TÉCNICO LISBOA

Programação Orientada por Objetos

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Projeto

Math in Casinos: Video Poker

Grupo nº 14

João Vieira, nº 79191

Guilherme Reis de Moura, nº 78767

Ana Catarina Santos, nº78409

Professora Alexandra Carvalho

Índice

Introdução

Neste projeto, teve-se como objetivo a programação do jogo *VideoPoker*, na variante *Double Bonus 10/7*. Para isso, tomaram-se várias decisões em relação ao design do UML do projeto e, consequentemente, à implementação do código.

Começou-se por criar as classes de baixo nível, como as cartas, o baralho e a mão, passando para a verificação de todas as combinações possíveis. Quando se concluiu a implementação de toda a estratégia associada a esta variante, passou-se para a implementação dos vários modos de jogo e, finalmente, para a implementação da interface gráfica.

Ao longo deste relatório são explicadas as decisões de lógica tomadas para o funcionamento do programa, quer para o design do UML, quer para a escrita do código, assim como o objetivo de se criar os vários ficheiros de teste para utilizar no modo *Debug* e algumas estatísticas realizadas para analisar o desempenho do programa.

UML

Nesta secção são explicadas as decisões mais importantes relacionadas com o design do UML criado para o projeto.

A ideia geral do design do UML foi conseguir-se alterar o jogo adicionando-se novas classes, nomeadamente criar uma variante de poker sem ser o *DoubleBonus*, com regras específicas relacionadas com o novo tipo de jogo, tendo que implementar a interface *PlayVariant*. Também é possível colocar no baralho as cartas que se pretendam, não estando obrigado a usar sempre o baralho normal de 52 cartas, usado para este jogo específico.

Os modos de jogo e a *GUI* têm todos uma estrutura idêntica, pelo que se decidiu criar uma classe chamada Match que teria os atributos e métodos em comum a todos e, assim, os 3 modos de jogo e a *GUI* serão subclasses do *Match*.

Lógica Implementada

Neste capítulo apresentam-se as explicações e os porquês de se terem feito algumas decisões, ao longo do desenvolvimento do trabalho, em cada classe implementada.

Cards

Para esta classe, considerou-se o tipo enumerado para definir o número e o naipe de cada carta. Utilizou-se este tipo de dados porque as cartas só podem ter determinados números e naipes e, assim, evitam-se tentativas de adicionar números ou naipes que não existam, sem precisar de se fazerem constantes verificações.

Nesta classe, apenas se implementaram métodos para verificações de certos atributos das cartas.

Deck

Para esta classe, são utilizados dois construtores – um para a versão do jogo *Interactive* e *Simulation*, onde é necessário um baralho pré-definido, com as 52 cartas habituais; outro para a versão *Debug*, que é construído com as que lhe são passadas como argumento.

Para baralhar as cartas, foi utilizado o método da Classe *Collections* denominado *shuffle*, que utiliza o algoritmo *Fisher-Yates shuffle*.

Os restantes métodos são considerados simples como, por exemplo, adicionar ou remover cartas do baralho.

DoubleBonus

O quadro com os ganhos fornecidos no enunciado foi armazenado num *HashMap*. Decidiu-se, também, colocar num *HashMap* a tabela das estatísticas, para se associar uma *String* a valores inteiros de maneira eficiente.

No método *advice*, são criados três *HashMap*: um para identificar as cartas que tenham o mesmo número, outro para cartas com o mesmo naipe e um último que mostra os vários candidatos de *straights* possíveis. Os três métodos criados para implementar estes *HashMap* estão descritos na classe abstrata *PokerRules*, que é estendida pelo *DoubleBonus*.

- *computeSameRank* → neste método, é criado um *HashMap* que contém como *keys* todos os *ranks* das cartas e endereça *ArrayLists* compostas dos endereços das cartas do *rank* da *key*. Por exemplo, se na mão existirem 3 cincos, é guardado no índice correspondente ao *rank five* as três posições correspondentes às cartas.
- *computeSameSuit* → raciocínio análogo ao do método *computeSameRank*, mas para os naipes.
- *computeCandidateStraight* → O raciocínio utilizado para a criação deste método pressupõe a ordenação das cartas por *rank* e que todos os possíveis *straights* podem ser contabilizados através do grafo apresentado na figura 1.

$$\begin{cases} 0-1-2-3-4 \\ 1-2-3-4-0 \end{cases}$$

$$\begin{cases} 0 & \begin{cases} 1 & \begin{cases} 2 & \begin{cases} 3 \\ 4 \end{cases} \\ 3-4 \end{cases} \\ 2- & 3- & 4 \end{cases} \\ 1 & \begin{cases} 2 & \begin{cases} 3 & \begin{cases} 4 \\ 0 \end{cases} \\ 4 & -0 \end{cases} \\ 3 & -4 & -0 \end{cases} \\ 2 & -3 & -4 & - & 0 \end{cases}$$

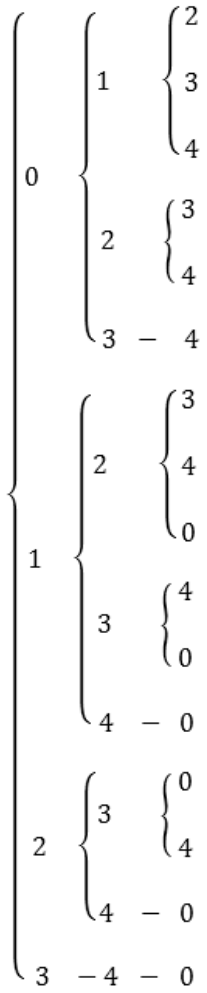

Figura 1 - esquema em forma de árvore com as posições das cartas possíveis para a existência de um straight.

É de notar que o grafo da figura 1 apresenta possibilidades de *straight* com um mínimo de três cartas, pois, para este jogo, não são contabilizados candidatos a *straights* com apenas duas cartas. Também é tido em conta o facto da existência do ás como o *low* ou *high ace*. Após a criação do *HashMap* com todas as possibilidades apresentadas na figura 1, sendo a sua *key* o número de cartas em falta para o straight, ou seja, 0, 1 e 2, são realizados cálculos para verificar se existe, ou não, um *straight* com as combinações possíveis. Para esta verificação, é subtraído o valor da última carta pelo valor da primeira e, se o resultado for inferior a cinco, indica que há a possibilidade de ocorrência de um *straight*. É tido em especial atenção o caso da carta *ace* e o caso de existirem cartas com o mesmo *rank*. Esta é uma função geral para filtrar todas as possibilidades de geração de *straights*.

Estes métodos são usados apenas uma vez no código, para cada chamada do método *advice* e *evaluateHand*, sendo apenas consultados os *HashMap* criados por estes.

PokerRules

Nesta classe, estão definidos todos os métodos para as verificações da lista de prioridades encontrada no enunciado do projeto, usada na função *advice*, assim como métodos auxiliares (como, por exemplo, o método *intersect*, que é útil nas funções de *straight* e *flush*) para uma implementação mais eficiente dos métodos gerais. Escolheu-se fazer um método para cada possibilidade para uma mais fácil leitura do código, mas principalmente porque alguns métodos são

mais específicos de outros como, por exemplo, os vários casos de *straight*, que utilizam o método geral *straight*.

Game

De maneira análoga ao que acontece na classe *Deck*, existem também dois construtores: um para se utilizar o baralho *default* de 52 cartas e outro para se atribuir cartas dadas pelo utilizador.

Foi criado um método *PrepareRound* que chama o método *shuffle* do *deck* porque o método *Debug* não utiliza este método e, assim, este método é apenas utilizado no modo *Interactive* e *Simulation*.

Match

De acordo com a lógica explicada no enunciado, é feita uma máquina de estados, composta por 4 estados: *start*, *round*, *interval* e *afterBet*. Estes estados são aplicados nos modos de jogo *Interactive* e *Debug*, uma vez que, no modo *Simulation*, os comandos são geridos automaticamente pelo programa. Na figura 2 encontra-se uma representação da máquina de estados.

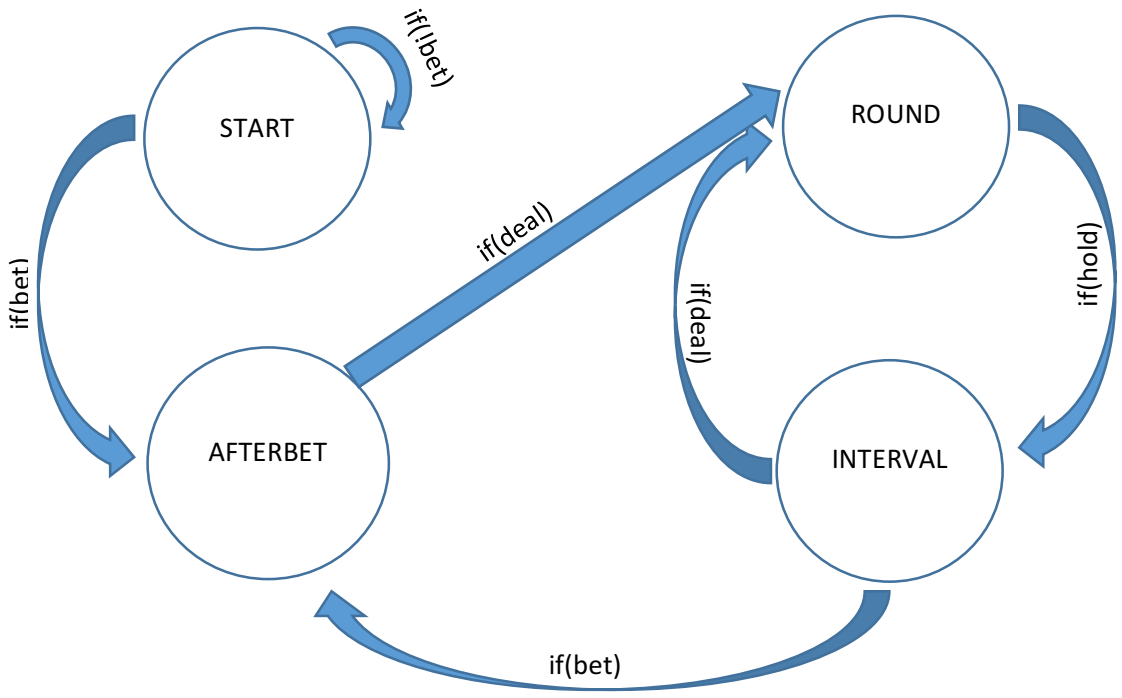

Figura 2 - Máquina de Estados para a execução do jogo no modo interactive e Debug.

- *start* → estado de começo do jogo. Apenas se podem fazer os comandos *bet* e *quit*.
- *afterBet* → estado seguinte ao *start*, ou seja, estado seguinte a ser feita uma aposta nova. Só se pode fazer o comando *deal*.
- *round* → estado seguinte ao *afterBet*, ou seja, estado em que se decidem as cartas com que se quer, ou não, ficar na mão. Podem-se fazer os comandos *hold* e *advice*.
- *interval* → estado seguinte ao Round, ou seja, o estado em que o jogador recebe os dados, antes de se fazer uma nova aposta. Podem fazer-se os comandos *bet*, *deal* e *quit*.

Os comandos *credits* e *statistics* podem ser efetuados em qualquer um dos estados, não sendo alterado o mesmo.

Debug

A diferença entre esta classe e a classe *Interactive* é a forma de ler os argumentos: enquanto que, no *Interactive*, os argumentos são lidos do teclado, no modo *Debug* são lidos de ficheiros. Na leitura dos ficheiros, são ignorados os espaços independentemente do seu tamanho e o ficheiro também pode ter várias linhas. Todos os valores retirados do ficheiro dos comandos são colocados num *ArrayList*, sendo que comandos que possam ter argumentos extra, como o *bet* e o *hold*, são armazenados como *String* numa só célula do *ArrayList*, para facilitar a implementação nos métodos já descritos.

GUI

Foi decidido implementar uma interface gráfica para a utilização dos modos *Interactive* e *Simulation*. Inicialmente, é apresentada uma janela a pedir o crédito inicial e, de seguida, aparece a janela principal com o jogo, conforme a figura 3.

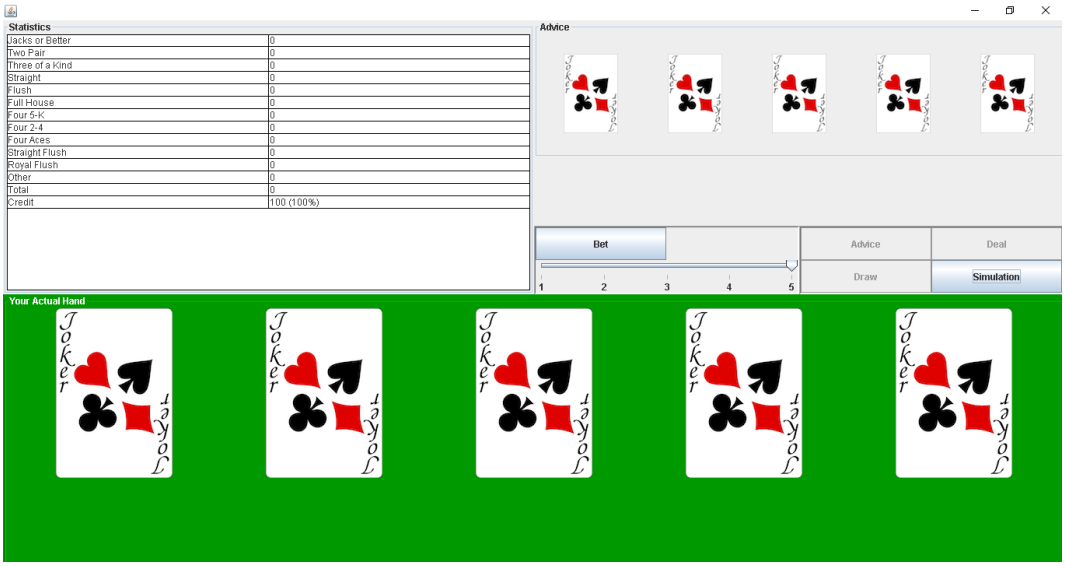

Figura 3 - interface gráfica na fase inicial do jogo.

Se se pretender correr o jogo no modo *Simulation*, a qualquer momento do jogo, é possível premir-se o botão *Simulation* e apresenta-se a janela apresentada na figura 4, mas sem os valores em *Credit*, *Bet* e *N Bets* preenchido.

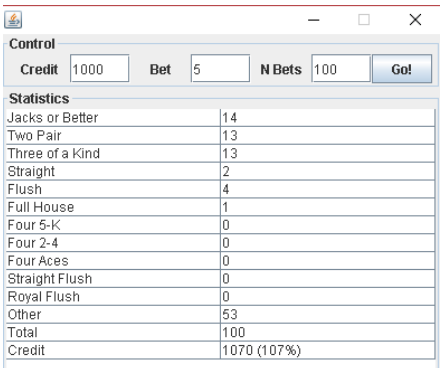

Figura 4 - Janela da Interface para o jogo em modo Simulation.

Na parte inferior, são mostradas as cartas que, inicialmente, estão viradas para baixo; no canto superior esquerdo, apresenta-se uma tabela que vai sendo atualizada ao longo do jogo com as estatísticas e, no canto superior direito, encontra-se uma janela com o *advice*, que mostra as cartas com que se devem ficar, outra janela com o texto de vitória ou derrota e, por fim, uma janela com botões que implementa os comandos possíveis de usar pelo utilizador. Esta interface implementa a máquina de estados descrita anteriormente, não deixando o utilizador realizar funções que não estão permitidas em cada estado. Em relação ao comando *bet*, o utilizador pode escolher o valor da sua aposta, pelo *JSlider* ou, se não for utilizado, a aposta é feita de acordo com o enunciado, ou seja, ou aposta os 5 créditos caso seja a primeira vez, ou aposta o valor anterior. Em relação ao comando *hold*, este é efetuado pelo utilizador ao carregar nas cartas com que pretende ficar, seguido da pressão do botão *draw*, de maneira análoga ao que acontece no exemplo.

Ficheiros de teste criados

Os primeiros 3 ficheiros usados no modo *Debug* foram focados na demonstração da correta análise de todas as mãos vencedoras através de todas as funções implementadas, consoante alguns casos apresentados no enunciado.

Para o quarto conjunto de ficheiros de cartas e comandos, é testada a máquina de estados, ou seja, se é possível correr certos comandos em certas partes da máquina de estados, como, por exemplo, começar o jogo com um *deal*. É também testado a entrada de comandos ilegais como, por exemplo, uma aposta com um valor diferente de [1, 5], o comando *hold* com índices fora do intervalo [1, 5] ou comandos que não estejam programados.

Para o último conjunto de ficheiros, foram testados erros relacionados com os ficheiros das cartas como, por exemplo, cartas que não existam. Caso algum *input* não seja válido, o programa não é iniciado.

Estatísticas

A tabela 1 mostra as estatísticas para cada mão possível, considerando mil rondas, cada uma com 1 milhão de apostas. Estas estatísticas podem ser comparáveis com as presentes no site onde o projeto foi inspirado. Verifica-se que os valores obtidos são muito semelhantes aos do site.

Name of Hand	Average number of hits in 1M rounds (*)	Probability
Jacks or Better	188206	18,8207%
Two Pair	124262	12,4262%
Three of a Kind	72162	7,2162%
Straight	14650	1,4651%
Flush	15260	1,5261%
Full House	11192	1,1192%
Four 5-K	1607	0,1607%
Four 2-4	522	0,0523%
Four Aces	201	0,0201%
Straight Flush	110	0,0111%
Royal Flush	19	0,0019%
Other	571803	57,1803%

Tabela 1 - Probabilidade de se obter as mãos consideradas no jogo.

(*) média de ocorrências em mil sequências de 1 milhão de rondas.

A tabela 2 mostra a percentagem de retorno em relação aos valores das apostas feitas em cada ronda. Estes resultados foram obtidos a partir da média da percentagem do retorno de mil sequências, cada uma com um milhão de apostas. Também é mostrado o intervalo a 95% para estes resultados.

Amount for each bet	Average return	Confidence Interval (95%)
1	98,5736%	[97,8057; 99,3241]
2	97,2186%	[95,6812; 98,8322]
3	95,8337%	[93,6151; 98,1100]
4	94,4357%	[91,2880; 97,5112]
5	98,2470%	[93,3180; 103,5335]

Tabela 2 - Percentagem de retorno tendo em conta o valor apostado em cada ronda.

A figura 5 contém os histogramas cuja análise produziu os resultados apresentados na tabela 2.

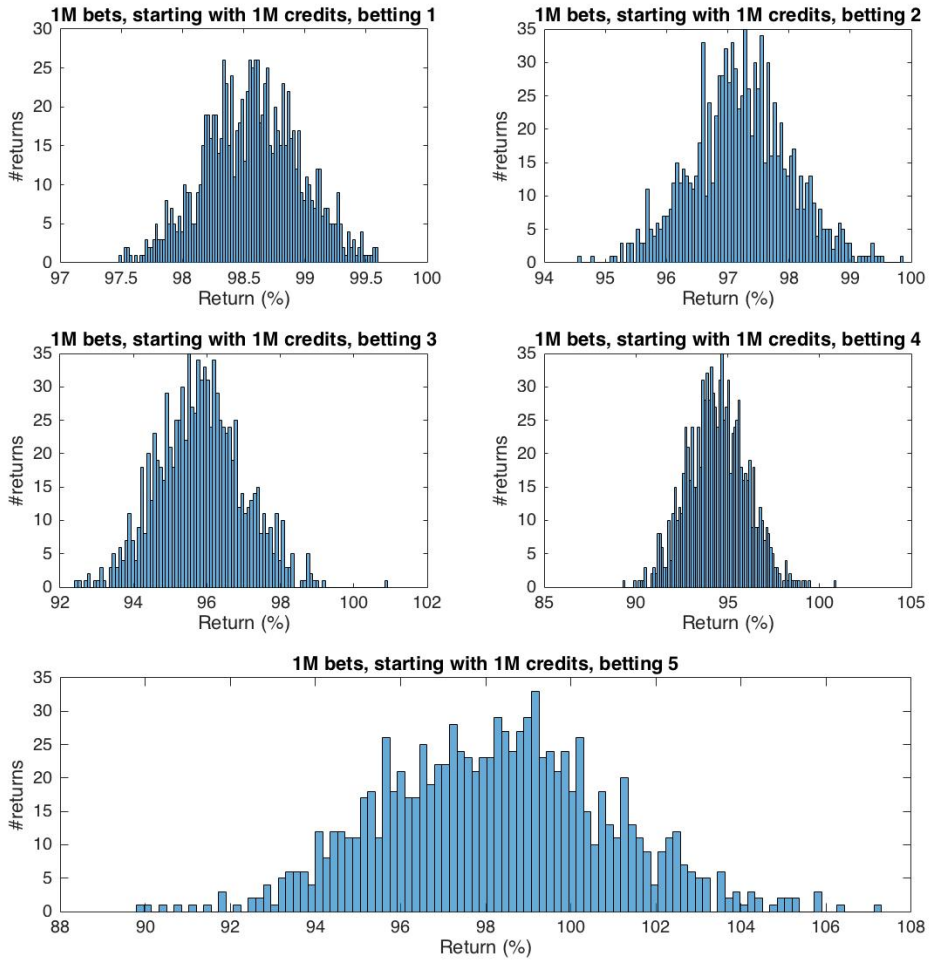

Figura 5 - Histogramas analisados para a realização das estatísticas apresentadas na tabela 2.

Conclusão

Com este projeto, foi possível desenvolver toda uma aplicação na linguagem de programação Java, o que proporciona aos alunos toda uma gestão de tempo, recursos e decisões que, de outra forma, não seria possível.

Na nossa opinião, a parte mais complexa do trabalho foi a implementação do método *advice*, pois é este o responsável, para além do correto funcionamento do programa, do desempenho eficiente do mesmo.

O correto funcionamento do programa foi testado de várias formas: com as 81 mãos difíceis fornecidas no enunciado (estas foram deixadas no programa num método privado apenas para efeitos de desenvolvimento) e com os ficheiros de teste criados e submetidos.

Todo o projeto foi desenvolvido de forma a poder ser utilizado para outras variantes de jogo, sem ser necessário alterarem-se as classes implementadas. Também se decidiu criar a interface gráfica com os modos *Interactive* e *Simulation*, para uma melhor experiência ao utilizar a aplicação.

Foram ainda feitas estatísticas acerca do retorno em função do valor apostado fazendo sequências de mil simulações com 1 milhão de rondas cada a começar com um crédito de 1 milhão para cada montante de aposta possível. Pela análise dos resultados verificou-se que o retorno teórico para cada um dos casos aproxima-se do intervalo de confiança obtido por simulação e que a probabilidade de saída de cada mão está de acordo com aquelas apresentadas no site onde o projeto foi inspirado.

Concluindo, pensamos que o trabalho foi implementado de uma maneira correta e eficiente de acordo com os requisitos.