FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

MIEIC

Mestrado Integrado em Engenharia Informática e Computação

Projecto 2 - PROG

Programação

João dos Santos Rodrigues Soares dos Reis
Vasco Fernandes Gonçalves

Turma 1
Grupo 6

Sábado, 21 de Maio de 2011

Introdução:

Este trabalho foi realizado no âmbito da unidade curricular de Programação com o objectivo de aplicar os conhecimentos adquiridos ao longo do ano lectivo.

O projecto consiste na implementação de um jogo de Dominó *All-Fives*, cujas regras podem ser consultadas no seguinte *link*:

http://www.domino-games.com/domino-rules/allfives-rules.html

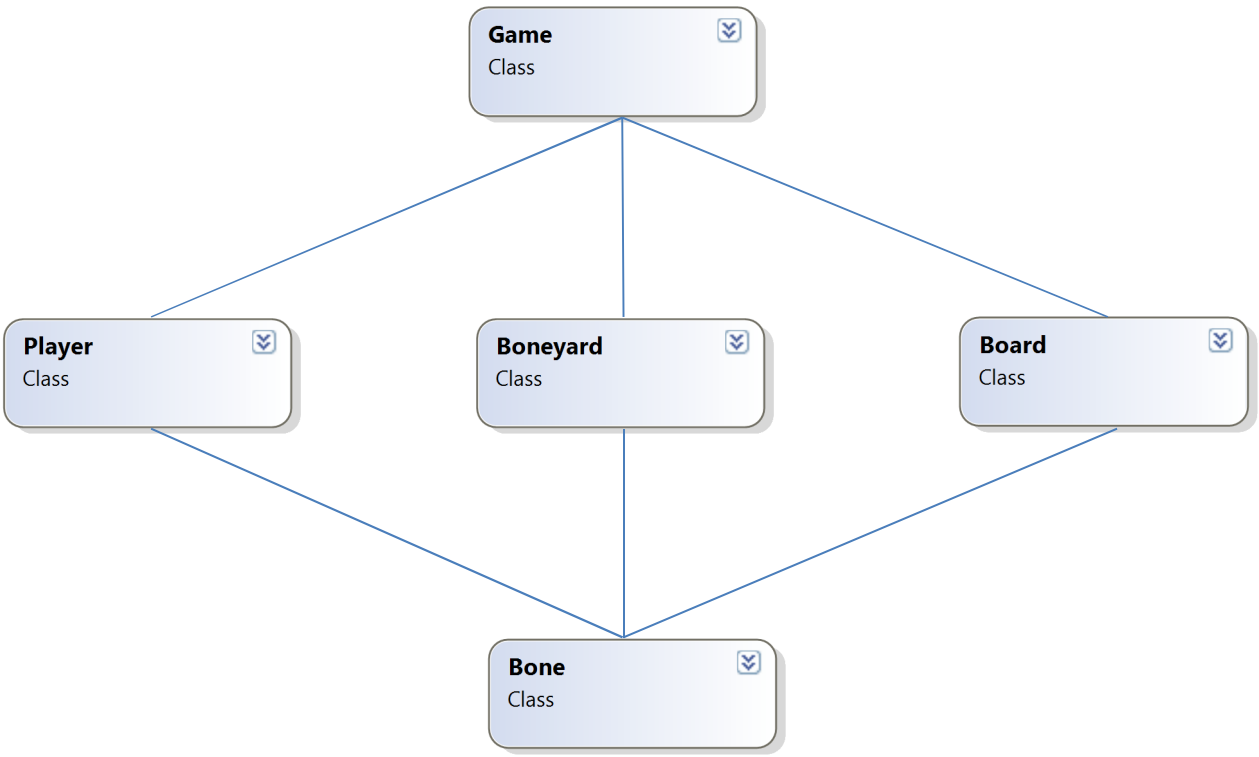
Temos como funcionalidades a hipótese de jogarem quatro jogadores, humanos ou controlados pelo computador.

Implementamos todas as funcionalidades solicitadas, e acrescentamos mais funções que podem ser vistas nas páginas seguintes.

Concepção e Implementação da Solução

Estrutura de Classes

Além das classes que foram sugeridas no enunciado (*Bone*, *Boneyard*, *Player*, *Board*), decidimos criar uma classe chamada *Game* que vai ser responsável por todas as interacções entre as diferentes classes.

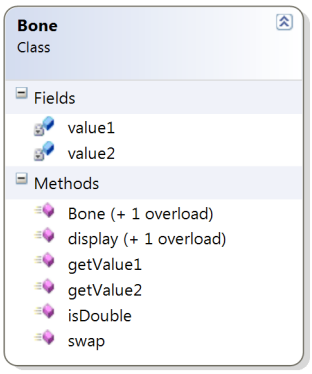


Implementação das Classes

Em relação ao diagrama de classes que já foi entregue anteriormente, o nosso projecto final tem algumas alterações.

Uma delas foi a remoção da classe *AI* que seria uma classe *derivada* da classe *Player*. A classe *AI* iria ser a classe do jogador automático, porém, constatámos, durante a implementação das classes, que seria muito mais eficiente e simples diferenciar um jogador humano de um jogador automático através do atributo *boolAI* que será *true* quando determinado objecto da classe *Player* for um jogador automático e *false* quando não o for. Assim um jogador automático e um jogador humano são objectos que pertencem à mesma classe, a classe *Player*.

Bone (classe da peça de dominó)



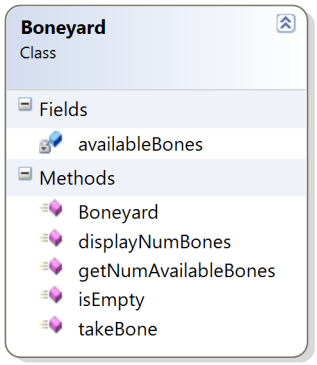
Para esta classe, desenvolvemos dois construtores. O construtor por defeito cria uma peça com os valores 7|7. O outro construtor cria uma peça com os valores a|b (sendo a e b inteiros passados como argumentos ao construtor).

Criámos dois métodos para visualizar as peças no ecrã (*Bone::display(int x, int y)* e *Bone::display(int &x, int &y, char row)*). Aqui, *x* e *y* são coordenadas utilizadas pela função *gotoxy(int x, int y)*. O método sem o argumento *row* é usado para visualizar as peças da mão de um jogador. O método com o argumento *row* é usado para visualizar as peças do tabuleiro (*board*). Este último método modifica os valores de *x* e *y* para as coordenadas ficarem a apontar para a próxima posição do tabuleiro disponível para ser utilizado por uma peça.

Implementámos dois métodos *get*, sendo que cada um devolve o valor respectivo da peça, e também um método *Bone::isDouble()* que devolve *true* se a peça for *double* e *false* se não.

Por fim, criámos um método *Bone::swap()* que troca os atributos: *value1* passa a ser *value2* e vice-versa. Este método é útil para o algoritmo usado nos métodos de visualização da classe *Board*, os quais são descritos mais à frente.

Boneyard (classe do monte de Bones)



O atributo *availableBones* é um *vector<Bone>* que contém as peças que estão no monte.

O único construtor para esta classe, *Boneyard::Boneyard()* cria objectos com todas as 28 peças do jogo.

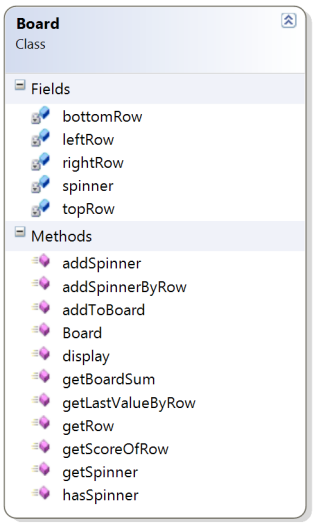
O método *Boneyard::getNumAvailableBones()* devolve o número de peças que estão no monte.

O método *Boneyard::isEmpty()* devolve *true* se o monte está vazio ou *false* se não.

O método *Boneyard::takeBone()* devolve uma peça que é retirada do monte aleatoriamente.

O método *Boneyard::displayNumBones()* visualiza no ecrã o número de peças disponíveis no monte numa posição fixa da interface gráfica do jogador.

Board (classe do tabuleiro)



O atributo *spinner* é a primeira peça *double* jogada que fica centrada no ecrã.

Os restantes quatro atributos são do tipo *vector<Bone>* que contêm as peças jogadas na respectiva fila.

O construtor desta classe é um construtor vazio.

O método *Board::addSpinner(Bone)* atribui a *spinner* a peça passada por argumento.

Nota: enquanto não há *spinner* as peças são armazenadas no *vector<Bone> rightRow*, ordenados da esquerda para a direita.

O método *Board::addSpinnerByRow(Bone, char)* é usado quando já existem peças jogadas no momento em que é jogado o primeiro *double* pelo que esta peça pode ser jogada em duas direcções diferentes. O argumento do tipo *char* contém a letra (N/S/E/W) que representa a direcção onde a peça está a ser jogada. Se o jogador quiser jogar o *spinner* no lado direito, as peças armazenadas em *rightRow* são movidas para *leftRow*. Se o jogador jogar o *spinner* no lado esquerdo, as peças mantêm-se armazenadas em *rightRow*.

O método *Board::addToBoard(Bone, char)* é usado para adicionar uma peça sem ser o primeiro *double* ao tabuleiro.

O método *Board::display()* é o método que faz visualizar no ecrã todo o tabuleiro.

O método *Board::getBoardSum()* devolve a soma dos valores extremos de cada fila do tabuleiro.

O método *Board::getLastValueByRow(char)* devolve o valor extremo da fila representada pelo *char* passado por argumento.

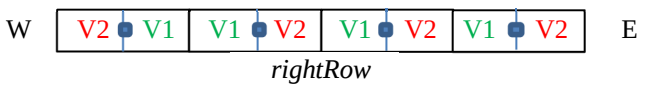
O método *Board::getRow(char)* devolve o *vector<Bone>* da fila representada pelo *char* passado por argumento.

O método *Board::getScoreOfRow(char)* devolve a pontuação da fila representada pelo *char* passado por agumento.

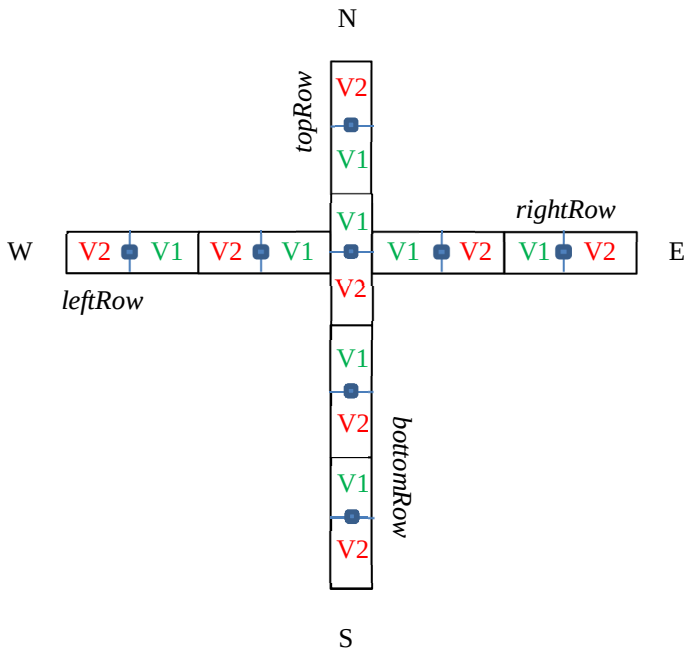
O método Board::getSpinner() devolve o *spinner*.

O método Board::hasSpinner() devolve *true* se já existe um *spinner* no tabuleiro e *false* se o objecto associado a *spinner* ainda é resultado do construtor por defeito da classe *Bone*.

O algoritmo que nós usámos para a visualização do tabuleiro relaciona os atributos de um objecto da classe *Bone* (*value1* e *value2*) com a orientação da peça no tabuleiro.



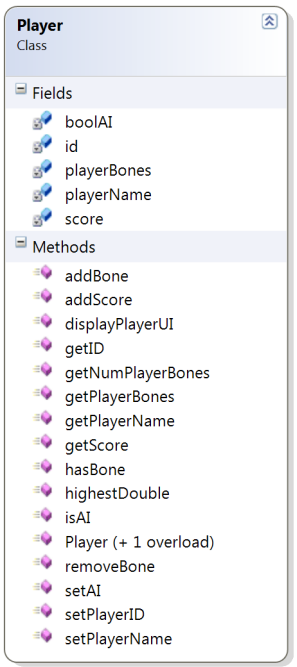
A ilustração anterior mostra os atributos de cada *Bone* numa situação de jogo sem *spinner* (V2 – *value2*, V1 – *value1*).



A ilustração anterior mostra os atributos de cada *Bone* numa situação de jogo com *spinner* (V2 – *value2*, V1 – *value1*).

Como está demonstrado na ilustração, o nosso algoritmo usa o método *Bone::swap()* para, ao adicionar peças ao tabuleiro, estabelecer *value1* como a parte da peça virada para o centro e *value2* a outra parte da peça. Assim torna-se simples testar se uma peça pode ser adicionada ao tabuleiro e recolher a sua pontuação.

Player (classe do jogador)



O atributo *boolAI* é *true* quando o objecto é um jogador automático e *false* quando o objecto é um jogador humano.

O atributo *id* é um inteiro que representa um número de identificação do jogador.

O atributo *vector<Bone> playerBones* armazena as peças do jogador.

O atributo *playerName* é uma *string* que contém o nome do jogador.

O atributo *score* é um inteiro que representa a pontuação do jogador.

O construtor por defeito desta classe inicializa *boolAI* como *false*, *id* a 0, *playerName* como uma *string* vazia e *score* como 0.

Ao outro construtor, é passado por argumento uma *string* que contém o nome a ser atribuído ao jogador e inicializa o resto das variáveis como o construtor por defeito.

O método *Player::addBone(Bone)* adiciona uma peça à mão do jogador.

O método *Player::addScore(int)* soma o valor passado como argumento à pontuação do jogador.

O método Player::displayPlayerUI() visualiza no ecrã a interface gráfica do jogador.

O método Player::getID() devolve o atributo *id*.

O método Player::getNumPlayerBones() devolve o número de peças que o jogador tem na mão.

O método Player::getPlayerBones() devolve o atributo *vector<Bone> playerBones*.

O método Player::getScore() devolve a pontuação do jogador.

O método Player::hasBone() devolve *true* se o jogador tem alguma peça na mão e *false* se não tiver nenhuma peça.

O método Player::highestDouble() devolve o indíce do maior *double* na mão do jogador ou -1 se não tiver nenhum *double*.

O método Player::isAI() devolve o atributo *boolAI*.

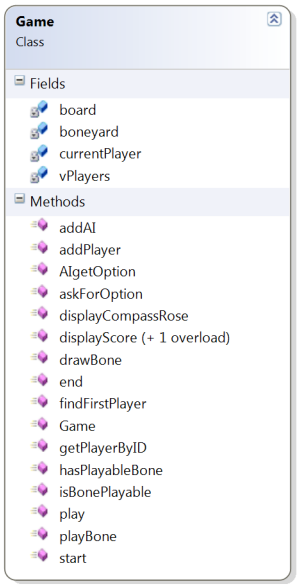
O método Player::removeBone(Bone) remove a peça passada por argumento da mão do jogador.

O método Player::setAI() modifica *boolAI* para *true*.

O método Player::setPlayerID(int) modifica o *id* do jogador para o valor passado por argumento.

O método Player::setPlayerName(string) atribui a *playerName* a *string* passada por argumento.

Game (classe do jogo)



O atributo *board* representa o tabuleiro do jogo.

O atributo *boneyard* representa o monte de peças do jogo.

O atributo *currentPlayer* é um *unsigned int* que consiste no *id* do próximo jogador a jogar.

O atributo *vector<Player> vPlayers* armazena os jogadores.

O único construtor desta classe é o construtor por defeito que cria um *board* e um *boneyard*.

O método *Game::addAI(Player)* adiciona um jogador automático ao jogo e atribui um *id* a esse jogador.

O método Game::addPlayer(Player) adiciona um jogador humano ao jogo e atribui um *id* a esse jogador.

O método Game::AIgetOption(unsigned int, int&, char&) atribui a *int&* a opção do jogador automático naquela jogada (opção é um inteiro que representa um *Bone* da mão do jogador e consiste no resultado da soma do índice do *Bone* e 1)

e a *char&* a respectiva direcção onde vai ser jogada a peça. Este valor que é devolvido é resultado de um algoritmo que selecciona a opção que permite ao jogador automático ganhar mais pontuação naquela jogada. Este algoritmo selecciona a opção do *double* maior (ou do *double* 5-5 mesmo que o jogador tenha na mão o 6-6, visto que o *double* 5-5 permite ao jogador ganhar 10 pontos) que estiver na mão do jogador, caso o tabuleiro ainda esteja vazio.

O método Game::askForOption(unsigned int, int&, char&) pergunta ao jogador humano que *Bone* quer jogar e em que direcção e atribui a opção e a direcção a *int&* e *char&* respectivamente.

Os métodos Game::displayCompassRose() e Game::displayScore() visualizam a rosa-dos-ventos e a pontuação de cada jogador, respectivamente.

O método Game::displayScore(unsigned int) tem a mesma função que Game::displayScore() mas escreve ao lado da pontuação do jogador que ganhou uma mensagem adicional. Este método só é chamado no final do jogo.

O método Game::drawBone(unsigned int) efectua a operação de retirar uma peça do monte e adicioná-la à mão do jogador que tem como *id* o valor passado por argumento.

O método Game::findFirstPlayer() devolve o *id* do jogador que deve jogar primeiro, segundo as regras do jogo (o jogador que tiver o *double* mais alto ou, no caso de nenhum jogador tiver *doubles*, o jogador que tiver a peça de maior valor.

O método Game::getPlayerByID(unsigned int) devolve um apontador para o jogador que tem como *id* o valor passado por argumento.

O método Game::hasPlayableBone(unsigned int) devolve *true* se o jogador com *id* igual ao valor passado por argumento possui alguma peça jogável e *false* se não.

O método Game::isBonePlayable(Bone) devolve *true* se a peça for jogável nas condições em que se encontra o tabuleiro e *false* se não.

O método Game::playBone(unsigned int, Bone, char) faz com que o jogador com *id* igual ao valor passado por argumento, tente jogar a peça passada por argumento no tabuleiro na direcção passada por argumento e devolve *true* se for bem sucedida a jogada ou *false* se a jogada não for permitida.

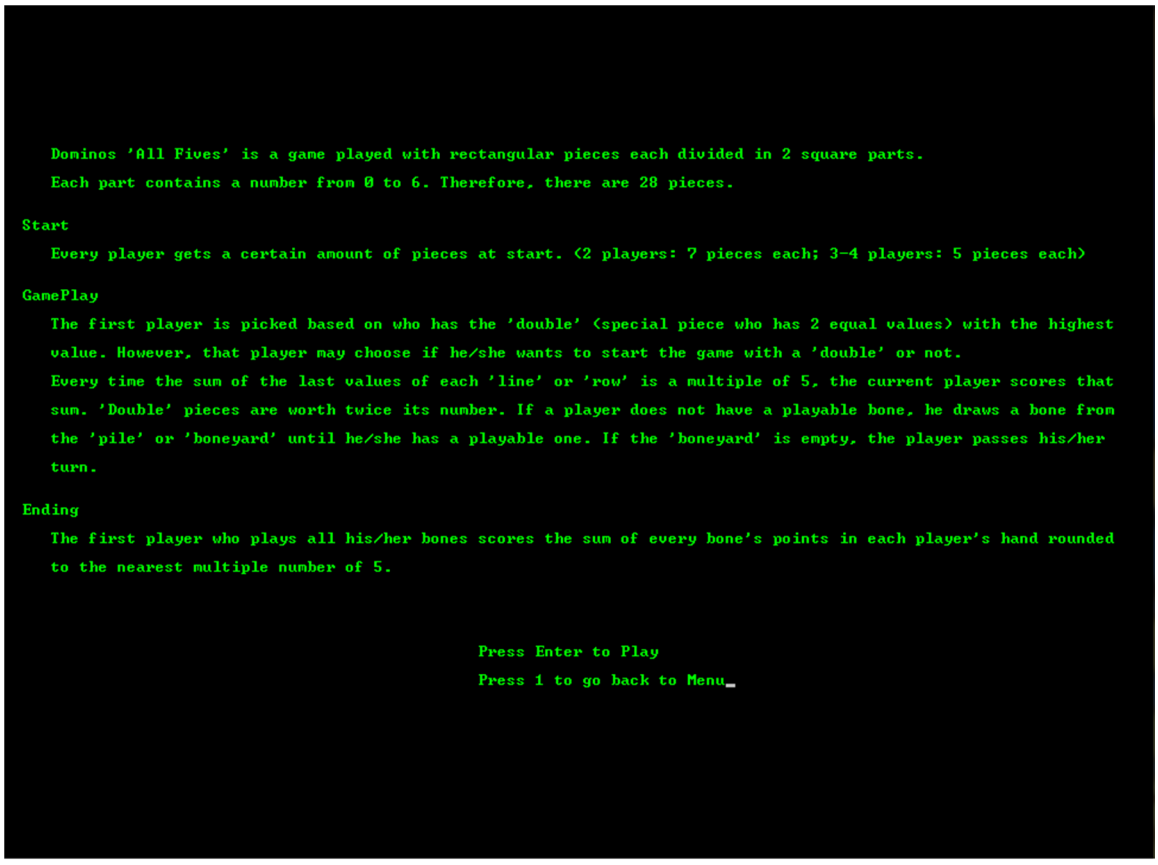
O método Game::start(vector<Player>,vector<Player>) inicializa as variáveis do jogo que vai ter como jogadores humanos os elementos do primeiro *vector* passado por argumento e jogadores automáticos o segundo *vector* passado por argumento.

O método Game::play() inicia o jogo.

O método Game::end(unsigned int, bool) termina o jogo, faz os cálculos necessários para se saber quem é o vencedor e visualiza uma mensagem a mostrar quem venceu. O primeiro argumento é o *id* do jogador que ficou sem peças para jogar. No caso do jogo ter bloqueado, ou seja, todos os jogadores ficaram sem jogadas possíveis, embora todos tenham peças na mão, o primeiro argumento é o *id* do jogador que tem o conjunto de peças na mão menos valioso.

Anexo (Screenshots):

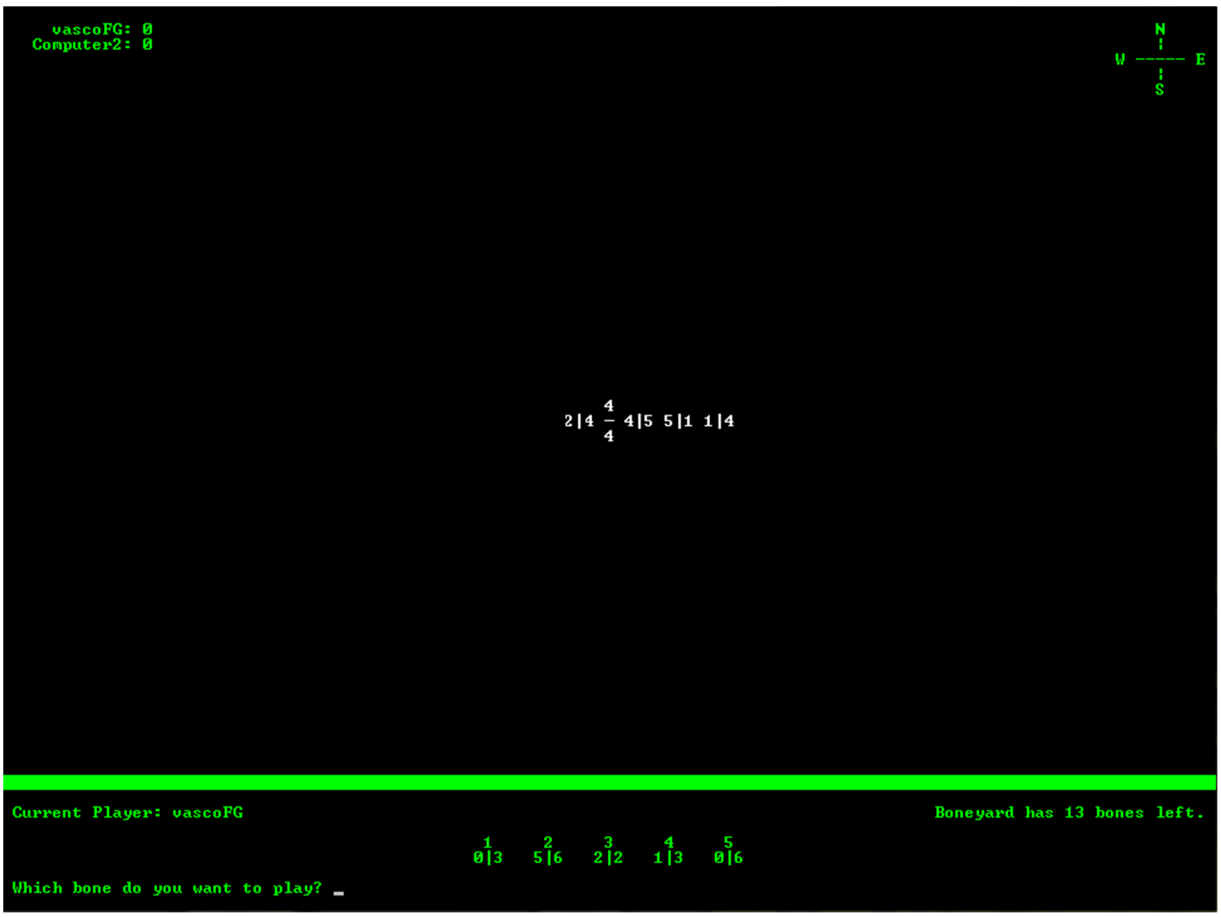
Ecrã inicial



Instruções

Número de jogadores humanos

Número de jogadores controlados pelo computador



Ecrã de selecção de jogada

Vitória

Conclusão:

Concluímos que é muito complicado planear a estrutura de classes de um projecto um pouco complexo antes de iniciar a implementação delas pois só depois de começarmos a programar conseguimos perceber as limitações que estão associadas à estrutura que planeámos seguir. No entanto, estamos confiantes que seguimos a ideia geral da estrutura que planeámos no início. Existem bastantes funções novas nos diagramas de classes, mas elas são resultado da simplificação do código. Se não as tivéssemos criado, poderíamos ter acabado o projecto com exactamente o mesmo número de funções que estavam planeadas mas o código seria muito difícil de ler e compreender, o que, na nossa opinião, é uma pior solução.

Pensamos ter superado os objectivos deste trabalho e aplicado os conhecimentos adquiridos durante este semestre de uma maneira produtiva e correcta.