Programação Modular
Trabalho Final
Visualização Interativa de Agentes Inteligentes

Daniel Diniz - 2008046090 - daniel.diniz@dcc.ufmg.br
Diogo Santana - 2011054308 - diogo.santana@dcc.ufmg.br
Frederico Figueiredo - 2010054371 - fredfig@dcc.ufmg.br
Sandro Miccoli - 2009052409 - smiccoli@dcc.ufmg.br

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

17 de junho de 2015

Resumo. *Esse relatório descreve como foi implementado a Visualização Interativa de Agentes Inteligentes, proposto como tema para o Trabalho Final. A Seção 1 introduz o problema proposto e dá uma visão geral da solução implementada. Cada seção irá descrever detalhes do sistema desenvolvido, abrangendo desde o planejamento (Seção 2), as decisões de implementação (Seção 4) e os testes realizados (Seção 6). Finalmente concluímos (Seção 7) a documentação com reflexões sobre o aprendizado durante a execução do trabalho.*

1. Introdução

Durante as aulas da disciplina vimos muitos exemplos de como implementar formas geométricas dinâmicas, então decidimos aplicar isso no nosso trabalho final. O objetivo do trabalho é unir os conceitos que aprendemos sobre programação modular e padrões de projeto e aplicá-los num contexto que seja esteticamente agradável. Isso será feito utilizando apenas formas geométricas e linhas com o intuito de gerar padrões visuais interessantes. O modo como chegaremos nesse resultado será utilizando algoritmos de movimentação de inteligência artificial, usando conceitos como bug algorithm, campos de potencial e desvio de obstáculos (steering). O planejamento inicial consiste em construir agentes inteligentes que se atraem e repelem. Cada grupo de agentes terá uma atração a certo grupo e repulsão a outros. Por causa dessa característica de atração e repulsão, no momento em que vários agentes são atraídos por um único agente, é esperado que eles entrem em equilíbrio e formem padrões geométricos regulares. Já é possível perceber várias possibilidades de utilização de padrões de projeto: Flyweight, caso instanciarmos muitos elementos gráficos; Factory Method, para a geração de diferentes tipos de agentes; State, para definir e controlar dinamicamente o estado de cada agente.

Modelagem inicial do problema. Construção de agentes básicos. Implementação do bug algorithm com campos de potencial. Implementar comportamentos aleatórios para os agentes. Implementação de uma interface gráfica Escrever documentação do projeto

Possível utilizar a biblioteca controlP5 do Processing, que permite alterar valores do código em tempo de execução.

O problema proposto neste trabalho foi a especificação e implementação de uma visualização jogo de cartas a ser escolhido pela dupla. O jogo escolhido por nós foi o

Pôquer Chinês (Big Two). O objetivo principal do jogo é vencer ao acabar com todas as cartas da mão recebida pelo jogador. A proposta do trabalho prático é proporcionar uma interface simples por linha de comando que seja jogável. O programa implementado será detalhado superficialmente aqui, porém a (Seção 4) irá explicar em profundo os detalhes de implementação.

2. Planejamento

O planejamento que antecedeu à implementação ocorreu da seguinte forma: estudamos as regras do jogo existentes na internet, como são muitas variantes, tivemos que escolher uma e implementar ela. Após isso, definimos os módulos principais que o jogo deveria ter, suas classes e métodos. Com isso, geramos um diagrama UML para melhor compreensão do que iríamos desenvolver. Além disso criamos um repositório git online para que pudéssemos trabalhar em paralelo no mesmo código e caminhar rápido com o trabalho.

3. Regras

O Pôquer Chinês [BigTwo] é um jogo de cartas cujo objetivo é ser o primeiro a desfazer de todas as suas cartas. É por vezes chamada de pôquer chinês devido à sua origem chinesa e uso de mãos de pôquer, embora seja um jogo de natureza completamente diferente. Ele é jogado tanto casualmente como por dinheiro. É geralmente jogado com dois a quatro jogadores, com o baralho inteiro sendo distribuído para todos os jogadores, ou 13 cartas para cada jogador. Após as cartas terem sido distribuídas, no geral, o jogador com a menor carta, o 3 de ouros, joga primeiro, com o jogo prosseguindo em uma direção horária. Cartas podem ser colocadas na mesa como únicas, duplas, trios ou em grupo de cinco, utilizando as regras do pôquer, inclusive na primeira jogada feita por quem tem o 3 de ouros. Depois do início do jogo dado pela jogada do primeiro jogador, o jogador a seguir precisa colocar um mesmo número de cartas que sejam maiores do que aqueles colocados pelo jogador anterior. Caso um jogador não possa ou não queira colocar uma carta ou combinação, ele passa seu turno. Caso todos os oponentes decidam passar, então o jogador remanescente possui o direito de colocar qualquer combinação permitida. O naipe das cartas é utilizado como critério de desempate, sendo a ordem do menor para o maior: ouros, paus, copas e espadas.

3.1. Tipos de combinações de jogo

Essa subseção irá listar todas as possíveis combinações de jogo.

Uma carta

Qualquer carta do baralho, ordenadas pelo valor da carta (sendo 3 o menor e 2 o maior), com o naipe sendo o critério de desempate. Por exemplo, o 2 de espadas é maior do que qualquer outra carta do jogo, o A de espadas é maior do que o A de ouros, esta sendo maior do que um K de espadas.

Par

Quaisquer duas cartas iguais, com o maior naipe das duplas sendo utilizado como critério de desempate. Um par de K de espadas e K de ouros é melhor que um par de K de copas e K de paus.

Trinca

Três cartas de mesmo valor, o desempate é feito através do maior valor.

Cinco cartas

Existem cinco mãos válidas, ordenadas a seguir do menor para o maior:

Sequência

Cinco cartas em sequência, não do mesmo naipe. Caso empate ganha aquele com a maior sequência: 10-J-Q-K-A >... >2-3-4-5-6 >A-2-3-4-5, ou seja, o rank da maior carta da sequência determina qual é a maior, o A e o 2 nas sequências baixas não influenciam no valor da sequência;

Flush

Cinco cartas do mesmo naipe, não em sequência. O desempate é determinado primeiro pela carta de maior valor, e então pelo naipe mais alto;

Full House

Uma trinca e um par, a trinca mais alta é utilizada como critério de desempate;

Quadra + 1

Quatro cartas do mesmo valor mais uma carta qualquer. Observe que, ao contrário do pôquer, uma quadra não pode ser utilizada por sí só. O desempate é determinado pelo valor de uma das cartas da quadra.

Sequência de mesmo naipe

Uma sequência e um flush ao mesmo tempo. O desempate é determinado primeiro pela maior carta, e então pelo maior naipe.

Sequência real

O mesmo que a sequência de mesmo naipe, mas limitado a 10, J, Q, K e A. Esta é a maior combinação possível. Apenas uma sequência real com um naipe maior pode vencer outra sequência real. A sequência real de espadas é a combinação de cartas mais poderosa do jogo.

4. Implementação

Após gerar o diagrama UML que indicava quais classes precisariam ser implementadas, foi definida como seria montada a estrutura de dados do programa. Abaixo, na Figura 1, é possível ver uma versão simplificada do diagrama de classes do sistema implementado. Simplificada pois não contém nenhuma informação de atributos ou métodos, apenas de relacionamento entre classes e pacotes. Isso foi feito para não poluir a documentação com um diagrama tão complexo. A seguir também é possível ver o diagrama de atividades (Figura 2) do fluxo do programa. Ao final deste documento é possível ver o diagrama de classes completo (Figura 3).

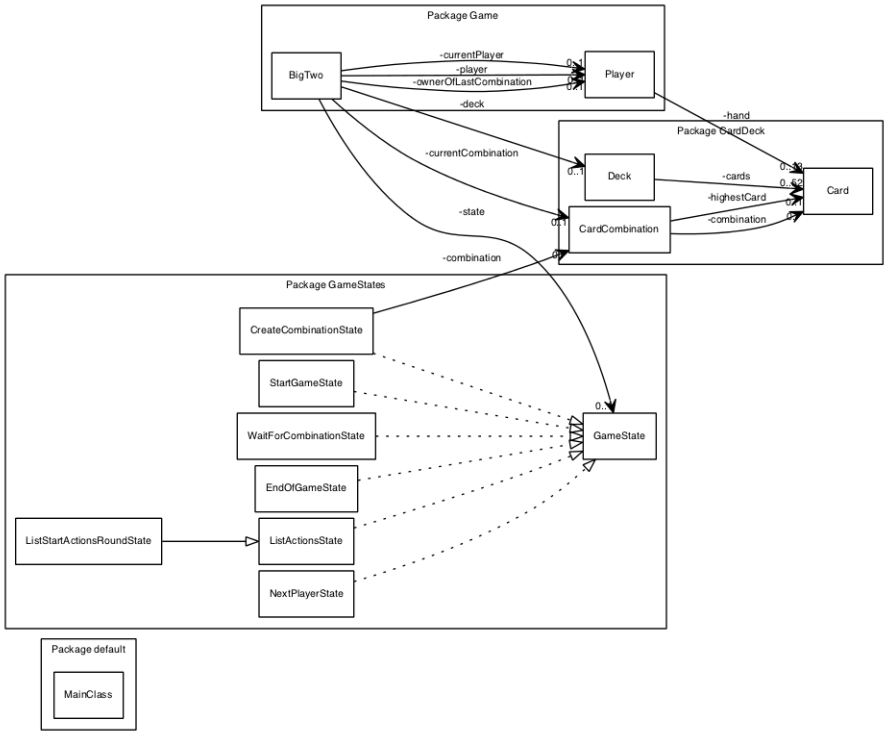

Figura 1. Diagrama de classes simplificado

5. Classes implementadas

MainClass

Classe principal que inicia a execução do jogo.

Card

Classe que contém as informações sobre uma carta, como o seu valor e o naipe, os métodos padrões *getters*, um para o valor e outro para o naipe, além de um método para comprar qual carta é maior.

CardCombination

Classe que lida com todas as combinações possíveis de cada jogada. Possue os métodos que validam se uma determinada combinação é válida e também se uma combinação é maior ou menor que a outra numa comparação.

Deck

Classe que cria uma abstração de um baralho, contendo uma carta de cada valor para cada naipe, o método padrão *get*, além de um método para realizar o embaralhamento das cartas do baralho.

BigTwo

Classe que cuida dos detalhes para inicio do jogo. Contém métodos para a criação dos jogadores, distribuição das cartas entre eles, verifica se a primeira jogada contém o 3 de ouros, cuida da ordem do jogo, quem joga depois de quem, além de *getters* e *setters* padrões.

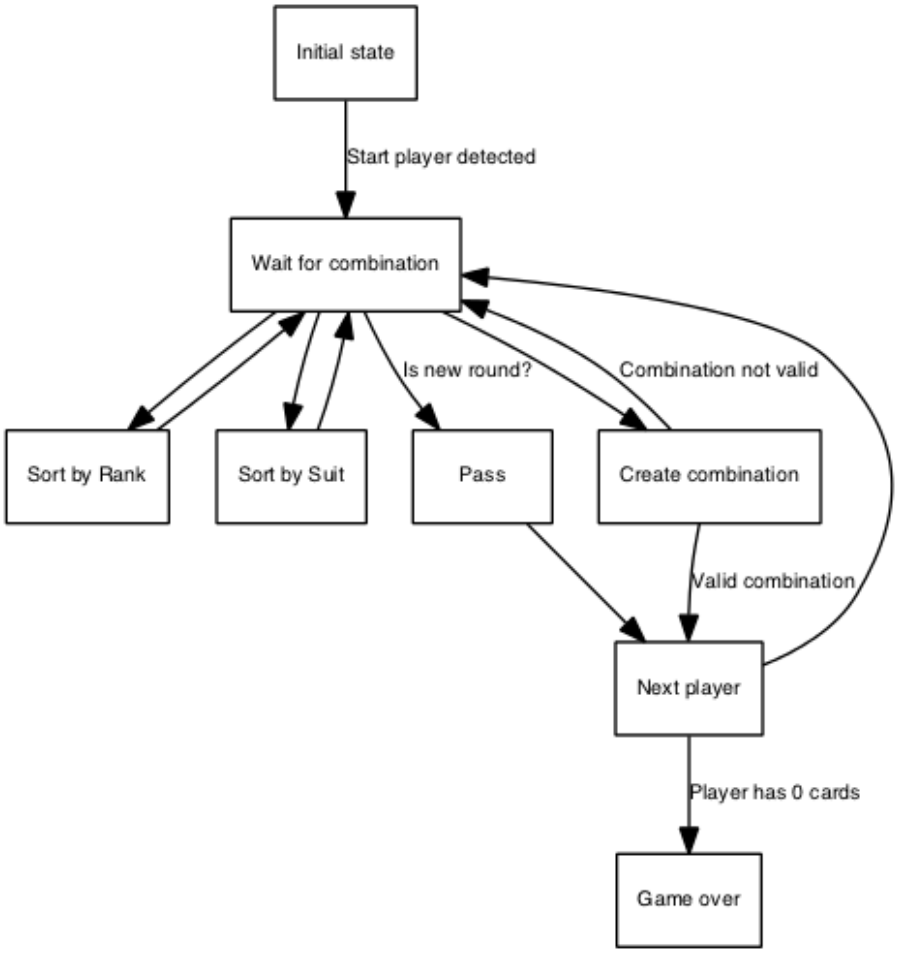

Figura 2. Diagrama de atividades

Player

Classe que representa o jogador. Ele possui uma mão de cartas no jogo, um nome próprio e possui método que pode ordenar suas cartas por naipe ou por valor.

CreateCombinationState

Classe que cuida das mudanças de estado do jogo. Possue métodos que lidam com o fluxo do jogo, sobre a montagem das combinações para as jogadas e se as combinações são válidas para modificar o estado do jogo.

EndOfGameState

Classe que lida com o final do jogo. Realiza a contagem dos pontos e indica o vencedor.

ListActionsState

Classe que lida com os comandos do jogo, quais ações podem ser tomadas para que o jogo continue.

ListStartRoundActionsState

Classe que lida com os comandos da primeira rodada do jogo.

NextPlayerState
Classe que cuida do fluxo de jogadores, ela que determina qual o próximo jogador após cada jogada válida.

StartState
Classe que determina quem é o primeiro jogador da partida.

WaitCombinationState
Classe que cuida da espera por uma ação do jogador.

GameState
Interface que implementa o método de início do jogo.

5.1. Principais métodos implementados

Essa subseção irá listar os principais métodos implementados no sistema. Funções triviais não serão listadas, como *getters*, *setters* ou similares. Os métodos detalhados a seguir são os principais utilizados para realizar as comparações entre as cartas, além dos métodos que tratam dos estados possíveis do jogo.

5.1.1. Card

- *public boolean compare(Card that)*

  Descrição: Realiza a comparação entre duas cartas para determinar qual é maior.
  Parâmetros: Recebe uma carta que se quer comparar com a instância de carta em questão.
  Retorno: Retorna falso se a instância é menor que a carta passada como parametro e verdadeiro se for maior.

5.1.2. CardCombination

- *public boolean checkCombination()*

  Descrição: Realiza a chamada do método adequado para verificar se a combinação é válida.
  Retorno: Retorna a chamada de método para cada tipo de combinação e falso para o caso de não ser válida.

- *private boolean checkTwoCards()*

  Descrição: Verifica se a combinação é válida como par.
  Retorno: Retorna verdadeiro se for um par válido e falso se não for um par válido.

- *private boolean checkThreeCards()*

  Descrição: Verifica se a combinação é válida como trinca.
  Retorno: Retorna verdadeiro se for uma trinca válida e falso se não for uma trinca válida.

- *private boolean checkFiveCards()*

  Descrição: Realiza a chamada do método adequado para verificar se a combinação de cinco cartas é válida.
  Retorno: Retorna a chamada de método para cada tipo de combinação de cinco cartas e falso para o caso de não ser válida.

- *private boolean checkStraight()*

  Descrição: Verifica se a combinação é do tipo sequência.
  Retorno: Retorna verdadeiro se for uma sequência válida e falso se não for uma sequência válida.

- *private boolean checkFlush()*

  Descrição: Verifica se a combinação é do tipo flush.
  Retorno: Retorna verdadeiro se for um flush válido e falso se não for um flush válido.

- *private boolean checkFullHouse()*

  Descrição: Verifica se a combinação é do tipo full house.
  Retorno: Retorna verdadeiro se for um full house válido e falso se não for um full house válido.

- *private boolean checkFour()*

  Descrição: Verifica se a combinação é do tipo four.
  Retorno: Retorna verdadeiro se for um four válido e falso se não for um four válido.

- *private boolean checkThreeOfDiamonds()*

  Descrição: Verifica se na combinação existe o 3 de ouros.
  Retorno: Retorna verdadeiro se existir o 3 de ouros na combinação e falso se não existir.

- *public boolean checkFirstCombination()*

  Descrição: Verifica se a primeira combinação do jogo é válida e se ela possui o 3 de ouros.
  Retorno: Retorna verdadeiro se for válido e existir o 3 de ouros na combinação e falso se algum dos dois for falso.

- *private void sortByRank()*

  Descrição: Realiza a ordenação das cartas da combinação por valor.

- *public void sortBySuit()*

  Descrição: Realiza a ordenação das cartas da combinação por naipe.

- *public boolean isHigherThan(CardCombination that)*

  Descrição: Realiza a comparação entre duas combinações de cartas para determinar qual é maior.
  Parâmetros: Recebe uma combinação que se quer comparar com a instância de combinação em questão.
  Retorno: Retorna falso se a instância é menor que a combinação passada como parâmetro e verdadeiro se for maior.

5.1.3. Deck

- *public void shuffle()*

  Descrição: Realiza o embaralhamento das cartas utilizando uma função padrão do Java para lidar com Collections.

5.1.4. BigTwo

- *public void startGame()*

  Descrição: Chama os métodos que de fato inicia o jogo.

- *private void createPlayers()*

  Descrição: Realiza a criação dos 4 jogadores.

- *private void distributeCards()*

  Descrição: Realiza a distribuição das cartas para os jogadores, cada um com 13 cartas.

- *public void setNextPlayer()*

  Descrição: Determina qual o jogador da próxima rodada.

- *public void printCurrentCombination()*

  Descrição: Imprime na tela a combinação que está na mesa atualmente e que deve ser vencida para que se possa jogar.

- *public boolean gameOver()*

  Descrição: Verifica se algum jogador terminou suas cartas, imprime que o jogo acabou.
  Retorno: Retorna verdadeiro se o algum jogador terminou suas carta e falso caso contrário.

- *public void updateLowestHand(Player p)*

  Descrição: Atualiza para cada jogador para sempre ficar armazenado qual jogador tem o menor número de cartas do jogo.
  Parâmetro: Recebe como parâmetro o jogador que acabou de jogar.

5.1.5. CreateCombinationState

- *public void doAction(BigTwo game)*

  Descrição: Realiza o controle da criação de uma combinação para ser jogada.
  Parâmetro: Recebe como parâmetro o próprio jogo.

- *public void listHand(BigTwo game)*

  Descrição: Realiza a impressão das opções de cartas e um índice de escolhas.
  Parâmetro: Recebe como parâmetro o próprio jogo.

- *public void playCombination(BigTwo game)*

  Descrição: Realiza o controle da jogada em si, chamando os métodos para validar a jogada.
  Parâmetro: Recebe como parâmetro o próprio jogo.

- *public boolean createCombination()*

  Descrição: Realiza o controle da formação da combinação, ao selecionar carta por carta.
  Parâmetro: Recebe como parâmetro o próprio jogo.
  Retorno: Retorna verdadeiro se a combinação foi criada com sucesso e falso caso contrário.

- *public boolean setSizeOfCombination()*

  Descrição: Determina o tamanho da combinação a ser formada em seguida.
  Retorno: Retorna verdadeiro se o tamanho da combinação for válida e falso caso contrário.

- *public boolean confirmCombination()*

  Descrição: Realiza o controle da validação da jogada, verificando todos os detalhes para a jogada ser válida.
  Retorno: Retorna verdadeiro se a combinação for jogada com sucesso e falso caso contrário.

- *public void checkCombination(BigTwo game)*

  Descrição: Verifica se a combinação é válida, sobretudo se for a primeira rodada do jogo.
  Parâmetro: Recebe como parâmetro o próprio jogo.

- *public void removeCombinationFromHand(BigTwo game)*

  Descrição: Remove as cartas da combinação jogada da mão do jogador.
  Parâmetro: Recebe como parâmetro o próprio jogo.

5.1.6. EndOfGameState

- *public void doAction(BigTwo game)*

  Descrição: Imprime que o jogo acabou e quem foi o vencedor.
  Parâmetro: Recebe como parâmetro o próprio jogo.

5.1.7. ListActionsState

- *public void doAction(BigTwo game)*

  Descrição: Realiza o controle da ação de escolha do comando.
  Parâmetro: Recebe como parâmetro o próprio jogo.

- *public int waitingForInput()*

  Descrição: Espera pelo comando do usuário.
  Retorno: Retorna o inteiro que representa o comando escolhido.

- *public void listActions()*

  Descrição: Lista os comandos possíveis.

- *public GameState nextState(int action, BigTwo game)*

  Descrição: Realiza o controle do estado, modificando ele de acordo com o comando.
  Parâmetro: Recebe como parâmetro o próprio jogo e o inteiro com o comando.
  Retorno: Retorna o estado do jogo.

5.1.8. ListStartRoundActionsState

- *public void listActions()*

  Descrição: Lista os comandos possíveis.

- *public GameState nextState(int action, BigTwo game)*

  Descrição: Realiza o controle do estado, modificando ele de acordo com o comando.
  Parâmetro: Recebe como parâmetro o próprio jogo e o inteiro com o comando.
  Retorno: Retorna o estado do jogo.

5.1.9. NextPlayerState

- *public void doAction(BigTwo game)*

  Descrição: Realiza a mudança do jogador atual depois de uma jogada.
  Parâmetro: Recebe como parâmetro o próprio jogo.

5.1.10. StartState

- *public void doAction(BigTwo game)*

  Descrição: Realiza o controle do inicio do jogo, imprimindo o primeiro jogador.
  Parâmetro: Recebe como parâmetro o próprio jogo.

5.1.11. WaitCombinationState

- *public void doAction(BigTwo game)*

  Descrição: Realiza a espera pela próxima ação do próximo jogador.
  Parâmetro: Recebe como parâmetro o próprio jogo.

6. Testes

Para esse trabalho, os testes foram os mais legais possíveis, pois testamos o jogo jogando! Não criamos nenhum teste automatizado, mas criamos uma planilha de BUGs para serem resolvidos. Para fins de demonstração, imprimimos uma rodada do jogo para ilustrar como ficou a interação do jogador.

6.1. Listagem da saída

```
Starting game...
Shuffling the deck...
Creating players...
Distributing cards...
Player 2's plays first.
Waiting for Player 2's action...
Player 2 choose your action:
1) Sort hand by suit
2) Sort hand by rank
3) Create combination of cards
Selection your action...
2
Sorting hand by rank...
Current hand...
THREE of SPADES
THREE of DIAMONDS
FOUR of HEARTS
SEVEN of CLUBS
SEVEN of SPADES
EIGHT of SPADES
EIGHT of CLUBS
TEN of CLUBS
JACK of DIAMONDS
ACE of CLUBS
ACE of HEARTS
ACE of DIAMONDS
TWO of SPADES
Player 2 choose your action:
1) Sort hand by suit
2) Sort hand by rank
3) Create combination of cards
```

```
Selection your action...
3
Choices of cards...
[ 1 ] THREE of SPADES
[ 2 ] THREE of DIAMONDS
[ 3 ] FOUR of HEARTS
[ 4 ] SEVEN of CLUBS
[ 5 ] SEVEN of SPADES
[ 6 ] EIGHT of SPADES
[ 7 ] EIGHT of CLUBS
[ 8 ] TEN of CLUBS
[ 9 ] JACK of DIAMONDS
[ 10 ] ACE of CLUBS
[ 11 ] ACE of HEARTS
[ 12 ] ACE of DIAMONDS
[ 13 ] TWO of SPADES
How many cards will you combination have? (1, 2, 3 or 5)
(press any letter to go back to actions)
2
Enter next card:
1
Enter next card:
2
Pair
THREE of DIAMONDS
THREE of SPADES
Play this combination? (y/n)
y
Current hand...
FOUR of HEARTS
SEVEN of CLUBS
SEVEN of SPADES
EIGHT of SPADES
EIGHT of CLUBS
TEN of CLUBS
JACK of DIAMONDS
ACE of CLUBS
ACE of HEARTS
ACE of DIAMONDS
TWO of SPADES
--------------------------------
Waiting for Player 3's action...
Player 3 choose your action:
1) Sort hand by suit
2) Sort hand by rank
3) Create combination of cards
```

```
4) Pass
Selection your action...
```

7. Conclusão

O trabalho prático foi prazeroso de ser realizado. Os conceitos de modelagem de classe aprendidos em sala foram bastante reforçados quando nos deparamos com um problema complexo como este.

Foi necessário lidar com o desafio de não ter uma especificação detalhada, o que traz uma série de dificuldades e decições complicadas de serem tomadas. Apesar disso, foi muito interessante e acreditamos que a escolha de um jogo como tema livre proporcionou uma motivação a mais para os alunos do que se fosse um assunto menos divertido.

Referências

BigTwo. Página do wikipédia sobre o big two. http://en.wikipedia.org/wiki/Big_Two.

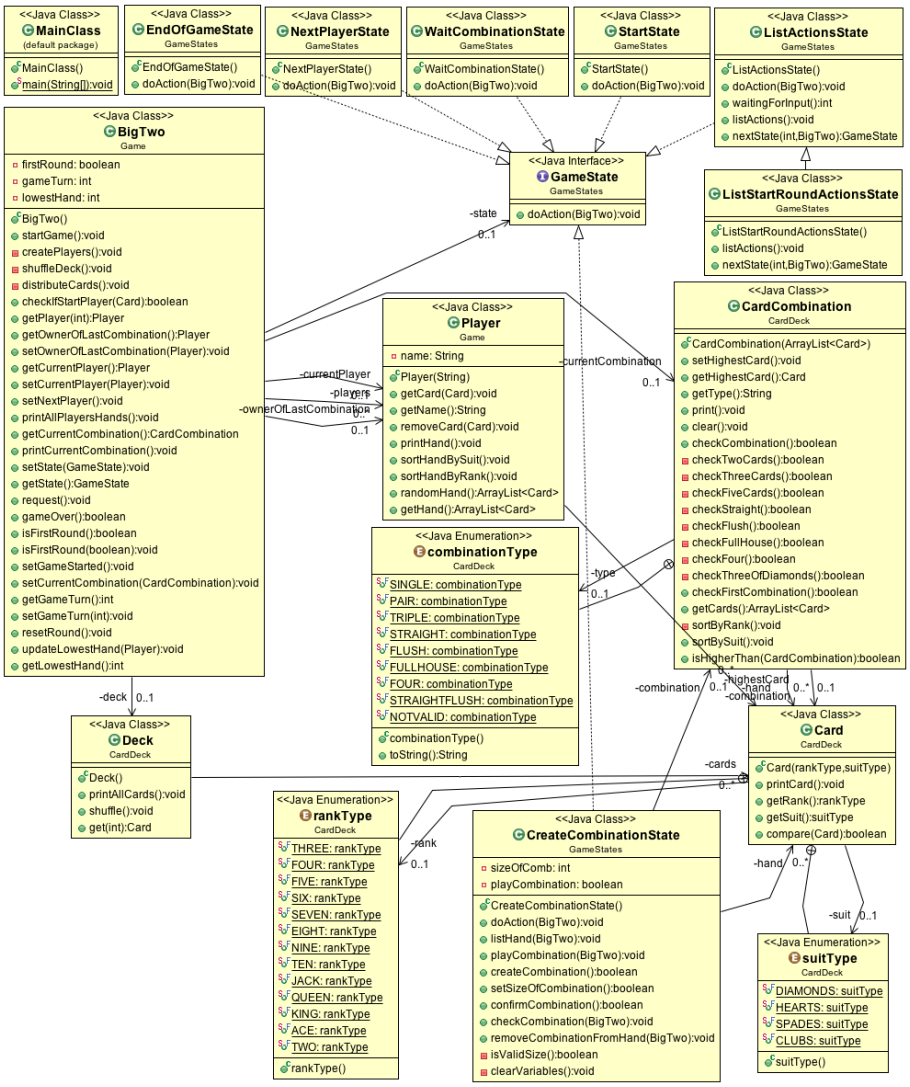

Figura 3. Diagrama de classes completo