**Package Game**

Beginning

1st Command

“$”

Other cases

“d”

“s”

“q”

“b” or “b bet”

Print Credit

Illegal Command

Make a Bet

Leave Game

Print Statistics

Make a Bet

2nd Command

Other cases

“d”

“s”

“$”

Deal Cards

Print Statistics

Print Credit

Illegal Command

3rd Command

Print Advice

“h” or “h cards”

“s”

“$”

Other cases

“a”

Print Credit

Print Statistics

Illegal Command

Hold cards

Check Result

Present Results

**Beginning**

Em primeiro lugar, o programa fará o processamento de todos os argumentos dados pelo utilizador, fazendo as devidas verificações. Se os argumentos não vieram da forma especificada pelo enunciado do projeto, o nosso programa irá sair corretamente.

Caso estejamos perante o modo Interactive, este inicializará os atributos declarados na classe abstrata Game. Estas variáveis são:

* Hand – Esta variável serve para o utilizador ter sempre uma mão com 5 cartas. Mais à frente teremos mais detalhes;
* Credit – Esta variável servirá para o utilizador se poder manter a par do seu crédito;
* Runs – Número de jogadas que já foram feitas. É inicializada a zero;
* Stat – Variável que guarda as estatísticas para depois serem apresentadas;
* Paytable – Variável que guarda a paytable, que se acederá para saber qual é o ganho correspondente a cada jogada.

Depois de inicializados todos os atributos declarados, o programa irá aceder ao método init() e o programa passa para o Core.

Caso estejamos no modo de Simulation, para além de serem inicializados todos os atributos descritos acima, são inicializados outros dois, que estão declarados apenas para esta classe que estende a classe Game:

* Nbdeals – Esta variável representa o número de jogadas que foram feitas;
* betAmount – Esta variável representa o crédito que irá ser jogado em todas as jogadas, e caso o crédito esteja abaixo deste valor, o programa terminará.

Caso estejamos no modo Debug, para além dos atributos supramencionados (excluindo o Nbdeals e betAmount), são inicializados alguns atributos declarados nesta classe que mais uma vez estende a classe Game:

* cmdcount – índice do vetor de comandos. Este vai sendo incrementado ao longo do jogo, sendo que este se encontra sempre sobre o próximo comando a ser lido;
* cardcount – índice homólogo ao anterior para as cartas. Também vai sendo incrementado ao longo do jogo para que esteja sempre sobre a próxima carta a ser lida;
* String [] cmdaux – este vetor de Strings serve para ir buscando o próximo comando a executar.

Assim que o programa é iniciado, o utilizador terá de fornecer dois ficheiros de texto, um correspondente aos comandos e outro correspondente à sequência de cartas que irá ser jogada ao longo de todo o jogo. Estes dois ficheiros serão ambos processados, de forma a ficarem num formato de vetor de Strings, em que cada posição desse vetor corresponde a um carácter no caso da leitura do ficheiro de comandos. Caso nos refiramos à leitura do ficheiro de cartas, cada posição do vetor de Strings terá uma carta. Este é processado por sua vez e após o mesmo, teremos um Deck cujo o número de posições é igual ao número de cartas que estavam presentes no ficheiro de entrada de cartas.

**Core**

A base do programa está na classe abstrata *Game*. Como referido acima, esta classe abstrata é estendida pelas classes Interactive, Simulation e Debug. Cada uma delas é um modo de jogo, portanto cada uma delas irá ter os métodos correspondentes para ir buscar os comandos:

* Interactive irá recorrer ao teclado;
* Simulation irá fazer sempre os mesmos comandos ao longo do jogo;
* Debug irá ler o vetor de Strings correspondente ao comando de entrada do ficheiro de comandos.

No fluxograma, que se encontra na figura X, foram omitidos alguns passos e algumas verificações, contudo é possível verificar a estrutura do Algoritmo. Para ir buscar cada um dos três comandos, na classe Game existem 3 métodos abstratos (Process1(), Process2() e Process3()) que são implementados precisamente por cada uma das classes que estendem a classe Game (Interactive, Debug e Simulation).

1º Comando

Para começar, é recebido o primeiro comando. Só é possível avançar desta primeira fase, caso o comando recebido seja duma aposta válida ou então um *deal*, caso já tenha sido feita uma jogada anteriormente. Em qualquer um dos casos, a aposta é considerada válida se cumprir todos os requisitos descritos no enunciado e se o crédito for igual ou superior à aposta feita.

2º Comando

Aqui, o utilizador tem poucas opções, sendo que todas elas são de apenas um carácter. A primeira é de imprimir o crédito atual e a segunda é a de receber as 5 cartas. Caso, no 1º comando se tenha obtido um comando deal válido, o algoritmo passa à frente desta requisição de comando e não pede nenhum comando.

3º Comando

Existem várias opções válidas aqui. Independentemente de ser o programa a “ler do ficheiro” ou de ser o utilizador a escrever do teclado, existe sempre a possibilidade de fazer Hold, Advice, Credit ou Statistics. Sendo que a única forma de avançar é fazer uma jogada de Hold válida. Para o caso do modo Simulation, o 3º comando será sempre o que é retornado pelo método Advice, de forma a fazer a melhor jogada possível.

Depois de todas as cartas jogadas, é verificado através do método Check Result se a mão ganhou e é verificada na Paytable o crédito correspondente, imprimindo depois os respetivos resultados.

É importante referir ainda que existem mais dois métodos que são abstratos:

* getHand() – A forma como cada uma das classes estendidas da classe abstrata Game irá buscar a mão é diferente, portanto aqui optou-se por criar um método abstrato para o tipo hand;
* holdPlay(String cmd3) – Este método também é abstrato, porque mais uma vez cada uma das classes irá fazer o comando Hold será feito de maneira diferente pelas classes estendidas.

Credit

Esta classe é apenas para o utilizador fazer as apostas e guardar o valor das mesmas, o saldo inicial e o saldo atual. Desta forma é possível imprimir os respetivos valores e fazer as devidas contas com os mesmos.

**Package Deck, Hand e Cards**

Cards

As cartas são a base deste package, sendo que apenas são constituídas pelo suit, pelo rank e por um inteiro diferente para cada carta que é o value. O value servirá mais tarde para algumas verificações.

Deck

O Deck é o baralho das cartas e aqui temos dois construtores:

* O primeiro não necessita de argumentos, porque irá criar um baralho de cartas com apenas 52 cartas baralhadas completamente aleatoriamente. Este construtor é utilizado pelo modo de jogo Interactive e Simulation;
* O segundo necessita de saber quais são as cartas a colocar no baralho e este não é baralhado ao início do jogo. Este baralho irá ser utilizado pelo objeto instanciado do modo de jogo Debug.

Devido a termos dois baralhos com a mesma estrutura, um deles em que queremos que este seja constantemente baralhado e que tenha apenas as 52 cartas e outro em que queremos que ele esteja pela ordem lida no ficheiro e que tenha tantas cartas quantas possíveis, recorremos ao *Overloading* de métodos para podermos ter estes dois Decks numa só classe. Para além das duas principais diferenças descritas atrás, no segundo caso temos que ter atenção às cartas que vão buscar e para este efeito, foi criado um inteiro (cardcount) para manter o registo do índice da próxima carta a ir buscar.

Hand

Esta classe irá ter um vetor de 5 cartas que é a mão do jogador e um baralho correspondente. Uma vez mais, existem vários métodos que são *overloaded* para que o baralho correspondente ao modo de Debug seja instanciável e para que seja possível ir buscar cartas, mantendo o índice do vetor de cartas do baralho atualizado.

Nesta classe encontra-se a método *Hold* que é das mais importantes do Projeto. Esta recebe como parâmetro uma *String* ‘*hold*’, que contém o comando de *hold* das cartas, e vai retirar do

baralho novas cartas para substituir as que descarta.

(TO BE CONTINUED)

**Package stats**

Statistics

Como o nome sugere, nesta classe estão as estatísticas referentes ao jogo. Um objeto desta classe é instanciado logo no início do jogo. Desta forma, mesmo antes de fazer qualquer aposta é possível imprimir as estatísticas de jogo.

De cada vez que é feita uma jogada e a respetiva verificação, este objeto sofre uma atualização.

Credit

Esta classe é pública, mas com *fields* privados de modo a esconder a implementação feita. Esta contém um crédito inicial que é atribuído no construtor, um crédito atual que pode ser atualizado através de métodos criados e ainda o valor da quantia a ser apostada(*prevBet*), que pode ser alterada. Existe um método chamado *bet* que através do *prevBet* atualiza o crédito atual.

**Package strategy**

PayTable

A *PayTable* é uma classe pública e abstrata, que pode ser utilizada para qualquer variante de jogo, uma vez que permite associar entre si dois objetos e um inteiro. Deste modo é possível introduzir na tabela o valor de um crédito de qualquer tipo de objeto associado a um tipo de mão de um qualquer objeto. A implementação é feita com uma *hashtable* que possua vez possui como *keys* os tipos de mãos e como *values* outras *hashtables*. Estas últimas têm como *key* o crédito e como *value* o montante associado. Existem métodos para retornar os montantes da tabela, para retirá-los e para inseri-los.

PayTable107

Esta classe é pública e uma subclasse da *PayTable* em que o construtor cria de imediato a tabela com os montantes válidos para a variante do jogo *double bónus 10/7 video poker*. Neste caso os tipos das mãos introduzidos na tabela são *Strings* e os créditos são inteiros.

Verify

Esta classe é publica abstrata e tem como objetivo receber como argumento um objeto *hand* e verificar o resultado do tipo de mão. Apenas se colocam mãos que são gerais nos vários jogos de poker, como Royal Flush, trio, *flush*, etc. Muitas das vezes o retorno dos métodos é um objeto da classe RetVerify que irá ser explicada posteriormente.

Verify107

*Verify107* é uma subclasse de *Verify* e contém verificação de mãos mais específicas para a variante de poker deste projeto. A visibilidade é do tipo *package* para não ser estendido fora do package e esconder a implementação que é demasiado específica. Alguns exemplos de métodos são o *FourAces* e o *InsideStraight*.

RetVerify

O propósito desta classe é de tornar possível enviar as posições das cartas e um inteiro com uma informação extra. As posições das cartas são introduzidas num *field* posRet, que é um vetor de inteiros, e o *nRet* é o inteiro que a maior parte das vezes indica quantas cartas são necessárias para que o tipo de mão se verifique. Como é utilizada acessoriamente definiu-se a visibilidade como *package*.

Strategy

A *Strategy* é basicamente uma classe abstrata pública que indica que ao ser criada uma subclasse desta, existem dois métodos *static* que deveram ser *override* para cada tipo de jogo. Os métodos são o *CheckResult*, onde se deve verificar o resultado de cada ronda de um jogo de poker, e o *Advice* que informa ao jogador a melhor jogada a tomar de acordo com a estratégia em vista. Para estes métodos serem *static* têm um corpo em que apenas é retornado *null*.

Strategy107

Esta é uma subclasse pública de *Strategy* em que é desenvolvido, no método Advice, a estratégia das melhores jogadas a fazer em cada mão referente à nossa variante de jogo. Também foi alterado o *CheckResult* onde é: verificado o resultado da mão de poker, são atualizadas as estatísticas e o crédito, e ainda é retornado uma *String* com a informação do resultado da ronda. Existe ainda um método estático privado que é chamado apenas dentro do *Advice* com o intuito de ser criada uma *string* com as cartas a que o jogador deve fazer *hold*.

**Observações**

Os packages foram divididos segundo a semelhança lógica entre as classes e também de modo a ser possível esconder a implementação, utilizando a visibilidade *package*, sem comprometer a funcionalidade do programa.

Decidiu-se colocar a maior parte das classes públicas de modo a dar a possibilidade a futuros implementadores utilizarem pequenos módulos de classe para fazerem os seus próprios jogos em vez de apenas tornar público o jogo, deixando privadas as classes que este utiliza.

Graças a isto é possível estender o *Deck*, a *Hand*, o *Strategy*, entre outros, para criar novas variantes de jogo, em que se pode reutilizar métodos já criados.