
    ha_nova_dama = false ;
    return valor_a_devolver;
}
void Tabuleiro::terminaSimulacao()
{
    assert(cumpreInvariante());
    for(vector< vector<Celula> >::size_type linha = 0; linha !=
celulas_do_tabuleiro.size(); ++linha)
    {
        for(vector< vector<Celula> >::size_type coluna  = 0; coluna !=
celulas_do_tabuleiro[linha].size(); ++coluna)
        {
            celulas_do_tabuleiro[linha][coluna].paraSimulacao();
        }
    }
    assert(cumpreInvariante());
}
bool Tabuleiro::podeSeleccionarAlgumaPosicao(Peca::Jogador jogador)
const
```

```
{
    assert(cumpreInvariante());
    for(vector< vector<Celula> >::size_type linha = 0; linha !=
celulas_do_tabuleiro.size(); ++linha)
    {
        for(vector< vector<Celula> >::size_type coluna  = 0; coluna !=
celulas_do_tabuleiro[linha].size(); ++coluna)
        {
            if ((celulas_do_tabuleiro[linha]
[coluna].jogadorDaPecaQueOcupaCelula() == jogador) and
                (celulas_do_tabuleiro[linha]
[coluna].tipoDaPecaQueOcupaCelula() != Peca::sem_peca))
            {
                Tabuleiro::EstadoDePosicao estado =
podeSerOrigem(Posicao(linha,coluna),jogador,Peca::sem_peca);
                if ((estado == Tabuleiro::possivel) or (estado ==
Tabuleiro::pode_comer))
                    return true;
            }
        }
    }
    return false;
}
Peca::Tipo Tabuleiro::executaJogada(Posicao const &origem,Posicao const
&destino)
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino));
    Peca::Tipo peca_comida = Peca::sem_peca;
    Peca::Tipo tipo_da_nova_peca = celulas_do_tabuleiro[origem.linha()]
[origem.coluna()].tipoDaPecaQueOcupaCelula();;
    if (((destino.linha() == 0) or (destino.linha() == 7)) and
(tipo_da_nova_peca != Peca::dama))
    {
        tipo_da_nova_peca= Peca::dama;
        ha_nova_dama = true;
    }
    Peca::Jogador jogador_da_nova_peca =
celulas_do_tabuleiro[origem.linha()]
[origem.coluna()].jogadorDaPecaQueOcupaCelula();
    celulas_do_tabuleiro[destino.linha()]
[destino.coluna()].substituiPeca(Peca(tipo_da_nova_peca,jogador_da_nova_
peca));
    celulas_do_tabuleiro[origem.linha()][origem.coluna()].eliminaPeca();
    int distancia = abs(destino.linha() - origem.linha());
    if (distancia == 1)
        peca_comida = Peca::sem_peca;
    else
    {
        int linha = destino.linha() + (((destino.linha() -
origem.linha()) > 0) ? -1 : +1 );
        int coluna = destino.coluna() + (((destino.coluna() -
origem.coluna()) > 0) ? -1 : +1 );
        peca_comida = celulas_do_tabuleiro[linha]
[coluna].tipoDaPecaQueOcupaCelula();
        celulas_do_tabuleiro[linha][coluna].eliminaPeca();
    }
    assert(cumpreInvariante());
```

```
    return peca comida;
}
int Tabuleiro::numeroDePecasEntreDiagonal(Posicao const &origem,Posicao
const &destino) const
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino));
    int numero de pecas entre diagonal;
    if ( destino == origem )
        numero de pecas entre diagonal = 0;
    else
    {
        bool sentido vertical = ((destino.linha() - origem.linha()) >
0); // 1->para baixo
        bool sentido horizontal = ((destino.coluna() -
origem.coluna()) > 0); // 1->para a direita
        Posicao origem a calcular = origem +
Dimensao((sentido_vertical ? +1 : -1 ) , (sentido_horizontal ? +
1 : -1));
        if (celulas do tabuleiro[origem a calcular.linha()]
[origem a calcular.coluna()].estaOcupada())
            numero de pecas entre diagonal = 1 +
numeroDePecasEntreDiagonal(origem a calcular,destino);
        else
            numero de pecas entre diagonal = 0 +
numeroDePecasEntreDiagonal(origem a calcular,destino);
    }
    return numero de pecas entre diagonal;
}
Tabuleiro::EstadoDePosicao Tabuleiro::podeSerDestino(Posicao const
&origem,Posicao const &destino,
                                                    Peca::Jogador const
jogador actual,bool const e sequencia)
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino));
    Tabuleiro::EstadoDePosicao estado = Tabuleiro::impossivel;
    bool e diagonal = (abs((destino.linha() - origem.linha())) ==
abs((destino.coluna() - origem.coluna())));
    bool destino esta vazio = not(celulas do tabuleiro[destino.linha()]
[destino.coluna()].estaOcupada());
    if (e diagonal){
        if (destino esta vazio)
        {
            bool sentido correcto = ((destino.linha() - origem.linha() >
0) == int(jogador actual));
            int distancia = abs(destino.linha() - origem.linha());
            int numero_de_pecas_entre_diagonal =
numeroDePecasEntreDiagonal(origem,destino);
            if ((distancia == 1) and sentido correcto)
                estado = Tabuleiro::possivel;
            else
            {
                int sentido_vertical = destino.linha() +
(((destino.linha() - origem.linha()) > 0) ? -1 : +1 );
                int sentido horizontal = destino.coluna() +
(((destino.coluna() - origem.coluna()) > 0) ? -1 : +1 );
```

```
                Celula junto ao destino =
celulas do tabuleiro[sentido vertical][sentido horizontal];
                bool peca junto ao destino adversaria =
((junto ao destino.jogadorDaPecaQueOcupaCelula() != jogador actual)
                                                                and
(junto ao destino.tipoDaPecaQueOcupaCelula() != Peca::sem peca));
                Peca::Tipo tipo da peca origem =
celulas do tabuleiro[origem.linha()]
[origem.coluna()].tipoDaPecaQueOcupaCelula();
                if ((distancia == 2) and (numero de pecas entre diagonal
== 1) and //peaoao a comer
                    peca junto ao destino adversaria and
sentido correcto)
                    estado = Tabuleiro::pode comer;
                else if (tipo da peca origem == Peca::dama){ //damas
                    if ((distancia == 2) and
(numero de pecas entre diagonal == 1) and
                        peca junto ao destino adversaria and
e sequencia)
                        estado = Tabuleiro::pode comer; //komer kuando
sequencia
                    else if ((numero de pecas entre diagonal == 1) and
peca junto ao destino adversaria
                            and (tipo da peca origem == Peca::dama))
                        estado = Tabuleiro::pode comer; //komer normal
                    else if (numero de pecas entre diagonal == 0)
                        estado = Tabuleiro::possivel;
                }
            }
        }
        else
            estado = Tabuleiro::destino ocupado;
    }
    if(estado == Tabuleiro::impossivel)
        if (simulacaoDePercurso(origem,destino,jogador actual))
            estado = Tabuleiro::mais do que um movimento;
    assert(cumpreInvariante());
    return estado;
}
Tabuleiro::EstadoDePosicao Tabuleiro::podeSerOrigem(Posicao const
&posicao do tabuleiro,Peca::Jogador const jogador actual
                                                    ,Peca::Tipo const
tipo_da_peca) const
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(posicao do tabuleiro));
    Tabuleiro::EstadoDePosicao estado = Tabuleiro::impossivel;
    int linha = posicao do tabuleiro.linha();
    int coluna = posicao do tabuleiro.coluna();
    bool pode comer = false;
    bool possivel = false;
    Celula celula a verificar = celulas do tabuleiro[linha][coluna];
    Peca::Jogador jogador da peca = (tipo da peca == Peca::sem peca) ?
celula a verificar.jogadorDaPecaQueOcupaCelula() : jogador actual;
    //no caso do 3 parm ser explicitado, simula uma peça na origem
definida
    Peca::Tipo tipo de peca = (tipo da peca == Peca::sem peca) ?
celula a verificar.tipoDaPecaQueOcupaCelula() : tipo da peca;
    bool celula ocupada = (tipo da peca == Peca::sem peca) ?
```

```
celula a verificar.estaOcupada() : true;
    if(celula ocupada)
    {
        if (jogador actual != jogador da peca)
            estado = Tabuleiro::jogador diferente da seleccao;
        else
        {
            int numero de iteracoes = ((tipo de peca == Peca::dama)? 4 :
2);
            int iteracao = 1;
            while(iteracao < (numero de iteracoes + 1))
            {
                int sentido horizontal = int(pow( -1.0 , iteracao));
                int sentido vertical = ((jogador actual ==
Peca::brancas) ? -1 :1) * ((iteracao < 3) ? 1 : -1);
                int linha a calcular = linha + sentido vertical;
                int coluna a calcular = coluna + sentido horizontal;
                if ((coluna a calcular < 0) or (coluna a calcular > 7)
or
                    (linha a calcular < 0) or (linha a calcular > 7))
//limita o tabuleiro
                {
                    coluna a calcular = coluna;
                    linha a calcular = linha;
                }
                Celula proxima celula diagonal =
celulas do tabuleiro[linha a calcular][coluna a calcular];
                linha a calcular = linha + (2 * sentido vertical);
                coluna a calcular = coluna + (2 * sentido horizontal);
                if ((coluna a calcular < 0) or (coluna a calcular > 7)
or
                    (linha a calcular < 0) or (linha a calcular > 7))
                {
                    coluna a calcular = coluna;
                    linha a calcular = linha;
                }
                Celula celula captura =
celulas do tabuleiro[linha a calcular][coluna a calcular];
                Peca::Jogador jogador captura =
proxima celula diagonal.jogadorDaPecaQueOcupaCelula();
                if (not(proxima celula diagonal.estaOcupada()))
                    possivel = true;
                else if (proxima celula diagonal.estaOcupada() and
                        not(celula captura.estaOcupada()) and
(jogador actual != jogador captura))
                    pode comer = true;
                else
                    estado = Tabuleiro::impossivel;
                ++iteracao;
            }
        }
    }
    else
        estado = Tabuleiro::origem sem peca;
    if (pode comer)
        estado = Tabuleiro::pode_comer;
    else if (possivel)
        estado = Tabuleiro::possivel;
    return estado;
```

```
}
bool Tabuleiro::simulacaoDePercurso(Posicao const &origem,Posicao const
&destino,
                                    Peca::Jogador const jogador actual)
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino));
    Peca::Tipo tipo de peca = celulas do tabuleiro[origem.linha()]
[origem.coluna()].tipoDaPecaQueOcupaCelula();
    if (tipo de peca == Peca::dama)
    {
        int linha = origem.linha();
        int coluna = origem.coluna();
        int numero de iteracoes = ((tipo de peca == Peca::dama)? 4 : 2);
        int iteracao = 1;
        while(iteracao < (numero de iteracoes + 1))
        {
            int sentido horizontal = int(pow( -1.0 , iteracao));
            int sentido vertical = ((jogador actual ==
Peca::brancas) ? -1 :1) * ((iteracao < 3) ? 1 : -1);
            bool estado do ciclo = true;
            int contador = 1;
            do
            {
                int linha actual = linha + (sentido vertical *
contador);
                int coluna actual = coluna + (sentido horizontal *
contador);
                if (not(((coluna actual < 0) or (coluna actual > 7) or
                         (linha actual < 0) or (linha actual > 7))))
                {
                    Celula cell a verificar =
celulas do tabuleiro[linha actual][coluna actual];
                    bool casa ocupada = cell a verificar.estaOcupada();
                    bool jogador compativel =
(cell a verificar.jogadorDaPecaQueOcupaCelula() != jogador actual);
                    if (casa ocupada and jogador compativel)
                    {
                        int linha destino = linha actual +
sentido vertical;
                        int coluna destino = coluna actual +
sentido horizontal;
                        if (not(((coluna destino < 0) or
(coluna destino > 7) or
                                 (linha destino < 0) or (linha destino >
7))))
                        {
                            if(not(celulas do tabuleiro[linha destino]
[coluna destino].estaOcupada())) //destino vazio
                            {
                                Posicao
junto a primeira adversaria(linha destino,coluna destino);
                                bool resultado =
podePercorrer(junto a primeira adversaria,destino,jogador actual,tipo de
peca);
                                terminaSimulacao();
                                return resultado;
                            }
```

```
                                estado do ciclo = false;
                            }
                        }
                        else if(casa ocupada)
                            estado do ciclo = false;
                    }
                    else
                        estado do ciclo = false;
                    ++contador;
                }
                while(estado do ciclo);
                    ++iteracao;

            }
            return false;
        }
        else
        {
            bool resultado =
podePercorrer(origem,destino,jogador actual,tipo de peca);
            terminaSimulacao();
            return resultado;
        }
}
bool Tabuleiro::podePercorrer (Posicao const &origem,Posicao const
&destino,
                               Peca::Jogador const
jogador actual,Peca::Tipo tipo de peca )
{
    assert(cumpreInvariante());
    assert(cumpreLimitesDePosicao(origem) and
cumpreLimitesDePosicao(destino));
    if (destino == origem)
        return true;
    if (podeSerOrigem(origem,jogador actual,tipo de peca)==
Tabuleiro::pode comer)
    {
        int linha = origem.linha();
        int coluna = origem.coluna();
        int numero de iteracoes = ((tipo de peca == Peca::dama)? 4 : 2);
        int iteracao = 1;
        while(iteracao < (numero de iteracoes + 1))
        {
            int sentido_horizontal = int(pow( -1.0 , iteracao));
            int sentido vertical = ((jogador actual ==
Peca::brancas) ? -1 :1) * ((iteracao < 3) ? 1 : -1);
            int linha a calcular = linha + (2 * sentido vertical);
            int coluna a calcular = coluna + (2 * sentido horizontal);
            Posicao
destino_a_calcular(linha_a_calcular,coluna_a_calcular);

            if (not(((coluna a calcular < 0) or (coluna a calcular > 7)
or
                     (linha a calcular < 0) or (linha a calcular > 7))))
            {
                int peca_intermedia_linha = linha + sentido_vertical;
                int peca_intermedia_coluna = coluna +
sentido horizontal;
                if (not(((peca intermedia coluna < 0) or
```

```
(peca_intermedia_coluna > 7) or
                          (peca_intermedia_linha < 0) or
(peca_intermedia_linha > 7))))
                {
                    bool destino_vazio =
                        (celulas_do_tabuleiro[linha_a_calcular]
[coluna_a_calcular].tipoDaPecaQueOcupaCelula() == Peca::sem_peca );
                    Peca::Jogador junto_ao_destino =
                        celulas_do_tabuleiro[peca_intermedia_linha]
[peca_intermedia_coluna].jogadorDaPecaQueOcupaCelula();
                    bool peca_junto_ao_destino_adversaria =
(junto_ao_destino != jogador_actual);
                    bool peca_junto_ao_destino =
                        (celulas_do_tabuleiro[peca_intermedia_linha]
[peca_intermedia_coluna].tipoDaPecaQueOcupaCelula() != Peca::sem_peca);

                    if (peca_junto_ao_destino and
peca_junto_ao_destino_adversaria and destino_vazio)
                    {
                        if ((linha_a_calcular == 0) or (linha_a_calcular
== 7))
                            tipo_de_peca = Peca::dama;
                        celulas_do_tabuleiro[peca_intermedia_linha]
[peca_intermedia_coluna].simulaEliminaPeca();

if(podePercorrer(destino_a_calcular,destino,jogador_actual,tipo_de_peca)
)
                           return true;
                        celulas_do_tabuleiro[peca_intermedia_linha]
[peca_intermedia_linha].paraSimulacao();
                    }
                }
            }
            ++iteracao;
        }
    }
    return false;
}
void Tabuleiro::processaErroDeEstadoDePosicao(Tabuleiro::EstadoDePosicao
const estado) const
{
    assert(cumpreInvariante());
    string mensagem_a_mostrar;
    switch(estado)
    {
      case Tabuleiro::origem_sem_peca :
        mensagem_a_mostrar = "A posicao escolhida não està ocupada!";
        break;
      case Tabuleiro::destino_ocupado :
        mensagem_a_mostrar = "Não é possível mover a peças escolhida!";
        break;
      case Tabuleiro::jogador_diferente_da_seleccao:
        mensagem_a_mostrar = "A peca não tem a cor certa!";
        break;
      case Tabuleiro::impossivel :
        mensagem_a_mostrar = "Não é possível mover a peça escolhida!";
        break;
      case Tabuleiro::mais_do_que_um_movimento:
        mensagem_a_mostrar = "Jogada realizável em mais que um
```

```
movimento";
         break;
       default:
         break;
     }
     Aviso(mensagem_a_mostrar).interage();
     ecra.apaga();
     desenha();
     desenhaPecas();
}
//----------------------------------------
JOGO-----------------------------------------

Jogo::Jogo()
     :jogador_actual(Peca::brancas),e_sequencia(false),peoes_brancas(12),
damas_brancas(0)
      ,peoes_pretas(12),damas_pretas(0),contador_de_jogadas(0)
{
     assert(cumpreInvariante());
     srand(getpid());
     int meu_numero;
     int numero_dele;
     do
     {
         meu_numero = rand();
         mensageiro.envia(meu_numero);
         mensageiro.leMensagem();
         numero_dele = mensageiro.mensagemLida<int>();
     }while(meu_numero == numero_dele);
     if (meu_numero > numero_dele)
     {
         meu_jogador = Peca::brancas;
         Aviso("Jogas com as Brancas").interage();
     }
     else
     {
         meu_jogador = Peca::pretas;
         Aviso("Jogas com as Pretas").interage();
     }
     assert(cumpreInvariante());
}

bool Jogo::cumpreInvariante() const
{
     return (((0 <= peoes_brancas <= 12) and (0 <= peoes_pretas < 8))
             and ((0 <= damas_brancas <= 12) and (0 <= damas_pretas < 8))
             and (0 <= contador_de_jogadas <= 20) and
             ((0 <= origem.linha() <= 7 ) and (0 <= origem.coluna() <=
7)) and
             ((0 <= destino.linha() <= 7 ) and (0 <= destino.coluna() <=
7)) );

}
bool Jogo::jogadorActualGanhou() const
{
     assert(cumpreInvariante());
     return (((peoes_brancas == 0) and (damas_brancas == 0))
             or ((peoes_pretas == 0) and (damas_pretas ==0)));
}
```

```
void Jogo::verificaNovasDamas()
{
    assert(cumpreInvariante());
    if (tabuleiro_de_jogo.haNovaDama())
        calculaNovasInformacoesDeJogo(Peca::dama,false);
    assert(cumpreInvariante());
}
void Jogo::executaJogadasDaPilha()
{
    assert(cumpreInvariante());
    Posicao
origem (pilha_de_posicoes[0].linha(),pilha_de_posicoes[0].coluna());
    for(vector<Posicao>::size_type posicao_actual  = 1; posicao_actual !
= pilha_de_posicoes.size(); ++posicao_actual)
    {
        Posicao
destino (pilha_de_posicoes[posicao_actual].linha(),pilha_de_posicoes[pos
icao_actual].coluna());
        Peca::Tipo peca_tomada =
tabuleiro_de_jogo.executaJogada(origem ,destino );
        calculaNovasInformacoesDeJogo(peca_tomada,true);
        verificaNovasDamas();
        origem  = destino ;
    }
    assert(cumpreInvariante());
}
void Jogo::despachaPilha()
{
    assert(cumpreInvariante());
    int numero_de_posicoes = pilha_de_posicoes.size();
    mensageiro.envia(numero_de_posicoes);
    for(vector<Posicao>::size_type posicao_actual  = 0; posicao_actual !
= pilha_de_posicoes.size(); ++posicao_actual)
    {
        int linha = pilha_de_posicoes[posicao_actual].linha();
        int coluna = pilha_de_posicoes[posicao_actual].coluna();
        mensageiro.envia(linha);
        mensageiro.envia(coluna);
    }
    assert(cumpreInvariante());
}
Jogo::EstadoDeJogo Jogo::motorDeJogo()
{
    assert(cumpreInvariante());
    tabuleiro_de_jogo.desenha();
    tabuleiro_de_jogo.desenhaPecas();
    mostraInformacoes();
    ecra.refresca();
      do{ //ciclo referente ás 20 jogadas
        ++contador_de_jogadas;
        for(int jogador = 0; jogador!=2; ++jogador)
        { // ciclo referente a troca de jogadores
            pilha_de_posicoes.clear(); //vector volta a Zero
          if
(not(tabuleiro_de_jogo.podeSeleccionarAlgumaPosicao(jogador_actual)))
//testa empate
                return (jogador_actual == Peca::brancas) ?
Jogo::pretas_ganharam : Jogo::brancas_ganharam;
          bool jogada_terminada = false;
```

```
            if (jogador actual == meu jogador)
            {
                //
                while (not(jogada terminada)){
                    if (not(seleccaoInteractiva())) //testa desistencia
                        return (jogador actual == Peca::brancas) ?
Jogo::brancas desistiram : Jogo::pretas desistiram ;
                    Peca::Tipo peca tomada =
tabuleiro de jogo.executaJogada(origem,destino);
                    calculaNovasInformacoesDeJogo(peca tomada,true);
                    verificaNovasDamas();
                    if (jogadorActualGanhou()) //testa vitoria
                    {
                        despachaPilha();
                        return (jogador actual == Peca::brancas) ?
Jogo::brancas ganharam : Jogo::pretas ganharam;
                    }
                    if
((tabuleiro de jogo.podeSerOrigem(destino,jogador actual,Peca::sem peca)
                         == Tabuleiro::pode comer)
                        and peca tomada != Peca::sem peca)
                    {
                        MenuDeSimOuNao menu de sequencia("Quer
continuar?");
                        menu de sequencia.interage();
                        if (menu de sequencia.opcaoActual())
                        {

tabuleiro de jogo.activaSeleccaoOrigem(destino);
                            origem = destino;
                            e_sequencia = true;
                        }
                        else
                            jogada terminada = true;
                    }
                    else
                        jogada_terminada = true;
                }
                despachaPilha(); //envia as posicoes para o adeversario
            }
            else //WEEee Não sou eu a jogar!!
            {
                mensageiro.leMensagem();
                int numero de posicoes = mensageiro.mensagemLida<int>();
               for(int posicao actual = 0; posicao actual !=
numero de posicoes; ++posicao actual)
                {
                    mensageiro.leMensagem();
                    int linha = mensageiro.mensagemLida<int>();
                    mensageiro.leMensagem();
                    int coluna = mensageiro.mensagemLida<int>();
                    pilha de posicoes.push back(Posicao(linha,coluna));
                }
                executaJogadasDaPilha();
                if (jogadorActualGanhou()) //testa vitoria
                    return (jogador_actual == Peca::brancas) ?
Jogo::brancas ganharam : Jogo::pretas ganharam;
            }
```

```
            jogador actual = (jogador actual == Peca::brancas) ?
Peca::pretas : Peca::brancas; // muda de jogador
            e sequencia = false;
            tabuleiro de jogo.desactivaSeleccaoOrigem();
            tabuleiro de jogo.desenha();
            tabuleiro de jogo.desenhaPecas();
            mostraInformacoes();
            ecra.refresca();

        }
      }while(contador de jogadas != 20);
      return Jogo::empate limite jogadas;
}
void Jogo::calculaNovasInformacoesDeJogo(Peca::Tipo const
tipo peca tomada, bool const peca tomada)
{
    assert(cumpreInvariante());
    if (not(peca tomada))
    {
        if (jogador actual == Peca::brancas)
        {
            ++damas brancas;
            --peoes brancas;
        }
        else
        {
            ++damas pretas;
            --peoes pretas;
        }
    }
    else if(tipo peca tomada != Peca::sem peca)
    {
         contador de jogadas = 0;
        if (jogador actual != Peca::brancas)
        {
            if (tipo peca tomada == Peca::peao)
                --peoes brancas;
            else
                --damas brancas;
        }
        else
        {
            if (tipo peca tomada == Peca::peao)
                --peoes pretas;
            else
                --damas pretas;
        }

    }
    assert(cumpreInvariante());
}
void Jogo::mostraInformacoes() const
{
    assert(cumpreInvariante());
    int largura da celula = 7 + tabuleiro de jogo.valorDoAspecto();
    int altura da celula  = 3 + tabuleiro de jogo.valorDoAspecto();
    ecra << cursor(2,2 + 8*largura da celula + 3) << largura(20)
<< cor texto brancas << a esquerda << "Brancas";
    ecra << cursor(3,2 + 8*largura da celula + 3) << cor texto
```

```
<< "Peões: " << largura(2) << a direita << peoes brancas;
    ecra << cursor(4,2 + 8*largura da celula + 3) << cor texto
<< "Damas: " << largura(2) << a direita << damas brancas;

    ecra << cursor(6,2 + 8*largura da celula + 3) << largura(20)
<< cor texto pretas << a esquerda << "Pretas";
    ecra << cursor(7,2 + 8*largura da celula + 3) << cor texto
<< "Peões: " << largura(2) << a direita << peoes pretas;
    ecra << cursor(8,2 + 8*largura da celula + 3) << cor texto
<< "Damas: " << largura(2) << a direita << damas pretas;

    Posicao posicao indica jogador(8 * altura da celula -1,2 + 8 *
largura da celula + 3);
    desenhaQuadrado(posicao indica jogador,(jogador actual ==
Peca::pretas ? quadrado azul : quadrado ciano),3,20);
    ecra << cursor(posicao indica jogador.linha()+
1,posicao indica jogador.coluna())
         <<(jogador actual ? cor texto pretas : cor texto brancas)
<< largura(20) << ao centro
         <<(jogador actual ? "Jogam as Pretas" : "Jogam as brancas");
}
Tecla Jogo::selecciona(Posicao &posicao)
{
    assert(cumpreInvariante());
    assert((0 <= posicao.linha() < 8) and (0 <= posicao.coluna() < 8));
    tabuleiro de jogo.SeleccaoDestino(posicao);
    while (true)
     {
        if(ecra.foiRedimensionado())//em caso de redimensionamento
        {
            tabuleiro de jogo.actualizaMedidas();
            ecra.refresca();
            tabuleiro de jogo.desenha();
            tabuleiro de jogo.desenhaPecas();
            tabuleiro de jogo.SeleccaoDestino(posicao);
            mostraInformacoes();
        }
        if (teclado.haTeclaDisponivel(10))
        {
            teclado.leProximaTeclaDisponivel();
            Tecla tecla primida = teclado.teclaLida();
            if (tecla primida.eDeDeslocamento()) //quando é premido
deslocamento
            {
                switch(tecla primida)//delimita a mesa
                {
                  case Tecla::cima :
                    if(posicao.linha() > 0)
                        posicao.mudaLinhaPara(posicao.linha() - 1);
                    break;
                  case Tecla::baixo :
                    if(posicao.linha() < 7)
                        posicao.mudaLinhaPara(posicao.linha() + 1);
                    break;
                  case Tecla::esquerda :
                    if(posicao.coluna() > 0)
                        posicao.mudaColunaPara(posicao.coluna() - 1);
                    break;
                  case Tecla::direita :
```

```
                    if(posicao.coluna() < 7)
                        posicao.mudaColunaPara(posicao.coluna() + 1);
                    break;
                  default:
                    break;
                }
                tabuleiro_de_jogo.desenha();
                tabuleiro_de_jogo.desenhaPecas();
                tabuleiro_de_jogo.SeleccaoDestino(posicao);
            }
            else if ((tecla primida == Tecla::entrada) or (tecla primida
== Tecla::F4))// seleciona origem e destino
                return tecla_primida;
        }
        ecra.refresca();
    }
    assert(cumpreInvariante());
}
bool Jogo::seleccaoInteractiva()
{
    assert(cumpreInvariante());
    Tecla tecla primida = Tecla::F12;
    if (not(tabuleiro de jogo.haSeleccaoDeOrigem())) //origem
    {
        origem = destino;
        bool fim do ciclo = false;
        while(not(fim do ciclo))//validação de escolha
        {
            tecla primida = selecciona(origem);
            if (tecla primida == Tecla::F4)
                return false;
            Tabuleiro::EstadoDePosicao estado da seleccao =
tabuleiro de jogo.podeSerOrigem(origem,jogador actual,Peca::sem peca);
            fim do ciclo = ((estado da seleccao == Tabuleiro::possivel)
or (estado da seleccao == Tabuleiro::pode comer));
            if (not(fim do ciclo))
            {

tabuleiro de jogo.processaErroDeEstadoDePosicao(estado da seleccao);
                mostraInformacoes();
            }
        }
        tabuleiro_de_jogo.activaSeleccaoOrigem(origem);
        pilha_de_posicoes.push_back(origem); // adiciona a origem á
pilha IPC
        destino = origem;
    }
    bool fim do ciclo = false; //destino
    while(not(fim do ciclo))//validação de escolha
    {
        tecla primida = selecciona(destino);
        if (tecla primida == Tecla::F4)
            return false;
        Tabuleiro::EstadoDePosicao estado da seleccao =
tabuleiro de jogo.podeSerDestino(origem,destino,jogador actual,e sequenc
ia);
        if (e_sequencia)
        {
            if (estado da seleccao == Tabuleiro::pode comer)
```

```
                fim_do_ciclo = true;
            estado_da_seleccao = Tabuleiro::impossivel;
        }
        else
            fim_do_ciclo = ((estado_da_seleccao == Tabuleiro::possivel)
or (estado_da_seleccao == Tabuleiro::pode_comer));
        if (not(fim_do_ciclo))
        {

tabuleiro_de_jogo.processaErroDeEstadoDePosicao(estado_da_seleccao);
            mostraInformacoes();
        }
        tabuleiro_de_jogo.SeleccaoDestino(destino);
    }
    pilha_de_posicoes.push_back(destino); //adiciona + uma posicao a
pilha IPC
    return true;
}
void Jogo::processaRsultadoDeJogo(Jogo::EstadoDeJogo const
resultado_do_jogo) const
{
    assert(cumpreInvariante());
    string mensagem_a_mostrar;
    switch (resultado_do_jogo)
    {
      case Jogo::pretas_ganharam:
        mensagem_a_mostrar = "As Pretas Ganharam o Jogo";
        break;
      case Jogo::brancas_ganharam:
        mensagem_a_mostrar = "As Brancas Ganharam o Jogo";
        break;
      case Jogo::brancas_desistiram:
        mensagem_a_mostrar = "As Brancas desistiram do jogo";
        break;
      case Jogo::pretas_desistiram:
        mensagem_a_mostrar = "As Pretas desistiram do jogo";
        break;
      case Jogo::empate_limite_jogadas:
        mensagem_a_mostrar = "Empate, O limite de jogadas foi atingido";
        break;
      case Jogo::empate:
        mensagem_a_mostrar = "Os Jogadores empataram";
        break;
      default:
        break;
    }
    Aviso(mensagem_a_mostrar).interage();
}
int main ()
{
    //teclado.leProximaTeclaDisponivel(); //debug
    Jogo novo_jogo;
    Jogo::EstadoDeJogo resultado_do_jogo = novo_jogo.motorDeJogo();
    novo_jogo.processaRsultadoDeJogo(resultado_do_jogo);
}
//c++ -Wall -ansi -pedantic -g -I/usr/local/include/stlport
damas_entrega_final.C -lstlport_gcc_stldebug -lpthread -lefence -o
damas_entrega_final -lslang -lSlang++ -lUtilitarios -lIPC++
```