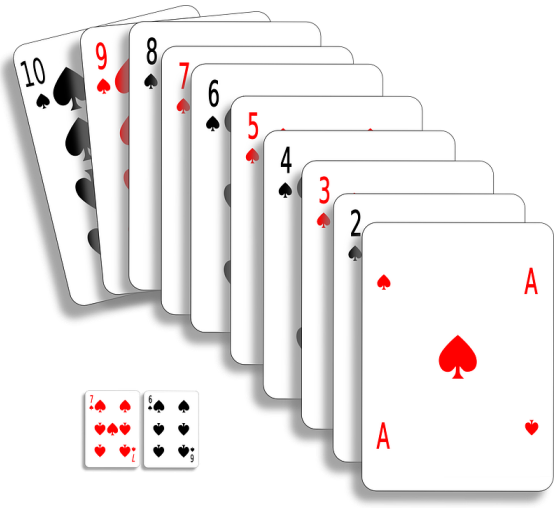
Modelação formal do jogo Russian Bank em VDM++



Mestrado Integrado em Engenharia Informática e Computação

4º ano | 1º semestre | MFES

Trabalho desenvolvido por:

Pedro Vieira Lamares Martins – 201005053 - ei10096@fe.up.pt

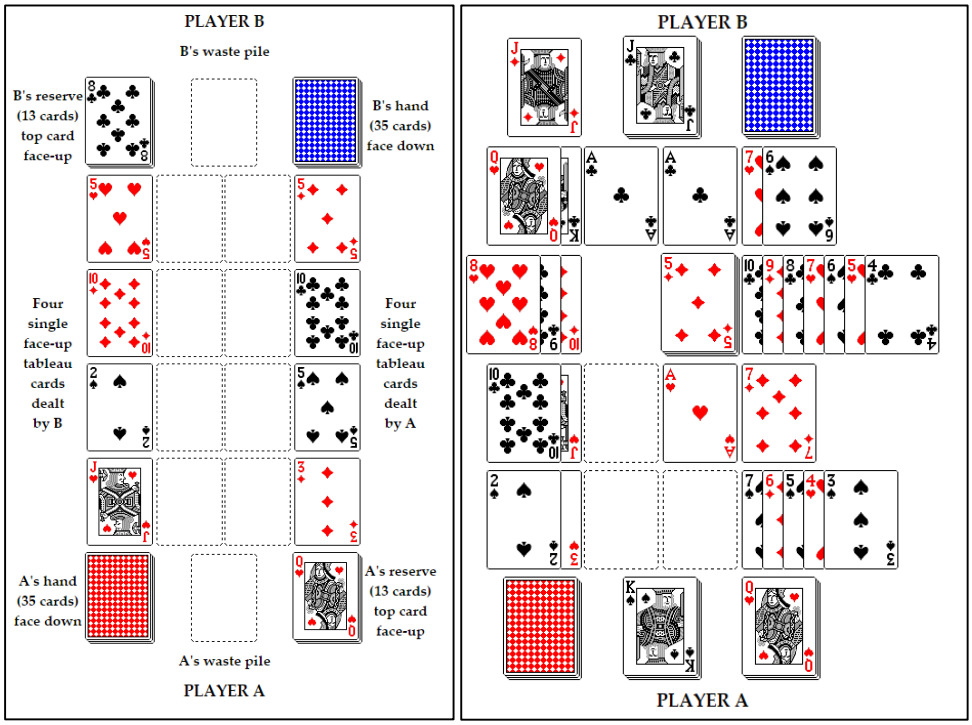
Filipe Diogo Soares Eiras – 201103055 – ei11087@fe.up.pt

14 de Dezembro de 2015

Índice

Descrição do sistema

Descrição informal do sistema



O objetivo deste projeto é formalizar um modelo baseado no jogo de cartas Russian Bank. Em cima, apresentam-se duas imagens do jogo: a primeira representa o estado inicial do jogo, sendo atribuído a cada jogador um baralho de cartas. Cada jogador vai dividir esse seu baralho em dois montes, a 'Hand' (ou mão), e a 'Reserve' (ou reserva), e colocar quatro cartas em cada casa da 'tableau' (chamadas simplesmente de casas). Existe ainda uma outra pilha, chamada de 'waste', ou lixo, que inicialmente está vazia. Além das oito casas da 'tableau', existem ainda oito casas no centro sem cartas inicialmente, chamadas de 'foundation' (ou fundações). Tanto as casas como as fundações são propriedade útil de ambos os jogadores. A figura 2 demonstra um jogo já num estado mais avançado:

- As cartas podem ser colocadas nas fundações se forem do mesmo naipe e por ordem ascendente, A23456789TJQK. Assim, por exemplo, um 2 de copas só pode ser jogado em cima de um A de copas.
- As cartas podem ser colocadas casas do 'tableau' de dentro para fora, alternando a cor do naipe, por ordem decrescente, KQJT987654321. Assim, por exemplo, um 8 de copas só pode ser colocado por cima de um 9 de espadas ou um 9 de paus.

As demais regras do jogo são as seguintes:

- Um jogador deve poder verificar se é o primeiro a poder jogar. O primeiro turno pertence ao jogador com a carta no topo da reserva de menor valor. Assim, por exemplo, caso o jogador A tenha um 8 de copas, e o jogador B um 7 de espadas, o primeiro turno pertence ao jogador B. Caso as cartas sejam iguais, a mesma regra aplica-se para as cartas das casas (começando pelas mais perto de cada jogador).

- No início do jogo estão disponíveis as seguintes cartas: a carta de topo na reserva (a que está virada para cima), e a carta em cada casa do tableau.
- Sempre que for virada uma carta da reserva, deve ser virada a carta imediatamente a seguir.
- Uma vez que a reserva fique sem nenhuma carta, não é possível colocar outra carta nessa reserva para o resto do jogo.
- Quando não for mais possível colocar cartas da reserva, o jogador deve usar as cartas da sua mão, virando uma a uma. Caso não seja possível fazer mais nenhum movimento, a carta da mão que foi virada deve ser colocada no lixo, virada para cima.
- Existem dois movimentos obrigatórios: quando uma carta da reserva pode ser movida para uma fundação, ou uma quando uma carta da mão pode ser movida para uma fundação.
- Se não houver mais nenhum movimento obrigatório, o jogador pode fazer movimentos secundários: colocar uma carta do mesmo naipe com o valor imediatamente acima ou imediatamente abaixo na reserva do oponente (por exemplo, se a carta de topo da reserva do oponente for um 4 de copas, o jogador pode lá colocar um 3 de copas ou um 5 de copas), ou colocar uma carta no lixo do oponente (seguindo as mesmas regras).
- Quando não há mais nenhum movimento disponível, obrigatório ou secundário, acaba o turno do jogador. Caso o turno seja trocado e o jogador mesmo assim não tenha nenhuma carta para jogar, deve tornar as cartas do seu lixo na sua nova mão.

Por motivos de simplificação, uma vez que a quantidade de movimentos possíveis, e portanto, a quantidade de testes a fazer sobre esses movimentos, seria disparatadamente grande (largamente ultrapassando o limite de 10 páginas imposto para grupos de duas pessoas), houve um foco da parte do grupo em concentrar os seus esforços para lidar com os casos mais abstratos do caso, e em conseguir cobrir todas as estruturas da linguagem VDM++. Sendo este o objetivo, encontra-se em baixo a lista de requisitos a implementar no nosso projeto.

Lista de requisitos

Para ir de encontro às regras do jogo, e para o modelo poder cumprir essas mesmas regras, definiram-se os seguintes requisitos:

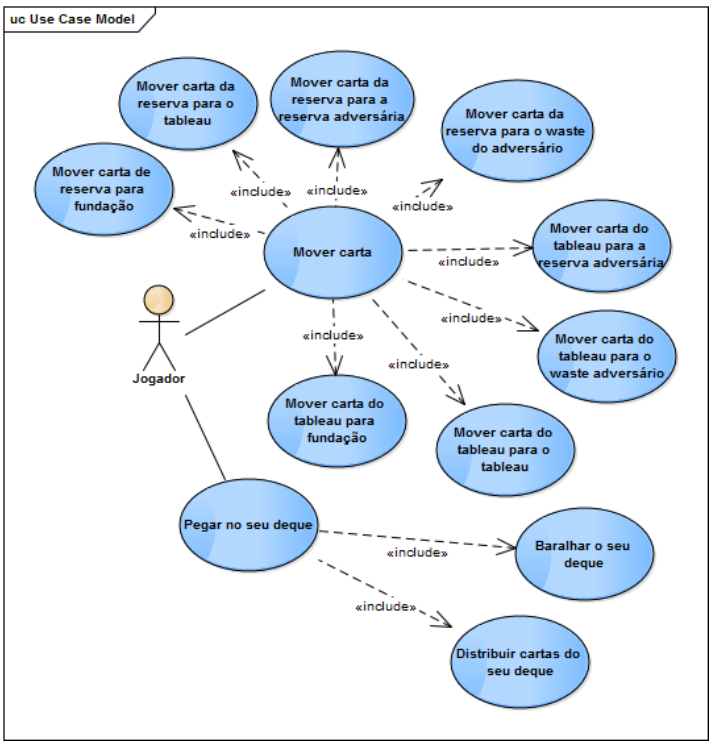
ID	Prioridade	Descrição
R1	Obrigatório	O jogador deve poder baralhar as suas cartas, dividi-las na sua reserva e na sua mão, e deve poder ainda colocar 4 cartas em 4 casas verticais do tableau.
R2	Obrigatório	O jogador, no seu turno, deve poder verificar se a sua carta de topo da reserva pode ser colocada numa fundação. Caso consiga, deve poder deslocá-la para lá, e levantar a sua carta do topo da reserva.
R3	Obrigatório	O jogador, no seu turno, deve poder verificar se alguma das cartas do tableau pode ser colocada numa fundação. Caso consiga, deve poder deslocar essa carta para lá.

R4	Obrigatório	O jogador, quando não pode mexer as cartas da sua reserva, deve poder mexer as cartas da sua mão, levantando a do topo. Deve colocá-la na waste caso mesmo assim a mesma possa não ser movida, ficando à espera da jogada do adversário.
R5	Obrigatório	O jogador deve poder mover cartas da sua reserva para o tableau.
R6	Obrigatório	O jogador deve poder mover cartas para a reserva e/ou para o waste adversário, se o mesmo não tiver mais nenhuma jogada disponível.
R7	Obrigatório	O jogador deve poder mover cartas, livremente, do seu tableau para a fundação, para outro tableau ou para a reserva/waste do adversário.
R8	Obrigatório	O jogador deve poder verificar quem já ganhou e com que pontuação.
R9	Opcional	O jogador deve poder mover as suas cartas do waste quando já não tem mais nenhuma carta disponível. Estas cartas do waste formarão a sua nova mão.
R10	Opcional	O jogador pode parar o jogo ao jogador adversário sempre que verificar que este não está a fazer uma jogada correta.

Os requisitos obrigatórios são devidamente implementados, sendo verificada a sua validade através de testes (que serão, também, descritos neste relatório). Os requisitos opcionais representam variáveis de jogo mais avançadas, com regras recursivas e dependentes umas das outras, que poderiam ser implementados num projeto futuro, de amplitude maior.

Modelo visual em UML

Casos de uso



Para cada caso de uso, o nosso projeto apresenta um teste para verificar a sua validade. Apresenta-se em baixo a descrição básica do que se pretende. De notar que, no caso dos vários movimentos possíveis de cartas de um lado para o outro, sujeitos a várias regras diferentes, não há necessidade de voltar a referir as mesmas (já descritas na descrição informal do problema). Esta análise inclui ainda as pre-condições, as pós-condições e os passos necessários para cada um dos casos de uso.

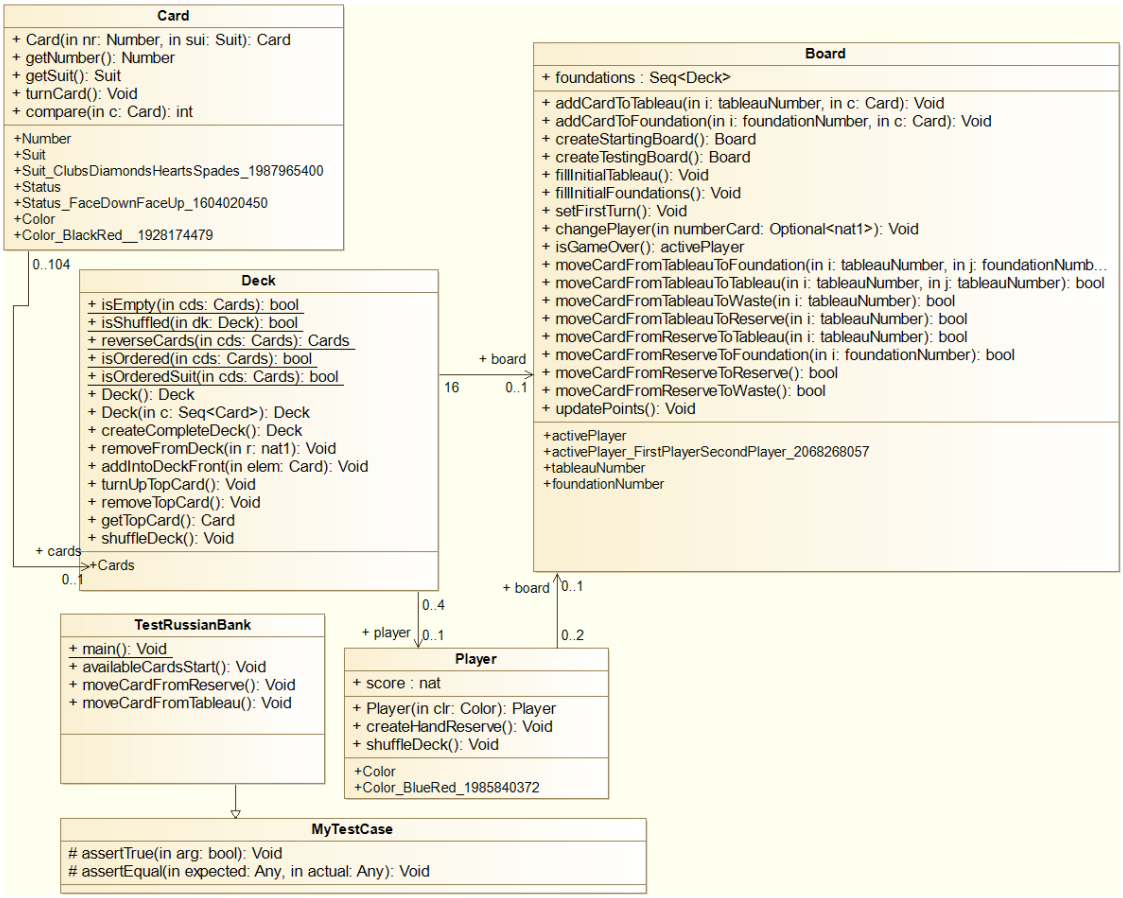
Cenário	**Baralhar o deque**
Descrição	Tal como o nome indica, o jogador deve poder baralhar o seu deque, de forma a que o mesmo fique pronto para jogar.
Pre-condições	O deque tem que estar ordenado.
Pós-condições	O deque tem que estar disposto de forma aleatória.
Passos	1º Cria um deque. 2º Escolhe uma carta aleatória, colocando-a noutro monte. 3º Faz isto até terminar o deque original.

Cenário	**Distribuir cartas do seu deque**
Descrição	O jogador distribuí as suas cartas para si mesmo (13 cartas para a reserva e 39 para a mão), colocando ainda outras quatro em quatro lugares do tableau (resultando em 35 cartas na mão).
Pre-condições	A mão e a reserva têm que estar vazias.
Pós-condições	A mão e a reserva têm que ter 35 e 13 cartas, respetivamente.
Steps	1º Retira 13 cartas do deque para a reserva. 2º Retira 39 cartas do deque para a mão. 3º Retira 4 cartas da mão para o tableau.
Notas	Este caso de uso é coberto por duas operações diferentes.

Cenário	**Mover carta de...para...**
Descrição	As várias funções de movimento, sujeitas a regras diferentes.
Pre-condições	Variam. Por exemplo, no caso do movimento da reserva para o tableau, uma das pre-condições seria identificar se a reserva não está vazia.
Pós-condições	Variam. Por exemplo, no caso anteriormente descrito, a pós-condição é o tableau ficar ordenado após inserção da carta (pelas regras do jogo).
Steps	Variam. Por exemplo, no caso anteriormente descrito: 1º Verifica o turno, podendo apenas jogar caso seja o seu turno. 2º Verifica, depois, se a jogada é inválida ou não. 3º Caso seja inválida, retorna 'false' e a carta não é movida. 4º Caso contrário, retorna 'true' e a carta é, então, movida.

Diagrama de classes

O nosso projeto contém quatro classes principais e uma classe de testes. O diagrama de classes foi gerado automaticamente usando a aplicação Modelio 3.3.



Classe	Descrição
Card	Define uma carta do jogo, sendo os atributos principais um identificador da mesma (através do número e do naipe), a cor da mesma e o estado atual (virado para baixo ou para cima).
Deck	Define um deque, ou um conjunto de cartas. Faz operações sobre essas mesmas cartas no contexto do jogo, tais como remover a do topo, vira-la para cima ou misturar todas as cartas no momento.
Player	Define um jogador, que possuí vários conjuntos de cartas (a mão, o waste e a reserva), o seu score e a cor que lhe está associada.
Board	Classe central, que é responsável por unificar as restantes, e criar as fundações e as casas do tableau que serão usadas por ambos os jogadores. Cobre e controla a validade de todos os movimentos, segundo as regras do jogo.
MyTestCase	Superclasse para os testes, disponibilizada pelo professor.
TestRussianBank	Define os testes a serem usados para verificar a integridade do nosso modelo formal do jogo.

Modelo formal em VDM++

A seguir apresenta-se toda a implementação do código em VDM++ para criar o modelo formal especificado em cima.

Classe Card

```
  class Card
/*
      Class that represents a single card
      Each card has a value, from 1 to 13, a color and its status
      (if it's faced down or not)
*/
      types
            public Number = nat1 -- 1 - Ace, 11 - Jack, 12 - Queen, 13
- King
                  inv cardValue == cardValue in set {1, 2, 3, 4, 5, 6,
7, 8, 9, 10, 11, 12, 13};
            public Suit = <Clubs> | <Hearts> | <Spades> | <Diamonds>;
            public Status = <Face_Up> | <Face_Down>;
            public SuitColor = map Suit to Player`Color;

      instance variables
            public number: Number;
            public suit: Suit;
            public status: Status := <Face_Down>;
            public color: Player`Color;
            public suitColors: SuitColor := {<Clubs> |-> <Black>,
<Spades> |-> <Black>, <Diamonds> |-> <Red>, <Hearts> |-> <Red>};

      operations
            -- constructor
            public Card: Number * Suit ==> Card
                  Card(nr, sui) == (
                  number := nr;
                  suit := sui;
                  color := suitColors(sui);
                  return self
            );

            -- gets
            public getNumber: () ==> Number
                  getNumber() ==
                        return number;

            public getSuit: () ==> Suit
                  getSuit() ==
                        return suit;

            -- turn a a card for movement purposes.
            public turnCard: () ==> ()
                  turnCard() ==
                        if (status = <Face_Down>) then
                              status := <Face_Up>
                        else
                              status := <Face_Down>;

            -- compares two cards, returning 1 if higher, -1 if lower
or 0 if equal, acording to the game rules
            public compare: Card ==> int
                  compare(c) == (
                  if (number > c.getNumber()) then
                        return 1
```

```
                elseif (number < c.getNumber()) then
                        return -1
                else
                        return 0;
        );

end Card
```

Classe Deck

```
class Deck
/*
        Class that represents a deck of cards, having
        some utilitary functions and operations to work on it
*/
        types
                public Cards = seq of Card; -- a set of cards represent a
deck

        instance variables
                public cards: Cards;
                public defaultCards: Cards := [new Card(1, <Clubs>), new
Card(2, <Clubs>), new Card(3, <Clubs>), new Card(4, <Clubs>), new
Card(5, <Clubs>),
                                new Card(6, <Clubs>), new Card(7, <Clubs>), new
Card(8, <Clubs>), new Card(9, <Clubs>), new Card(10, <Clubs>), new
Card(11, <Clubs>),
                                new Card(12, <Clubs>), new Card(13, <Clubs>),
new Card(1, <Hearts>), new Card(2, <Hearts>), new Card(3, <Hearts>),
new Card(4, <Hearts>),
                                new Card(5, <Hearts>), new Card(6, <Hearts>),
new Card(7, <Hearts>), new Card(8, <Hearts>), new Card(9, <Hearts>),
new Card(10, <Hearts>),
                                new Card(11, <Hearts>), new Card(12, <Hearts>),
new Card(13, <Hearts>), new Card(1, <Spades>), new Card(2, <Spades>),
new Card(3, <Spades>),
                                new Card(4, <Spades>), new Card(5, <Spades>),
new Card(6, <Spades>), new Card(7, <Spades>), new Card(8, <Spades>),
new Card(9, <Spades>),
                                new Card(10, <Spades>), new Card(11, <Spades>),
new Card(12, <Spades>), new Card(13, <Spades>), new Card(1,
<Diamonds>),
                                new Card(2, <Diamonds>), new Card(3,
<Diamonds>), new Card(4, <Diamonds>), new Card(5, <Diamonds>), new
Card(6, <Diamonds>),
                                new Card(7, <Diamonds>), new Card(8,
<Diamonds>), new Card(9, <Diamonds>), new Card(10, <Diamonds>), new
Card(11, <Diamonds>),
                                new Card(12, <Diamonds>), new Card(13,
<Diamonds>)];

        functions
                -- returns true if the deck is empty
                -- returns false otherwise.
                public isEmpty: Cards -> bool
                        isEmpty(cds) == (
                                if( len cds > 0)
                                        then false
                                else
                                        true
```

```
		);

	-- function that returns true if the deck is shuffled
	-- returns false otherwise.
	public isShuffled: Deck -> bool
		isShuffled(dk) == (
			if dk.cards = dk.defaultCards then
				false
			else
				true
		);
	-- reverse a seq of cards
	public reverseCards: Cards -> Cards
		reverseCards(cds) ==
			if(len cds = 1) then
				cds
			else
				reverseCards(tl cds) ^ [hd cds];

	-- check if a seq of cards is ordered sequentially by
number
	-- and alternately by color
	public isOrdered: Cards -> bool
			isOrdered(cds) ==
				let x = hd cds
				in let remainder = tl cds
					in if isEmpty(remainder) then true
						else let y = hd tl cds
							in if x.number <>
y.number - 1 or x.color = y.color then false
								else (
									if
isEmpty(tl cds) then true
										else
isOrdered(tl cds)
								)
	pre isEmpty(cds) = false;

	-- check if a seq of cards is ordered sequentially by
number
	-- and if it has the same suit
	public isOrderedSuit: Cards -> bool
			isOrderedSuit(cds) ==
				let x = hd cds
				in let remainder = tl cds
					in if isEmpty(remainder) then true
						else let y = hd tl cds
							in if x.number <>
y.number - 1 and x.suit <> y.suit then false
								else (
									if
isEmpty(tl cds) then true
										else
isOrderedSuit(tl cds)
								)
	pre isEmpty(cds) = false;

operations
	-- default constructor.
	public Deck: () ==> Deck
		Deck() == (
```

```
                cards := [];
                return self
        );

        -- constructor that accepts cards.
        public Deck: seq of Card ==> Deck
                Deck(c) == (
                cards := c
        );

        -- create a complete deck with 52 cards.
        -- pre and post conditions mean it can only be called once.
        public createCompleteDeck: () ==> Deck
                createCompleteDeck() == (
                cards := defaultCards;

                return self;
                )
                pre cards = []
                post len cards = 52;

        -- operation to remove a card from a deck
        public removeFromDeck: nat1 ==> ()
                removeFromDeck(r) == (
                        dcl length:nat := len cards;
                        if (r = 1) then -- removing the first element
                                cards := tl cards
                        elseif (r = length) then -- removing the last
element
                                cards := (Deck`reverseCards(tl
Deck`reverseCards(cards)))
                        elseif (r = length - 1) then -- removing
second-to-last element
                                cards := (cards(1,..., r - 1) ^ [hd
Deck`reverseCards(cards)])
                        else -- other cases
                                cards := (cards(1,..., r - 1) ^ cards(r +
1,...,length))
                );

                -- operation that adds a card to the front of a deck
                public addIntoDeckFront: Card ==> ()
                        addIntoDeckFront(elem) == (
                                cards := [elem] ^ cards
                );

                --operation that turns up the top card of a deck
                public turnUpTopCard: () ==> ()
                        turnUpTopCard() == (
                        dcl crd:Card := hd cards;
                        crd.turnCard();
                        cards := [crd] ^ tl cards;
                );

                -- remove the top element from a deck
                public removeTopCard: () ==> ()
                        removeTopCard() == (
                                self.removeFromDeck(1);
                );

                -- returns the top card from the deck
```

```
		public getTopCard: () ==> Card
			getTopCard() == (
				return hd cards;
			);

		-- operation that shuffles the deck for subsequent
uses
		public shuffleDeck: () ==> ()
			shuffleDeck() == (
				dcl res:seq of Card := [];
				dcl lenlist:nat := len cards;
				dcl i:nat := 1;

				while i <= lenlist do (
					dcl n:nat := MATH`rand(len cards) +
1;

					res := res ^ [cards(n)];
					self.removeFromDeck(n);
					i := i + 1;
				);

				cards := res;
			);

end Deck
```

Classe Player

```
class Player
/*
	Class that represents a player
	Each player has initially one deck, that will be divided on hand
cards
	and reserve cards. They also have a color for identifying the
player
	and its cards and a waste, were will be placed cards
*/
	types
		public Color = <Red> | <Black>;

	instance variables
		public score: nat := 0;
		public color: Color;
		public fullDeck: Deck := new Deck();
		public reserve: Deck := new Deck();
		public waste: Deck := new Deck();
		public hand: Deck := new Deck();

	operations
		-- constructor
		public Player: Color ==> Player
			Player(clr) == (
			color := clr;
			fullDeck := fullDeck.createCompleteDeck();
			return self;
		);

		-- divides the personal deck in two piles, one the reserve
and the other the hand
		public createHandReserve: () ==> ()
```

```
                createHandReserve() == (
                dcl crds:Deck`Cards := fullDeck.cards;
                reserve.cards := (crds(1,..., 13)); -- create reserve
                hand.cards := (crds(14,...,52)); -- create hand
                reserve.turnUpTopCard()
            ) pre Deck`isEmpty(hand.cards) and
Deck`isEmpty(reserve.cards)
                post len hand.cards = 39 and len reserve.cards = 13;

            -- shuffles the personal deck for subsequent uses
            public shuffleDeck: () ==> ()
                shuffleDeck() ==
                    fullDeck.shuffleDeck();

  end Player
```

Classe Board

```
/*
      Class Board
      Responsible to unite every concept of the game.
      Creates both players, both player's decks, and join both of them
with the foundations
      and the tableau. Also creates the movement rules that need to be
validated, changes turn,
      verifies if is game over or not, and returns the points for each
player.
*/
class Board
      types
            public activePlayer = [<FirstPlayer> | <SecondPlayer>];
            public tableauNumber = nat1
                  inv numbersOfTableau == numbersOfTableau in set
{1,...,8};
            public foundationNumber = nat1
                  inv numbersOfFoundation == numbersOfFoundation in set
{1,...,8};


      instance variables
            -- reunites both players
            public firstPlayer: Player := new Player();
            public secondPlayer: Player := new Player();

            -- foundations and the tableau are common proprierty of
both players.
            public foundations: seq of Deck := [];
            public tableau: seq of Deck := [];

            -- this variable indicates the player whose turn is in
            public turn: activePlayer := nil;

      operations

            -- for testing purposes
            -- for some reason, without this we can't use the operation
addIntoDeckFront
            -- on the test class
            public addCardToTableau: tableauNumber * Card ==> ()
```

```
                addCardToTableau(i,c) == (
                        tableau(i).addIntoDeckFront(c);
                );
        -- for testing purposes
        -- for some reason, without this we can't use the operation
addIntoDeckFront
        -- on the test class
        public addCardToFoundation: foundationNumber * Card ==> ()
                addCardToFoundation(i,c) == (
                        foundations(i).addIntoDeckFront(c);
                );

/*
        -- creates the starting board.
        public createStartingBoard: () ==> Board
                createStartingBoard() == (

                -- create both players
                firstPlayer := new Player(<Red>);
                secondPlayer := new Player(<Blue>);
                -- shuffle the decks for each player
                firstPlayer.shuffleDeck();
                secondPlayer.shuffleDeck();
                -- creates the hands and reserve pile for each player
                firstPlayer.createHandReserve();
                secondPlayer.createHandReserve();
                -- fills the tableau with 4 cards dealt from each
play's hand
                fillInitialTableau();
                -- create the empty foundations
                fillInitialFoundations();
                -- see who's the first player to make a move (the one
with the lowest-ranking reserve card)
                setFirstTurn();

                return self;
                );
*/
                -- fills the initial tableau with the 8 cards from
each player
                public fillInitialTableau: () ==> ()
                        fillInitialTableau() == (
                                for i = 1 to 4 do (
                                        tableau(i) := new
Deck([firstPlayer.hand.getTopCard()]);
                                        tableau(i).turnUpTopCard();
                                        firstPlayer.hand.removeTopCard()
                                );
                                for i = 5 to 8 do (
                                        tableau(i) := new
Deck([secondPlayer.hand.getTopCard()]);
                                        tableau(i).turnUpTopCard();
                                        secondPlayer.hand.removeTopCard()
                                );
                )
                pre len tableau = 0
                post len tableau = 8;

                -- create initial empty foundations
                public fillInitialFoundations: () ==> ()
                        fillInitialFoundations() == (
```

```
                        for i = 1 to 8 do (
                            foundations(i) := new Deck();
                        );
                )
                pre len foundations = 0
                post len foundations = 8;

                -- set who should play in the first turn
                public setFirstTurn: () ==> ()
                      setFirstTurn() == (
                        dcl topReserve1:Card :=
firstPlayer.reserve.getTopCard();
                        dcl topReserve2:Card :=
secondPlayer.reserve.getTopCard();

                        if (topReserve1.compare(topReserve2) = -
1) then
                            turn := <FirstPlayer> -- if lower-
ranking reserve card, then player1 plays first
                        elseif (topReserve1.compare(topReserve2)
= 1) then
                            turn := <SecondPlayer> -- or the
contrary
                        elseif
(tableau(1).getTopCard().compare(tableau(5).getTopCard()) = -1) then
                            turn := <FirstPlayer> -- if equal,
then if lower-ranking first hand card, then player1 plays first
                        elseif
(tableau(1).getTopCard().compare(tableau(5).getTopCard()) = 1) then
                            turn := <SecondPlayer> -- or the
contrary
                        elseif
(tableau(2).getTopCard().compare(tableau(6).getTopCard()) = -1) then
                            turn := <FirstPlayer> -- if lower-
ranking second hand card, then player1 plays first...and so on
                        elseif
(tableau(2).getTopCard().compare(tableau(6).getTopCard()) = 1) then
                            turn := <SecondPlayer>
                        elseif
(tableau(3).getTopCard().compare(tableau(7).getTopCard()) = -1) then
                            turn := <FirstPlayer>
                        elseif
(tableau(3).getTopCard().compare(tableau(7).getTopCard()) = 1) then
                            turn := <SecondPlayer>
                        elseif
(tableau(4).getTopCard().compare(tableau(8).getTopCard()) = -1) then
                            turn := <FirstPlayer>
                        elseif
(tableau(4).getTopCard().compare(tableau(8).getTopCard()) = 1) then
                            turn := <SecondPlayer>
                )
                pre turn = nil
                post turn = <FirstPlayer> or turn = <SecondPlayer>;
/*
                -- change the turn to the other player, putting one
card
                public changePlayer: () ==> ()
                      changePlayer() == (
                            if (turn = <FirstPlayer>) then (
                                if not
Deck`isEmpty(firstPlayer.hand.cards) then (
```

```
                                        if not
Deck`isEmpty(firstPlayer.waste.cards) then (

      firstPlayer.waste.turnUpTopCard();
                                        );
                                        if(not
Deck`isEmpty(firstPlayer.hand.cards)) then (

      firstPlayer.waste.addIntoDeckFront(firstPlayer.hand.cards(1));

      firstPlayer.waste.turnUpTopCard();

      firstPlayer.hand.removeFromDeck(1);
                                              )
                                  );
                                  turn := <SecondPlayer>;
                                  )
                              else (
                                  if not
Deck`isEmpty(secondPlayer.hand.cards) then (
                                        if not
Deck`isEmpty(secondPlayer.waste.cards) then (

      secondPlayer.waste.turnUpTopCard();
                                        );
                                        if(not
Deck`isEmpty(secondPlayer.hand.cards)) then (

      secondPlayer.waste.addIntoDeckFront(secondPlayer.hand.cards(1));

      secondPlayer.waste.turnUpTopCard();

      secondPlayer.hand.removeFromDeck(1);
                                        )
                                  );
                                  turn := <FirstPlayer>;
                              );
                        ) pre turn = <FirstPlayer> or turn =
<SecondPlayer>;
*/
                  -- check if game is over
                  public isGameOver: () ==> activePlayer
                        isGameOver() == (
                        if Deck`isEmpty(firstPlayer.hand.cards)
                            and Deck`isEmpty(firstPlayer.reserve.cards)
                            and Deck`isEmpty(firstPlayer.waste.cards)
then return <FirstPlayer>
                        else if Deck`isEmpty(secondPlayer.hand.cards)
                            and Deck`isEmpty(secondPlayer.reserve.cards)
                            and Deck`isEmpty(secondPlayer.waste.cards)
then return <SecondPlayer>;
                        return nil;
                        );

                  -- move a card from one tableau to one foundation
                  public moveCardFromTableauToFoundation: tableauNumber
* foundationNumber ==> bool
                        moveCardFromTableauToFoundation(i,j) == (
                              if(Deck`isEmpty(foundations(j).cards))
then (
```

```
        if(tableau(i).getTopCard().getNumber() <> 1) then
                                                return false
                                            else

        foundations(j).addIntoDeckFront(tableau(i).getTopCard());
                                                tableau(i).removeTopCard();
                                                if not
Deck`isEmpty(tableau(i).cards) then

        tableau(i).turnUpTopCard();
                                                return true;
                                            )
                                        else

        if(tableau(i).getTopCard().getSuit() =
foundations(j).getTopCard().getSuit()
                                                and
tableau(i).getTopCard().getNumber() =
foundations(j).getTopCard().getNumber() + 1) then (

        foundations(j).turnUpTopCard();

        foundations(j).addIntoDeckFront(tableau(i).getTopCard());

        tableau(i).removeTopCard();
                                                    if not
Deck`isEmpty(tableau(i).cards) then

        tableau(i).turnUpTopCard();
                                                    return true;
                                                )
                                            else return false;
                                    ) pre not Deck`isEmpty(tableau(i).cards)
                                        post (Deck`isEmpty(tableau(i).cards) or
Deck`isOrdered(tableau(i).cards)) and
Deck`isOrderedSuit(foundations(j).cards);

                    -- move a card from one tableau to another tableau
                    public moveCardFromTableauToTableau: tableauNumber *
tableauNumber ==> bool
                            moveCardFromTableauToTableau(i,j) == (
                                if(Deck`isEmpty(tableau(i).cards)) then (

        if(tableau(i).getTopCard().getNumber() <> 1) then
                                                return false
                                            else

        tableau(j).addIntoDeckFront(tableau(i).getTopCard());
                                                tableau(i).removeTopCard();
                                                if not
Deck`isEmpty(tableau(i).cards) then

        tableau(i).turnUpTopCard();
                                                return true;
                                            )
                                        else
                                            if(tableau(i).getTopCard().color <>
tableau(j).getTopCard().color
```

```
                                                and
tableau(i).getTopCard().getNumber() =
tableau(j).getTopCard().getNumber() - 1) then (

      tableau(j).turnUpTopCard();

      tableau(j).addIntoDeckFront(tableau(i).getTopCard());

      tableau(i).removeTopCard();
                                                if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();
                                                return true;
                                          )
                                    else return false;
                          ) pre not Deck`isEmpty(tableau(i).cards)
                                    post Deck`isOrdered(tableau(i).cards) and
Deck`isOrderedSuit(tableau(j).cards);

                    -- move a card from one tableau to other player waste
                    public moveCardFromTableauToWaste : tableauNumber ==>
bool
                          moveCardFromTableauToWaste(i) == (
                                if turn = <FirstPlayer> then (

      if(tableau(i).getTopCard().getSuit() <>
secondPlayer.waste.getTopCard().getSuit()
                                                and
tableau(i).getTopCard().getNumber() <>
secondPlayer.waste.getTopCard().getNumber() + 1
                                                and
tableau(i).getTopCard().getNumber() <>
secondPlayer.waste.getTopCard().getNumber() - 1) then
                                                return false
                                          else (
                                                if not
Deck`isEmpty(secondPlayer.waste.cards) then

      secondPlayer.waste.turnUpTopCard();

      secondPlayer.waste.addIntoDeckFront(tableau(i).getTopCard());
                                                tableau(i).removeTopCard();
                                                if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();
                                                return true;
                                          )
                                    ) else (

      if(tableau(i).getTopCard().getSuit() <>
firstPlayer.waste.getTopCard().getSuit()
                                                and
tableau(i).getTopCard().getNumber() <>
firstPlayer.waste.getTopCard().getNumber() + 1
                                                and
tableau(i).getTopCard().getNumber() <>
firstPlayer.waste.getTopCard().getNumber() - 1) then
                                                return false
                                          else (
```

```
                                        if not
Deck`isEmpty(firstPlayer.waste.cards) then

      firstPlayer.waste.turnUpTopCard();

      firstPlayer.waste.addIntoDeckFront(tableau(i).getTopCard());
                                        tableau(i).removeTopCard();
                                        if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();
                                        return true;
                                    )
                                )
                            ) pre not Deck`isEmpty(tableau(i).cards)
                                post Deck`isEmpty(tableau(i).cards) or
Deck`isOrdered(tableau(i).cards);

                    -- move a card from one tableau to other player waste

                    public moveCardFromTableauToReserve : tableauNumber
==> bool
                            moveCardFromTableauToReserve(i) == (
                                if turn = <FirstPlayer> then (

      if(tableau(i).getTopCard().getSuit() <>
secondPlayer.reserve.getTopCard().getSuit()
                                        and
tableau(i).getTopCard().getNumber() <>
secondPlayer.reserve.getTopCard().getNumber() + 1
                                        and
tableau(i).getTopCard().getNumber() <>
secondPlayer.reserve.getTopCard().getNumber() - 1) then
                                        return false
                                    else (
                                        if not
Deck`isEmpty(secondPlayer.reserve.cards) then

      secondPlayer.reserve.turnUpTopCard();

      secondPlayer.reserve.addIntoDeckFront(tableau(i).getTopCard());
                                        tableau(i).removeTopCard();
                                        if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();
                                        return true;
                                    )
                                ) else (

      if(tableau(i).getTopCard().getSuit() <>
firstPlayer.reserve.getTopCard().getSuit()
                                        and
tableau(i).getTopCard().getNumber() <>
firstPlayer.reserve.getTopCard().getNumber() + 1
                                        and
tableau(i).getTopCard().getNumber() <>
firstPlayer.reserve.getTopCard().getNumber() - 1) then
                                        return false
                                    else (
```

```
                                                    if not
Deck`isEmpty(firstPlayer.reserve.cards) then

      firstPlayer.reserve.turnUpTopCard();

      firstPlayer.reserve.addIntoDeckFront(tableau(i).getTopCard());
                                                    tableau(i).removeTopCard();
                                                    if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();
                                                    return true;
                                                )
                                            )
                                        ) pre not Deck`isEmpty(tableau(i).cards)
                                            post Deck`isEmpty(tableau(i).cards) or
Deck`isOrdered(tableau(i).cards);

                    -- move a card from players reserve to one tableau

                    public moveCardFromReserveToTableau : tableauNumber
==> bool
                            moveCardFromReserveToTableau(i) == (
                                if turn = <FirstPlayer> then (

      if(Deck`isEmpty(firstPlayer.reserve.cards) or
tableau(i).getTopCard().color = firstPlayer.reserve.getTopCard().color
                                                    or
tableau(i).getTopCard().number <=
firstPlayer.reserve.getTopCard().number) then
                                                    return false
                                                else (
                                                    if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();

      tableau(i).addIntoDeckFront(firstPlayer.reserve.getTopCard());

      firstPlayer.reserve.removeTopCard();
                                                    if not
Deck`isEmpty(firstPlayer.reserve.cards) then

      firstPlayer.reserve.turnUpTopCard();
                                                    return true;
                                                )
                                            ) else (

      if(Deck`isEmpty(secondPlayer.reserve.cards) or
tableau(i).getTopCard().color =
secondPlayer.reserve.getTopCard().color
                                                    or
tableau(i).getTopCard().number <=
secondPlayer.reserve.getTopCard().number) then
                                                    return false
                                                else (
                                                    if not
Deck`isEmpty(tableau(i).cards) then

      tableau(i).turnUpTopCard();
```

```
        tableau(i).addIntoDeckFront(secondPlayer.reserve.getTopCard());

        secondPlayer.reserve.removeTopCard();
                                                    if not
Deck`isEmpty(secondPlayer.reserve.cards) then

        secondPlayer.reserve.turnUpTopCard();
                                                    return true;
                                            )
                                    )
                            ) pre not Deck`isEmpty(tableau(i).cards)
                                    post Deck`isOrdered(tableau(i).cards);

                        -- move a card from players reserve to one foundation
                        public moveCardFromReserveToFoundation :
foundationNumber ==> bool
                                moveCardFromReserveToFoundation(i) == (
                                        if turn = <FirstPlayer> then (

        if(Deck`isEmpty(foundations(i).cards)) then (

        if(firstPlayer.reserve.getTopCard().number <> 1) then
                                                            return false
                                                    else (

        foundations(i).addIntoDeckFront(firstPlayer.reserve.getTopCard()
);

        firstPlayer.reserve.removeTopCard();
                                                            if not
Deck`isEmpty(firstPlayer.reserve.cards) then

        firstPlayer.reserve.turnUpTopCard();
                                                            return true;
                                                    );
                                            )
                                            else
if(foundations(i).getTopCard().getSuit() <>
firstPlayer.reserve.getTopCard().getSuit()
                                                    or
foundations(i).getTopCard().number <>
firstPlayer.reserve.getTopCard().number - 1) then
                                                            return false
                                            else (
                                                    if not
Deck`isEmpty(foundations(i).cards) then

        foundations(i).turnUpTopCard();

        foundations(i).addIntoDeckFront(firstPlayer.reserve.getTopCard()
);

        firstPlayer.reserve.removeTopCard();
                                                    if not
Deck`isEmpty(firstPlayer.reserve.cards) then

        firstPlayer.reserve.turnUpTopCard();
                                                    return true;
                                            )
                                    ) else (
```

```
      if(Deck`isEmpty(foundations(i).cards)) then (

      if(secondPlayer.reserve.getTopCard().number <> 1) then
                                                return false
                                          else (

      foundations(i).addIntoDeckFront(secondPlayer.reserve.getTopCard(
));

      secondPlayer.reserve.removeTopCard();
                                                if not
Deck`isEmpty(secondPlayer.reserve.cards) then

      secondPlayer.reserve.turnUpTopCard();
                                                return true;
                                          );
                                    )
                                    else
if(foundations(i).getTopCard().getSuit() <>
secondPlayer.reserve.getTopCard().getSuit()
                                          or
foundations(i).getTopCard().number <>
secondPlayer.reserve.getTopCard().number - 1) then
                                          return false
                                    else (
                                          if not
Deck`isEmpty(foundations(i).cards) then

      foundations(i).turnUpTopCard();

      foundations(i).addIntoDeckFront(secondPlayer.reserve.getTopCard(
));

      secondPlayer.reserve.removeTopCard();
                                          if not
Deck`isEmpty(secondPlayer.reserve.cards) then

      secondPlayer.reserve.turnUpTopCard();
                                          return true;
                                    )
                              )
                        )pre (not
Deck`isEmpty(firstPlayer.reserve.cards) and turn = <FirstPlayer>)
                                    or (not
Deck`isEmpty(secondPlayer.reserve.cards) and turn = <SecondPlayer>)
                        post Deck`isOrderedSuit(foundations(i).cards);

                  -- move a card from players reserve to opposing
reserve
                  public moveCardFromReserveToReserve : () ==> bool
                        moveCardFromReserveToReserve() == (
                              if turn = <FirstPlayer> then (

      if(firstPlayer.reserve.getTopCard().getSuit() <>
secondPlayer.reserve.getTopCard().getSuit()
                                    or
(firstPlayer.reserve.getTopCard().number <>
secondPlayer.reserve.getTopCard().number - 1
```

```
                                                and
firstPlayer.reserve.getTopCard().number <>
secondPlayer.reserve.getTopCard().number + 1)) then
                                                return false
                                        else (

      secondPlayer.reserve.turnUpTopCard();

      secondPlayer.reserve.addIntoDeckFront(firstPlayer.reserve.getTop
Card());

      firstPlayer.reserve.removeTopCard();
                                                if not
Deck`isEmpty(firstPlayer.reserve.cards) then

      firstPlayer.reserve.turnUpTopCard();
                                                return true;
                                        )
                                ) else (

      if(firstPlayer.reserve.getTopCard().getSuit() <>
secondPlayer.reserve.getTopCard().getSuit()
                                                or
(secondPlayer.reserve.getTopCard().number <>
firstPlayer.reserve.getTopCard().number - 1
                                                and
secondPlayer.reserve.getTopCard().number <>
firstPlayer.reserve.getTopCard().number + 1)) then
                                                return false
                                        else (

      firstPlayer.reserve.turnUpTopCard();

      firstPlayer.reserve.addIntoDeckFront(secondPlayer.reserve.getTop
Card());

      secondPlayer.reserve.removeTopCard();
                                                if not
Deck`isEmpty(secondPlayer.reserve.cards) then

      secondPlayer.reserve.turnUpTopCard();
                                                return true;
                                        )
                                )
                        ) pre not
Deck`isEmpty(firstPlayer.reserve.cards) and not
Deck`isEmpty(secondPlayer.reserve.cards);

                -- move a card from players reserve to opposing waste
                public moveCardFromReserveToWaste : () ==> bool
                        moveCardFromReserveToWaste() == (
                                if turn = <FirstPlayer> then (

      if(firstPlayer.reserve.getTopCard().getSuit() <>
secondPlayer.waste.getTopCard().getSuit()
                                                or
(firstPlayer.reserve.getTopCard().number <>
secondPlayer.waste.getTopCard().number - 1
                                                and
firstPlayer.reserve.getTopCard().number <>
secondPlayer.waste.getTopCard().number + 1)) then
```

```
                                        return false
                                    else (
                                        if not
Deck`isEmpty(secondPlayer.waste.cards) then

      secondPlayer.waste.turnUpTopCard();

      secondPlayer.waste.addIntoDeckFront(firstPlayer.reserve.getTopCa
rd());

      firstPlayer.reserve.removeTopCard();
                                        if not
Deck`isEmpty(firstPlayer.reserve.cards) then

      firstPlayer.reserve.turnUpTopCard();
                                        return true;
                                    )
                                ) else (

      if(secondPlayer.reserve.getTopCard().getSuit() <>
firstPlayer.waste.getTopCard().getSuit()
                                    or
(secondPlayer.reserve.getTopCard().number <>
firstPlayer.waste.getTopCard().number - 1
                                        and
secondPlayer.reserve.getTopCard().number <>
firstPlayer.waste.getTopCard().number + 1)) then
                                        return false
                                    else (
                                        if not
Deck`isEmpty(firstPlayer.waste.cards) then

      firstPlayer.waste.turnUpTopCard();

      firstPlayer.waste.addIntoDeckFront(secondPlayer.reserve.getTopCa
rd());

      secondPlayer.reserve.removeTopCard();
                                        if not
Deck`isEmpty(secondPlayer.reserve.cards) then

      secondPlayer.reserve.turnUpTopCard();
                                        return true;
                                    )
                                )
                            ) pre (not
Deck`isEmpty(firstPlayer.reserve.cards) and not
Deck`isEmpty(secondPlayer.waste.cards) and turn = <FirstPlayer>)
                                        or (not
Deck`isEmpty(secondPlayer.reserve.cards) and not
Deck`isEmpty(firstPlayer.waste.cards) and turn = <SecondPlayer>)
                                post (not
Deck`isEmpty(secondPlayer.waste.cards) and turn = <FirstPlayer> )
                                        or (not
Deck`isEmpty(firstPlayer.waste.cards) and turn = <SecondPlayer> );

                    -- if there's a winner, update the respective score
                    public updatePoints : () ==> ()
                        updatePoints() == (
                            dcl winner:activePlayer :=
self.isGameOver();
```

```
                        if winner = <FirstPlayer> then (
                                firstPlayer.score :=
firstPlayer.score + 30 + len secondPlayer.hand.cards
                                + len secondPlayer.waste.cards + 2
* len secondPlayer.reserve.cards;
                        )
                        else if winner = <SecondPlayer> then (
                                secondPlayer.score :=
secondPlayer.score + 30 + len firstPlayer.hand.cards
                                + len firstPlayer.waste.cards + 2 *
len firstPlayer.reserve.cards;
                        )
                );

  end Board
```

Validação do modelo

Classe MyTestCase

```
class MyTestCase
      /*
      Superclass for test classes, simpler but more practical than
VDMUnit`TestCase.
      For proper use, you have to do: New -> Add VDM Library -> IO.
      JPF, FEUP, MFES, 2014/15.
      */

      operations

            -- Simulates assertion checking by reducing it to pre-
condition checking.
            -- If 'arg' does not hold, a pre-condition violation will
be signaled.
            protected assertTrue: bool ==> ()
                  assertTrue(arg) ==
                  return
            pre arg;

            -- Simulates assertion checking by reducing it to post-
condition checking.
            -- If values are not equal, prints a message in the console
and generates
            -- a post-conditions violation.
            protected assertEqual: ? * ? ==> ()
            assertEqual(expected, actual) ==
                  if expected <> actual then (
                        IO`print("Actual value (");
                        IO`print(actual);
                        IO`print(") different from expected (");
                        IO`print(expected);
                        IO`println(")\n")
                  )
            post expected = actual;
end MyTestCase
```

Classe TestRussianBank

```
/*
TestRussianBank tests the various case scenarios that could be
implemented into this project.
> creates a board, verifying the available cards for each player and
the cards in the game.
> move card from reserve (to any available position, verifying the
rules of the game).
> move card from tableau (to any available position, verifying also
the rules of the game).
> ends the game, checking whose player won (and with how many points).
*/

class TestRussianBank is subclass of MyTestCase
	operations
		public  static main: () ==> ()
			main() == (
				new TestRussianBank().availableCardsStart();
				new TestRussianBank().moveCardFromReserve();
				new TestRussianBank().moveCardFromReserve2();
				new TestRussianBank().moveCardFromTableau();
				new TestRussianBank().moveCardFromTableau2();
				new TestRussianBank().calculateEndingPoints();
				new TestRussianBank().calculateEndingPoints2();
			);

		-- verifies if the available cards at the beginning of the
game
		-- are the ones defined by the game rules.
		-- answers the R1 requirement
		public availableCardsStart: () ==> ()
			availableCardsStart() == (
				dcl bd: Board := new Board();

				-- Operation createStartingBoard copied for
testing purposes
				bd.firstPlayer := new Player(<Red>);
				bd.secondPlayer := new Player(<Black>);


	assertEqual(Deck`isShuffled(bd.firstPlayer.fullDeck),false);

	assertEqual(Deck`isShuffled(bd.secondPlayer.fullDeck),false);

				-- shuffle the decks for each player
				bd.firstPlayer.shuffleDeck();
				bd.secondPlayer.shuffleDeck();


	assertEqual(Deck`isShuffled(bd.firstPlayer.fullDeck),true);

	assertEqual(Deck`isShuffled(bd.secondPlayer.fullDeck),true);

				-- creates the hands and reserve pile for each
player
				bd.firstPlayer.createHandReserve();
				bd.secondPlayer.createHandReserve();

				assertEqual(len bd.firstPlayer.hand.cards,39);
```

```
                        assertEqual(len bd.secondPlayer.hand.cards,39);

                        -- fills the tableau with 4 cards dealt from
each play's hand
                        bd.fillInitialTableau();

                        assertEqual(len bd.firstPlayer.hand.cards,35);
                        assertEqual(len bd.secondPlayer.hand.cards,35);
                        -- create the empty foundations
                        bd.fillInitialFoundations();
                        -- see who's the first player to make a move
(the one with the lowest-ranking reserve card)
                        bd.setFirstTurn();

                        for i = 1 to 8 do (

      assertEqual(bd.tableau(i).getTopCard().status, <Face_Up>);
                              assertEqual(len bd.tableau(i).cards,1);

      assertEqual(Deck`isEmpty(bd.foundations(i).cards),true);
                        );

      assertEqual(bd.firstPlayer.reserve.getTopCard().status,
<Face_Up>);

      assertEqual(bd.secondPlayer.reserve.getTopCard().status,
<Face_Up>);

      assertEqual(Deck`isEmpty(bd.firstPlayer.waste.cards),true);

      assertEqual(Deck`isEmpty(bd.secondPlayer.waste.cards),true);
                        for i = 2 to len bd.secondPlayer.reserve.cards
do (

      assertEqual(bd.firstPlayer.reserve.cards(i).status,
<Face_Down>);

      assertEqual(bd.secondPlayer.reserve.cards(i).status,
<Face_Down>);
                        );

                        assertEqual(len
bd.firstPlayer.reserve.cards,13);
                        assertEqual(len
bd.secondPlayer.reserve.cards,13);

                        return
                  );

            -- moves a card from reserve to waste, foundation and
tableau
            -- answers the R2, R5, R6 requirements
            public moveCardFromReserve: () ==> ()
                  moveCardFromReserve() == (
                        -- board for testing purposes
                        dcl bd:Board := new Board();
                        dcl cd1:Card := new Card(7,<Clubs>);
                        dcl cd2:Card := new Card(1,<Spades>);
                        dcl cd3:Card := new Card(6,<Hearts>);
                        dcl cd4:Card := new Card(5,<Diamonds>);
                        dcl cd5:Card := new Card(2,<Spades>);
```

```
                        dcl cd6:Card := new Card(3,<Spades>);
                        dcl cd7:Card := new Card(4,<Diamonds>);

                        bd.firstPlayer := new Player(<Red>);
                        bd.secondPlayer := new Player(<Black>);
                        bd.turn := <FirstPlayer>;

                        bd.tableau(1) := new Deck();
                        bd.addCardToTableau(1,cd1);

                        bd.foundations(1) := new Deck();

                        bd.firstPlayer.reserve.addIntoDeckFront(cd5);
                        bd.firstPlayer.reserve.addIntoDeckFront(cd4);
                        bd.firstPlayer.reserve.addIntoDeckFront(cd3);
                        bd.firstPlayer.reserve.addIntoDeckFront(cd2);
                        bd.firstPlayer.reserve.turnUpTopCard();

                        bd.secondPlayer.reserve.addIntoDeckFront(cd6);
                        bd.secondPlayer.reserve.turnUpTopCard();

                        bd.secondPlayer.waste.addIntoDeckFront(cd7);
                        bd.secondPlayer.waste.turnUpTopCard();

        assertEqual(bd.moveCardFromReserveToFoundation(1),true);

        assertEqual(bd.moveCardFromReserveToFoundation(1),false);

        assertEqual(bd.moveCardFromReserveToTableau(1),true);

        assertEqual(bd.moveCardFromReserveToTableau(1),false);

        assertEqual(bd.moveCardFromReserveToReserve(),false);

        assertEqual(bd.moveCardFromReserveToWaste(),true);

        assertEqual(bd.moveCardFromReserveToWaste(),false);

        assertEqual(bd.moveCardFromReserveToReserve(),true);

                );

                -- moves a card from reserve to waste, foundation and
tableau
                -- answers the R4, R5 and R6 requirements
            public moveCardFromReserve2: () ==> ()
                moveCardFromReserve2() == (
                    -- board for testing purposes
                    dcl bd:Board := new Board();
                    dcl cd1:Card := new Card(7,<Clubs>);
                    dcl cd2:Card := new Card(1,<Spades>);
                    dcl cd3:Card := new Card(6,<Hearts>);
                    dcl cd4:Card := new Card(5,<Diamonds>);
                    dcl cd5:Card := new Card(2,<Spades>);
                    dcl cd6:Card := new Card(3,<Spades>);
                    dcl cd7:Card := new Card(4,<Diamonds>);
```

```
                        bd.firstPlayer := new Player(<Red>);
                        bd.secondPlayer := new Player(<Black>);
                        bd.turn := <SecondPlayer>;

                        bd.tableau(1) := new Deck();
                        bd.addCardToTableau(1,cd1);

                        bd.foundations(1) := new Deck();

                        bd.secondPlayer.reserve.addIntoDeckFront(cd5);
                        bd.secondPlayer.reserve.addIntoDeckFront(cd4);
                        bd.secondPlayer.reserve.addIntoDeckFront(cd3);
                        bd.secondPlayer.reserve.addIntoDeckFront(cd2);
                        bd.secondPlayer.reserve.turnUpTopCard();

                        bd.firstPlayer.reserve.addIntoDeckFront(cd6);
                        bd.firstPlayer.reserve.turnUpTopCard();

                        bd.firstPlayer.waste.addIntoDeckFront(cd7);
                        bd.firstPlayer.waste.turnUpTopCard();


        assertEqual(bd.moveCardFromReserveToFoundation(1),true);

        assertEqual(bd.moveCardFromReserveToFoundation(1),false);


        assertEqual(bd.moveCardFromReserveToTableau(1),true);

        assertEqual(bd.moveCardFromReserveToTableau(1),false);


        assertEqual(bd.moveCardFromReserveToReserve(),false);


        assertEqual(bd.moveCardFromReserveToWaste(),true);

        assertEqual(bd.moveCardFromReserveToWaste(),false);


        assertEqual(bd.moveCardFromReserveToReserve(),true);

                );

            -- moves a card from tableau to foundation, other tableau,
opponent's reserve and waste
            -- answers the R3, R6, R7 requirements
            public moveCardFromTableau: () ==> ()
                  moveCardFromTableau() == (
                        -- board for testing purposes
                        dcl bd:Board := new Board();
                        dcl cd1:Card := new Card(7,<Clubs>);
                        dcl cd2:Card := new Card(6,<Hearts>);
                        dcl cd3:Card := new Card(1,<Spades>);
                        dcl cd4:Card := new Card(8,<Clubs>);
                        dcl cd5:Card := new Card(5,<Hearts>);
                        dcl cd6:Card := new Card(7,<Spades>);

                        bd.firstPlayer := new Player(<Red>);
                        bd.secondPlayer := new Player(<Black>);
```

```
                bd.turn := <FirstPlayer>;

                bd.tableau(1) := new Deck();
                bd.addCardToTableau(1,cd1);
                bd.addCardToTableau(1,cd2);

                bd.tableau(2) := new Deck();
                bd.addCardToTableau(2,cd3);

                bd.tableau(3) := new Deck();
                bd.addCardToTableau(3,cd6);

                bd.foundations(1) := new Deck();

                bd.secondPlayer.reserve.addIntoDeckFront(cd4);
                bd.secondPlayer.reserve.turnUpTopCard();

                bd.secondPlayer.waste.addIntoDeckFront(cd5);
                bd.secondPlayer.waste.turnUpTopCard();


assertEqual(bd.moveCardFromTableauToWaste(2),false);

assertEqual(bd.moveCardFromTableauToTableau(2,3),false);

assertEqual(bd.moveCardFromTableauToFoundation(2,1),true);

assertEqual(bd.moveCardFromTableauToFoundation(1,1),false);

assertEqual(bd.moveCardFromTableauToTableau(1,3),true);

assertEqual(bd.moveCardFromTableauToReserve(3),false);

assertEqual(bd.moveCardFromTableauToReserve(1),true);

assertEqual(bd.moveCardFromTableauToWaste(3),true);

            );

      -- moves a card from tableau to foundation, other tableau,
      -- opponent's reserve and waste
      -- answers the R3, R6, R7 requirements
      public moveCardFromTableau2: () ==> ()
            moveCardFromTableau2() == (
                  -- board for testing purposes
                  dcl bd:Board := new Board();
                  dcl cd1:Card := new Card(7,<Clubs>);
                  dcl cd2:Card := new Card(6,<Hearts>);
                  dcl cd3:Card := new Card(1,<Spades>);
                  dcl cd4:Card := new Card(8,<Clubs>);
                  dcl cd5:Card := new Card(5,<Hearts>);
                  dcl cd6:Card := new Card(7,<Spades>);

                  bd.firstPlayer := new Player(<Red>);
                  bd.secondPlayer := new Player(<Black>);
                  bd.turn := <SecondPlayer>;
```

```
                bd.tableau(1) := new Deck();
                bd.addCardToTableau(1,cd1);
                bd.addCardToTableau(1,cd2);

                bd.tableau(2) := new Deck();
                bd.addCardToTableau(2,cd3);

                bd.tableau(3) := new Deck();
                bd.addCardToTableau(3,cd6);

                bd.foundations(1) := new Deck();

                bd.firstPlayer.reserve.addIntoDeckFront(cd4);
                bd.firstPlayer.reserve.turnUpTopCard();

                bd.firstPlayer.waste.addIntoDeckFront(cd5);
                bd.firstPlayer.waste.turnUpTopCard();


assertEqual(bd.moveCardFromTableauToWaste(2),false);

assertEqual(bd.moveCardFromTableauToTableau(2,3),false);

assertEqual(bd.moveCardFromTableauToFoundation(2,1),true);

assertEqual(bd.moveCardFromTableauToFoundation(1,1),false);

assertEqual(bd.moveCardFromTableauToTableau(1,3),true);

assertEqual(bd.moveCardFromTableauToReserve(3),false);

assertEqual(bd.moveCardFromTableauToReserve(1),true);

assertEqual(bd.moveCardFromTableauToWaste(3),true);

            );

      -- answers the R8 requirement
      public calculateEndingPoints: () ==> ()
            calculateEndingPoints() == (
            -- board for testing purposes
                  dcl bd:Board := new Board();
                  dcl cd1:Card := new Card(7,<Clubs>);
                  dcl cd2:Card := new Card(6,<Hearts>);
                  dcl cd3:Card := new Card(1,<Spades>);
                  dcl cd4:Card := new Card(8,<Clubs>);
                  dcl cd5:Card := new Card(5,<Hearts>);
                  dcl cd6:Card := new Card(7,<Spades>);

                  bd.firstPlayer := new Player(<Red>);
                  bd.secondPlayer := new Player(<Black>);
                  bd.turn := <FirstPlayer>;

                  bd.tableau(1) := new Deck();
                  bd.addCardToTableau(1,cd1);
                  bd.addCardToTableau(1,cd2);
```

```
                    bd.tableau(2) := new Deck();
                    bd.addCardToTableau(2,cd3);

                    bd.tableau(3) := new Deck();
                    bd.addCardToTableau(3,cd6);

                    bd.foundations(1) := new Deck();

                    bd.secondPlayer.reserve.addIntoDeckFront(cd4);
                    bd.secondPlayer.reserve.turnUpTopCard();

                    bd.secondPlayer.waste.addIntoDeckFront(cd5);
                    bd.secondPlayer.waste.turnUpTopCard();


assertEqual(bd.moveCardFromTableauToWaste(2),false);

assertEqual(bd.moveCardFromTableauToTableau(2,3),false);

assertEqual(bd.moveCardFromTableauToFoundation(2,1),true);


assertEqual(bd.moveCardFromTableauToFoundation(1,1),false);

assertEqual(bd.moveCardFromTableauToTableau(1,3),true);


assertEqual(bd.moveCardFromTableauToReserve(3),false);

assertEqual(bd.moveCardFromTableauToReserve(1),true);


assertEqual(bd.moveCardFromTableauToWaste(3),true);

                    bd.updatePoints();

                    assertEqual(bd.firstPlayer.score,36);
            );

      -- answers the R8 requirement
      public calculateEndingPoints2: () ==> ()
            calculateEndingPoints2() == (
            -- board for testing purposes
                  dcl bd:Board := new Board();
                  dcl cd1:Card := new Card(7,<Clubs>);
                  dcl cd2:Card := new Card(6,<Hearts>);
                  dcl cd3:Card := new Card(1,<Spades>);
                  dcl cd4:Card := new Card(8,<Clubs>);
                  dcl cd5:Card := new Card(5,<Hearts>);
                  dcl cd6:Card := new Card(7,<Spades>);

                  bd.firstPlayer := new Player(<Red>);
                  bd.secondPlayer := new Player(<Black>);
                  bd.turn := <SecondPlayer>;

                  bd.tableau(1) := new Deck();
                  bd.addCardToTableau(1,cd1);
                  bd.addCardToTableau(1,cd2);

                  bd.tableau(2) := new Deck();
```

```
                        bd.addCardToTableau(2,cd3);

                        bd.tableau(3) := new Deck();
                        bd.addCardToTableau(3,cd6);

                        bd.foundations(1) := new Deck();

                        bd.firstPlayer.reserve.addIntoDeckFront(cd4);
                        bd.firstPlayer.reserve.turnUpTopCard();

                        bd.firstPlayer.waste.addIntoDeckFront(cd5);
                        bd.firstPlayer.waste.turnUpTopCard();


        assertEqual(bd.moveCardFromTableauToWaste(2),false);

        assertEqual(bd.moveCardFromTableauToTableau(2,3),false);

        assertEqual(bd.moveCardFromTableauToFoundation(2,1),true);

        assertEqual(bd.moveCardFromTableauToFoundation(1,1),false);

        assertEqual(bd.moveCardFromTableauToTableau(1,3),true);

        assertEqual(bd.moveCardFromTableauToReserve(3),false);

        assertEqual(bd.moveCardFromTableauToReserve(1),true);

        assertEqual(bd.moveCardFromTableauToWaste(3),true);

                        bd.updatePoints();

                        assertEqual(bd.secondPlayer.score,36);
                    );
end TestRussianBank
```

Verificação do modelo

Exemplo de verificação de pre e pós condição

Por exemplo, na operação '*fillInitialTableau*' da classe *Board*, a pre-condição e a pós-condição são as seguintes:

Pre-condição	pre len tableau = 0
Pós-condição	pre len tableau = 8

Efetivamente, a operação '*fillInitialTableau*' só pode ser usado uma única vez, no início do jogo. O modelo verifica exatamente isso, uma vez que o contrato estabelece que a pré-condição é o tableau ter tamanho 0 (ou seja, não ter nenhuma carta em nenhuma das 8 casas), e a pós-condição é o tableau ter tamanho 8 (ou seja, ter todas as as casas do tableau iniciadas com uma única carta). Após isto, esta operação não pode ser mais chamada durante a execução do código.

Exemplo de verificação de uma invariante

A invariante de seu nome *cardValue*, na classe *Card*, é gerada pelo proof obligations da seguinte forma:

110	Card`Number	Type invariant satisfiable

Esta invariável é verificada pelo nosso projeto, uma vez que é impossível adicionar valores que estejam fora do âmbito da definição da invariante:

```
(exists cardValue:Number & (cardValue in set {1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13}))
```

Geração de código em java

O grupo testou a funcionalidade do Overture em gerar código em Java a partir do modelo, e conseguiu fazê-lo com sucesso. As funções e as classes geradas mostraram-se úteis para perceber o potencial de conseguir integrar um projeto feito nesta ferramenta na linguagem Java, no entanto, o código gerado pelo Overture demonstrou alguns problemas:

- Quando executado, o código mostrava consistentemente erros do tipo "IndexOutOfBoundsException". O nosso grupo ainda tentou mudar algumas definições do modelo e até mesmo do código, mas após muitas alterações e o problema continuar a persistir, o grupo acabou por depositar as suas concentrações noutras partes do projeto.
- O Overture não conseguiu gerar, em funções Java, alguns métodos inerentes ao VDM++, tal como o 'reverse', que inverte uma lista. O nosso grupo acabou por implementar a função à parte, obtendo sucesso dessa maneira.

Apesar das limitações dessa geração, o nosso grupo ainda tentou perceber e admirar a geração do código, além de testar individualmente algumas funções, tais como a de criar uma carta e mudar os seus atributos. No entanto, e uma vez que o projeto não era, de todo, direcionado a este propósito, o grupo acabou por não avançar mais além.

Conclusões finais

Como conclusão final, o nosso grupo retira que o trabalho à volta do mesmo serviu para aprofundar e enriquecer os conhecimentos nas áreas de modelação formal em engenharia de software, que não tinham sido ainda exploradas até agora no curso.

O grupo ficou fortemente agradado com o resultado final, reconhecendo que a quantidade enorme de regras e pequenos detalhes inerentes ao próprio jogo seriam impossíveis de implementar em tempo útil, e aumentariam o tamanho do código para níveis não pretendidos (já mesmo assim, como foi possível verificar, o limite máximo de código em 10 páginas neste relatório foi levemente ultrapassado). Não obstante, os requisitos para este trabalho foram efetivamente cumpridos, e a possibilidade de interagir com ferramentas como o Overture e o Modelio mostrou-se bastante interessante. Este projeto aborda o uso das três estruturas de dados (*maps, sets* e *seqs*).

Como possíveis melhorias no futuro, haveria a possibilidade de se implementar mais regras, e por consequência mais testes. Outra possibilidade seria, por exemplo, a integração do código em java com uma interface gráfica, para uma futura interação mais intuitiva com o utilizador.

O projeto demorou, no total, cerca de 8 dias a ser concretizado (com esforço muito mais notável nos últimos 3 dias), e o trabalho foi equitativamente distribuído pelos dois elementos do grupo.

Referências

1. Overture tool web site, http://overturetool.org
2. http://www.pagat.com/patience/crapette.html, para ver as regras do jogo.
3. http://fm06.mcmaster.ca/VDM++%20tutorial%20FM%202006%20handouts.pdf , para aprender a programar em VDM++
4. https://moodle.up.pt/course/view.php?id=1085 , slides da Disciplina 2015/2016, Prof. João Pascoal-Faria, página do Moodle da UP, para reforçar a aprendizagem em VDM++
5. https://www.modelio.org/ , para modelar o UML apresentado neste relatório.