Faculdade de Engenharia da Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação
Métodos Formais em Engenharia de Software
4º Ano - 1º Semestre - 2012/2013

Especificação Formal do Jogo Marel (*Nine Men's Morris*)

Relatório

Diogo André Rocha Teixeira - ei09086@fe.up.pt

Marco André Moreira Amador - ei09006@fe.up.pt

7 de Dezembro de 2012

Índice

Requisitos e Restrições

Nine Men's Morris é um jogo de tabuleiro para dois jogadores, que surgiu nos tempos do Império Romano. Cada jogador dispõe de nove peças, e o jogo ocorre em duas fases distintas:

- Disposição das peças no tabuleiro – Jogadores colocam as suas peças numa posição livre do tabuleiro, alternadamente.
- Movimentação das peças – Após todas as peças dos jogadores estarem dispostas no tabuleiro, os jogadores podem movimentar uma das suas peças de cada vez. Esta fase ocorre até um dos jogadores ficar com menos de 3 peças, sendo que esse jogador perde o jogo.

A remoção de peças de um jogador ocorre quando o jogador adversário forma um "mill", ou seja, uma linha horizontal ou vertical de 3 das suas peças em posições adjacentes. Quando um "mill" é formado, o jogador que o forma pode retirar qualquer uma das peças do jogador adversário, exceto as que façam parte de um "mill" do adversário (a menos que não haja outras hipóteses, ou seja, peças do adversário que não façam parte de um "mill" formado por ele). No entanto, para voltar a remover uma peça do adversário, é necessário voltar a formar outro "mill", ou desfazer o já existente e voltar a criá-lo posteriormente.

Na figura seguinte é possível ver a forma do tabuleiro, bem como as posições possíveis do mesmo (movimentações só são possíveis entre posições ligadas por uma linha):

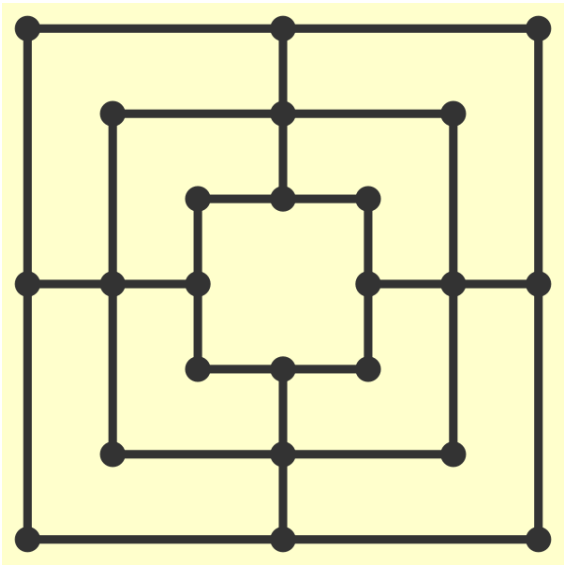


Fig. 1 – Estado inicial do tabuleiro, com as posições e movimentações possíveis.

Podemos igualmente ver o exemplo de uma situação de jogo em que cada um dos jogadores tem um "mill" formado (repare-se que o jogador com as peças brancas, ao mover a peça na posição *e3* para *d3* e vice-versa com o decorrer das jogadas, forma sempre um "mill", o que lhe permite tirar uma peça ao jogador preto por cada turno de jogo):

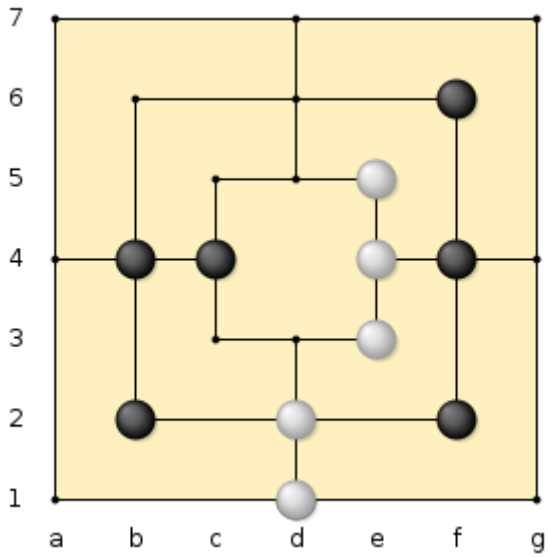


Fig.2 – Exemplo de situação de jogo, com formação de "mills"

Especificação das Principais Restrições

1. **Diferenciação das Fases de Jogo (Colocação e Movimentação)**

```
public isPhaseOne : () ==> bool
  isPhaseOne() ==
  (
    for all p in set players do
    (
      if p.getUnplayedPieces() > 0
      then return true;
    );
    return false;
  )
  pre players <> {};

  public isPhaseTwo : () ==> bool
  isPhaseTwo() ==
  (
    for all p in set players do
    (
      if p.getUnplayedPieces() > 0
```

```
      then return false;
    );
    return true;
  )
  pre players <> {};
```

2. Movimentação de peças só é possível para posições adjacentes.

```
public movable : PieceType * seq of nat1 * seq of nat1 ==> bool
  movable(currentPlayer, origin, dest) ==
  (
   if board(origin) = currentPlayer and board(dest) = <UNDEFINED>
   then return movableHorizontal(origin, dest) or
      movableVertical(origin, dest)
   else return false;
  )
  pre validCoords(dest) and validCoords(origin);

  public movableHorizontal : seq of nat1 * seq of nat1 ==> bool
  movableHorizontal(origin, dest) ==
  (
   if origin(1) = dest(1)
   then
   (
      if origin(1) = 1 or origin(1) = 7
      then return origin(2) = (dest(2) - 3) or origin(2) = (dest(2)
      + 3)
      else if origin(1) = 2 or origin(1) = 6
      then return origin(2) = (dest(2) - 2) or origin(2) = (dest(2)
      + 2)
      else return origin(2) = (dest(2) - 1) or origin(2) = (dest(2)
      + 1);
    )
    else return false;
  )
  pre validCoords(origin) and validCoords(dest);

  public movableVertical : seq of nat1 * seq of nat1 ==> bool
  movableVertical(origin, dest) ==
  (
    if origin(2) = dest(2)
    then
    (
      if origin(2) = 1 or origin(2) = 7
      then return origin(1) = (dest(1) - 3) or origin(1) = (dest(1)
      + 3)
      else if origin(2) = 2 or origin(2) = 6
      then return origin(1) = (dest(1) - 2) or origin(1) = (dest(1)
      + 2)
      else return origin(1) = (dest(1) - 1) or origin(1) = (dest(1)
      + 1);
    )
    else return false;
  )
  pre validCoords(origin) and validCoords(dest);
```

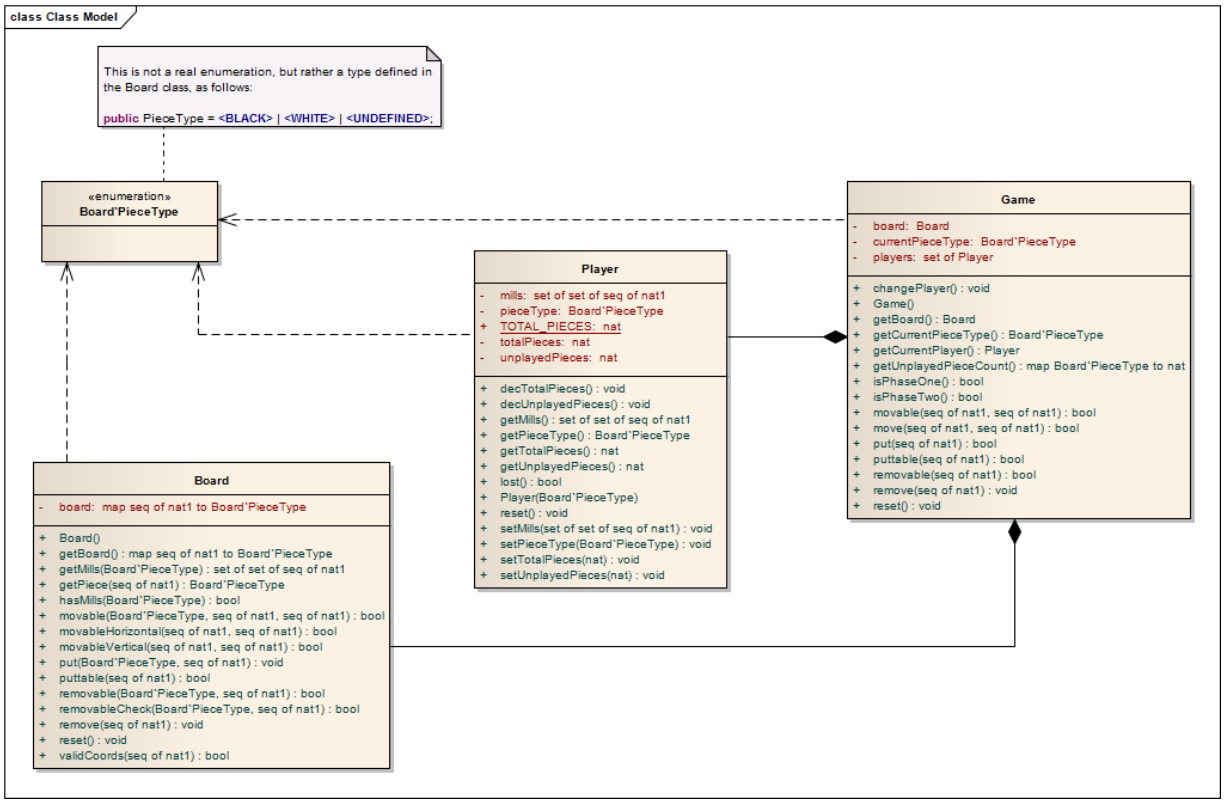
3. **Remoção de Peças de um "Mill" do Adversário só é possível quando o mesmo não tem peças "soltas" (que não façam parte de um "Mill").**

```
public removable : PieceType * seq of nat1 ==> bool
  removable(currentPlayer, coord) ==
  (
   dcl piece : PieceType := board(coord);
   if piece <> <UNDEFINED> and piece <> currentPlayer
   then return removableCheck(currentPlayer, coord)
   else return false;
  )
  pre currentPlayer <> <UNDEFINED> and validCoords(coord);

  public removableCheck : PieceType * seq of nat1 ==> bool
  removableCheck(currentPlayer, coord) ==
  (
    dcl player : PieceType;
    dcl allCoords : set of seq of nat1;
    dcl millCoords : set of seq of nat1;
    if currentPlayer = <WHITE>
    then player := <BLACK>
    else player := <WHITE>;
    allCoords := dom ( board :> { player } );
    millCoords := dunion getMills(player);
    return coord not in set millCoords or allCoords = millCoords;
  )
  pre currentPlayer <> <UNDEFINED> and validCoords(coord);

public getMills : PieceType ==> set of set of seq of nat1
  getMills(piece) ==
  (
   dcl coords : set of seq of nat1 := dom ( board :> { piece } );
   dcl mills : set of set of seq of nat1 := {};
 if ( { [1,1] , [1,4], [1,7] } subset coords ) then mills := mills
union  { { [1,1] , [1,4], [1,7] } };
 if ( { [2,2] , [2,4], [2,6] } subset coords ) then mills := mills
union  { { [2,2] , [2,4], [2,6] } };
 if ( { [3,3] , [3,4], [3,5] } subset coords ) then mills := mills
union  { { [3,3] , [3,4], [3,5] } };
 if ( { [5,3] , [5,4], [5,5] } subset coords ) then mills := mills
union  { { [5,3] , [5,4], [5,5] } };
 if ( { [6,2] , [6,4], [6,6] } subset coords ) then mills := mills
union  { { [6,2] , [6,4], [6,6] } };
 if ( { [7,1] , [7,4], [7,7] } subset coords ) then mills := mills
union  { { [7,1] , [7,4], [7,7] } };
 if ( { [1,1] , [4,1], [7,1] } subset coords ) then mills := mills
union  { { [1,1] , [4,1], [7,1] } };
 if ( { [2,2] , [4,2], [6,2] } subset coords ) then mills := mills
union  { { [2,2] , [4,2], [6,2] } };
 if ( { [3,3] , [4,3], [5,3] } subset coords ) then mills := mills
union  { { [3,3] , [4,3], [5,3] } };
 if ( { [3,5] , [4,5], [5,5] } subset coords ) then mills := mills
union  { { [3,5] , [4,5], [5,5] } };
 if ( { [2,6] , [4,6], [6,6] } subset coords ) then mills := mills
union  { { [2,6] , [4,6], [6,6] } };
 if ( { [1,7] , [4,7], [7,7] } subset coords ) then mills := mills
union  { { [1,7] , [4,7], [7,7] } };
 if ( { [4,1] , [4,2], [4,3] } subset coords ) then mills := mills
union  { { [4,1] , [4,2], [4,3] } };
```

```
 if ( { [4,5] , [4,6], [4,7] } subset coords ) then mills := mills
union  { { [4,5] , [4,6], [4,7] } };
 if ( { [1,4] , [2,4], [3,4] } subset coords ) then mills := mills
union  { { [1,4] , [2,4], [3,4] } };
 if ( { [5,4] , [6,4], [7,4] } subset coords ) then mills := mills
union  { { [5,4] , [6,4], [7,4] } };
 return mills;
  )
  pre piece <> <UNDEFINED>;
```

Diagrama de Classes do Sistema

No início da realização do trabalho, foi elaborado o seguinte diagrama de classes para o mesmo, contendo as estruturas e operações necessárias ao funcionamento da lógica do jogo.



Como se pode constatar pela imagem, foram definidas três classes para o jogo:

- ***Player*** – Responsável pelos dados de um jogador, como número de peças atual (para verificação de vencedor, no caso de ter menos do que três peças em jogo), número de peças por colocar (para verificação da fase de jogo atual), tipo de peças do jogador (Brancas ou Pretas) e "mills" formados pelo mesmo.
- ***Board*** – Refere-se à definição do tabuleiro, e validação de operações de colocação, movimentação e remoção de peças.

- ***Game*** – Responsável pelo ciclo de jogo, envolvendo as duas classes anteriores para a definição do tabuleiro e jogadores. Envolve também as operações de colocação, movimentação e remoção de peças, bem como verificação da fase de jogo atual e mudança de turnos.

Este diagrama foi usado para geração do esqueleto da especificação de VDM++, através da funcionalidade *UML Link* da ferramenta *VDM++ Toolbox*.

Definição das Classes em VDM++ (e Cobertura de Testes)

Classe Board

Definição em VDM++ da classe Board (com informação de Cobertura de Testes):

```
class Board
types
  public PieceType = <BLACK> | <WHITE> | <UNDEFINED>;

instance variables
  private board : map seq of nat1 to PieceType := {|->};

operations

  public Board : () ==> Board
  Board() ==
  (
   board := {
     [1,1] |-> <UNDEFINED>, [1,4] |-> <UNDEFINED>, [1,7] |->
<UNDEFINED>,
     [2,2] |-> <UNDEFINED>, [2,4] |-> <UNDEFINED>, [2,6] |->
<UNDEFINED>,
     [3,3] |-> <UNDEFINED>, [3,4] |-> <UNDEFINED>, [3,5] |->
<UNDEFINED>,
     [4,1] |-> <UNDEFINED>, [4,2] |-> <UNDEFINED>, [4,3] |->
<UNDEFINED>,
     [4,5] |-> <UNDEFINED>, [4,6] |-> <UNDEFINED>, [4,7] |->
<UNDEFINED>,
     [5,3] |-> <UNDEFINED>, [5,4] |-> <UNDEFINED>, [5,5] |->
<UNDEFINED>,
     [6,2] |-> <UNDEFINED>, [6,4] |-> <UNDEFINED>, [6,6] |->
<UNDEFINED>,
     [7,1] |-> <UNDEFINED>, [7,4] |-> <UNDEFINED>, [7,7] |->
<UNDEFINED>
   };
  );

  public reset : () ==> ()
  reset() ==
  (
    board := {
     [1,1] |-> <UNDEFINED>, [1,4] |-> <UNDEFINED>, [1,7] |->
<UNDEFINED>,
     [2,2] |-> <UNDEFINED>, [2,4] |-> <UNDEFINED>, [2,6] |->
<UNDEFINED>,
```

```
     [3,3] |-> <UNDEFINED>, [3,4] |-> <UNDEFINED>, [3,5] |->
<UNDEFINED>,
     [4,1] |-> <UNDEFINED>, [4,2] |-> <UNDEFINED>, [4,3] |->
<UNDEFINED>,
     [4,5] |-> <UNDEFINED>, [4,6] |-> <UNDEFINED>, [4,7] |->
<UNDEFINED>,
     [5,3] |-> <UNDEFINED>, [5,4] |-> <UNDEFINED>, [5,5] |->
<UNDEFINED>,
     [6,2] |-> <UNDEFINED>, [6,4] |-> <UNDEFINED>, [6,6] |->
<UNDEFINED>,
     [7,1] |-> <UNDEFINED>, [7,4] |-> <UNDEFINED>, [7,7] |->
<UNDEFINED>
   };
  );

  public getPiece : seq of nat1 ==> PieceType
  getPiece(coords) ==
  (
    return board(coords);
  )
  pre validCoords(coords) and board <> {|->};

  public getBoard : () ==> map seq of nat1 to PieceType
  getBoard() ==
  (
    return board;
  );

  public validCoords : seq of nat1 ==> bool
  validCoords(coord) ==
  (
   return coord in set dom board;
  )
  pre len coord = 2;

  public remove : seq of nat1 ==> ()
  remove(coord) ==
  (
   board := board ++ { coord |-> <UNDEFINED> };
  )
  pre board(coord) <> <UNDEFINED> and validCoords(coord)
  post board(coord) = <UNDEFINED>;

  public puttable : seq of nat1 ==> bool
  puttable(coord) ==
  (
   return board(coord) = <UNDEFINED>;
  )
  pre validCoords(coord);

  public movable : PieceType * seq of nat1 * seq of nat1 ==> bool
  movable(currentPlayer, origin, dest) ==
  (
   if board(origin) = currentPlayer and board(dest) = <UNDEFINED>
   then return movableHorizontal(origin, dest) or
movableVertical(origin, dest)
   else return false;
  )
  pre validCoords(dest) and validCoords(origin);

  public movableHorizontal : seq of nat1 * seq of nat1 ==> bool
```

```
  movableHorizontal(origin, dest) ==
  (
    if origin(1) = dest(1)
    then
    (
      if origin(1) = 1 or origin(1) = 7
      then return origin(2) = (dest(2) - 3) or origin(2) = (dest(2) +
3)
      else if origin(1) = 2 or origin(1) = 6
      then return origin(2) = (dest(2) - 2) or origin(2) = (dest(2) +
2)
      else return origin(2) = (dest(2) - 1) or origin(2) = (dest(2) +
1);
    )
    else return false;
  )
  pre validCoords(origin) and validCoords(dest);

  public movableVertical : seq of nat1 * seq of nat1 ==> bool
  movableVertical(origin, dest) ==
  (
    if origin(2) = dest(2)
    then
    (
      if origin(2) = 1 or origin(2) = 7
      then return origin(1) = (dest(1) - 3) or origin(1) = (dest(1) +
3)
      else if origin(2) = 2 or origin(2) = 6
      then return origin(1) = (dest(1) - 2) or origin(1) = (dest(1) +
2)
      else return origin(1) = (dest(1) - 1) or origin(1) = (dest(1) +
1);
    )
    else return false;
  )
  pre validCoords(origin) and validCoords(dest);

  public put : PieceType * seq of nat1 ==> ()
  put(piece, coord) ==
  (
   board := board ++ { coord |-> piece };
  )
  pre board(coord) = <UNDEFINED> and validCoords(coord)
  post board(coord) = piece;

  public hasMills : PieceType ==> bool
  hasMills(piece) ==
  (
   dcl coords : set of seq of nat1 := dom ( board :> { piece } );
 return
        -- horizontal
        ( { [1,1] , [1,4], [1,7] } subset coords ) or
        ( { [2,2] , [2,4], [2,6] } subset coords ) or
        ( { [3,3] , [3,4], [3,5] } subset coords ) or
        ( { [5,3] , [5,4], [5,5] } subset coords ) or
        ( { [6,2] , [6,4], [6,6] } subset coords ) or
        ( { [7,1] , [7,4], [7,7] } subset coords ) or
        -- vertical
        ( { [1,1] , [4,1], [7,1] } subset coords ) or
        ( { [2,2] , [4,2], [6,2] } subset coords ) or
        ( { [3,3] , [4,3], [5,3] } subset coords ) or
```

```
        ( { [3,5] , [4,5], [5,5] } subset coords ) or
        ( { [2,6] , [4,6], [6,6] } subset coords ) or
        ( { [1,7] , [4,7], [7,7] } subset coords ) or
        -- special
        ( { [4,1] , [4,2], [4,3] } subset coords ) or
        ( { [4,5] , [4,6], [4,7] } subset coords ) or
        ( { [1,4] , [2,4], [3,4] } subset coords ) or
        ( { [5,4] , [6,4], [7,4] } subset coords );
  )
  pre piece <> <UNDEFINED>;

  public getMills : PieceType ==> set of set of seq of nat1
  getMills(piece) ==
  (
   dcl coords : set of seq of nat1 := dom ( board :> { piece } );
   dcl mills : set of set of seq of nat1 := {};
 if ( { [1,1] , [1,4], [1,7] } subset coords ) then mills := mills
union  { { [1,1] , [1,4], [1,7] } };
 if ( { [2,2] , [2,4], [2,6] } subset coords ) then mills := mills
union  { { [2,2] , [2,4], [2,6] } };
 if ( { [3,3] , [3,4], [3,5] } subset coords ) then mills := mills
union  { { [3,3] , [3,4], [3,5] } };
 if ( { [5,3] , [5,4], [5,5] } subset coords ) then mills := mills
union  { { [5,3] , [5,4], [5,5] } };
 if ( { [6,2] , [6,4], [6,6] } subset coords ) then mills := mills
union  { { [6,2] , [6,4], [6,6] } };
 if ( { [7,1] , [7,4], [7,7] } subset coords ) then mills := mills
union  { { [7,1] , [7,4], [7,7] } };
 if ( { [1,1] , [4,1], [7,1] } subset coords ) then mills := mills
union  { { [1,1] , [4,1], [7,1] } };
 if ( { [2,2] , [4,2], [6,2] } subset coords ) then mills := mills
union  { { [2,2] , [4,2], [6,2] } };
 if ( { [3,3] , [4,3], [5,3] } subset coords ) then mills := mills
union  { { [3,3] , [4,3], [5,3] } };
 if ( { [3,5] , [4,5], [5,5] } subset coords ) then mills := mills
union  { { [3,5] , [4,5], [5,5] } };
 if ( { [2,6] , [4,6], [6,6] } subset coords ) then mills := mills
union  { { [2,6] , [4,6], [6,6] } };
 if ( { [1,7] , [4,7], [7,7] } subset coords ) then mills := mills
union  { { [1,7] , [4,7], [7,7] } };
 if ( { [4,1] , [4,2], [4,3] } subset coords ) then mills := mills
union  { { [4,1] , [4,2], [4,3] } };
 if ( { [4,5] , [4,6], [4,7] } subset coords ) then mills := mills
union  { { [4,5] , [4,6], [4,7] } };
 if ( { [1,4] , [2,4], [3,4] } subset coords ) then mills := mills
union  { { [1,4] , [2,4], [3,4] } };
 if ( { [5,4] , [6,4], [7,4] } subset coords ) then mills := mills
union  { { [5,4] , [6,4], [7,4] } };
 return mills;
  )
  pre piece <> <UNDEFINED>;

  public removable : PieceType * seq of nat1 ==> bool
  removable(currentPlayer, coord) ==
  (
   dcl piece : PieceType := board(coord);
   if piece <> <UNDEFINED> and piece <> currentPlayer
   then return removableCheck(currentPlayer, coord)
   else return false;
  )
  pre currentPlayer <> <UNDEFINED> and validCoords(coord);
```

```
  public removableCheck : PieceType * seq of nat1 ==> bool
  removableCheck(currentPlayer, coord) ==
  (
    dcl player : PieceType;
    dcl allCoords : set of seq of nat1;
    dcl millCoords : set of seq of nat1;
    if currentPlayer = <WHITE>
    then player := <BLACK>
    else player := <WHITE>;
    allCoords := dom ( board :> { player } );
    millCoords := dunion getMills(player);
    return coord not in set millCoords or allCoords = millCoords;
  )
  pre currentPlayer <> <UNDEFINED> and validCoords(coord);

end Board
```

Cobertura de Testes

name	***#calls***	***coverage***
Board`put	76	100%
Board`Board	17	100%
Board`reset	2	100%
Board`remove	3	100%
Board`movable	16	100%
Board`getBoard	2	100%
Board`getMills	19	100%
Board`getPiece	3	100%
Board`hasMills	3	100%
Board`puttable	29	100%
Board`removable	12	100%
Board`validCoords	226	100%
Board`removableCheck	11	100%
Board`movableVertical	8	100%
Board`movableHorizontal	14	100%
total		***100%***

Classe Game

Definição em VDM++ da classe Game (com informação de Cobertura de Testes):

```
class Game
instance variables
  private board : Board := new Board();
  private players : set of Player := {};
  private currentPieceType : Board`PieceType := <WHITE>;
  inv currentPieceType <> <UNDEFINED>

operations

  public Game : () ==> Game
  Game() ==
  (
    board := new Board();
    currentPieceType := <WHITE>;
    players := { new Player(<WHITE>), new Player(<BLACK>) };
  );

  public reset : () ==> ()
  reset() ==
  (
    board := new Board();
    currentPieceType := <WHITE>;
    players := { new Player(<WHITE>), new Player(<BLACK>) };
  );

  public put : seq of nat1 ==> bool
  put(coord) ==
  (
    dcl mills : set of set of seq of nat1;
    dcl player : Player := getCurrentPlayer();
    dcl pMills : set of set of seq of nat1 := player.getMills();
    board.put(currentPieceType, coord);
    mills := board.getMills(currentPieceType);
    player.setMills(mills);
    return card pMills < card mills;
  )
  pre board.validCoords(coord);

  public getBoard : () ==> Board
  getBoard() ==
  (
   return board;
  );

  public remove : seq of nat1 ==> ()
  remove(coord) ==
  (
    board.remove(coord);
    changePlayer();
    getCurrentPlayer().setMills(board.getMills(currentPieceType));
    changePlayer();
  )
  pre board.validCoords(coord);

  public isPhaseOne : () ==> bool
  isPhaseOne() ==
  (
```

```
  for all p in set players do
  (
    if p.getUnplayedPieces() > 0
    then return true;
  );
  return false;
)
pre players <> {};

public isPhaseTwo : () ==> bool
isPhaseTwo() ==
(
  for all p in set players do
  (
    if p.getUnplayedPieces() > 0
    then return false;
  );
  return true;
)
pre players <> {};

public getCurrentPlayer : () ==> Player
getCurrentPlayer() ==
(
  dcl player : Player;
  for all p in set players do
  (
    if p.getPieceType() = currentPieceType
    then player := p;
  );
  return player;
)
pre players <> {}
post isofclass(Player, RESULT) and RESULT in set players;

public move : seq of nat1 * seq of nat1 ==> bool
move(origin, dest) ==
(
  board.remove(origin);
  return put(dest);
)
pre board.validCoords(origin) and board.validCoords(dest);

public puttable : seq of nat1 ==> bool
puttable(coord) ==
(
  board.puttable(coord);
)
pre board.validCoords(coord);

public changePlayer : () ==> ()
changePlayer() ==
(
  if currentPieceType = <WHITE>
  then currentPieceType := <BLACK>
  else currentPieceType := <WHITE>;
)
post currentPieceType <> <UNDEFINED>;

public removable : seq of nat1 ==> bool
removable(coord) ==
```

```
  (
    return board.removable(currentPieceType, coord);
  )
  pre board.validCoords(coord);

  public movable : seq of nat1 * seq of nat1 ==> bool
  movable(origin, dest) ==
  (
    return board.movable(currentPieceType, origin, dest);
  )
  pre board.validCoords(origin) and board.validCoords(dest);

  public getCurrentPieceType : () ==> Board`PieceType
  getCurrentPieceType() ==
  (
    return currentPieceType;
  );

  public getUnplayedPieceCount : () ==> map Board`PieceType to nat
  getUnplayedPieceCount() ==
  (
    dcl count : map Board`PieceType to nat := {|->};
    for all p in set players do
    (
      count := count ++ { p.getPieceType() |-> p.getUnplayedPieces()
};
    );
    return count;
  )
  pre players <> {}
  post dom RESULT <> {} and card dom RESULT = 2;

end Game
```

Cobertura de testes

name	***#calls***	***coverage***
Game`put	7	100%
Game`Game	3	100%
Game`move	1	100%
Game`reset	1	100%
Game`remove	1	100%
Game`movable	2	100%
Game`getBoard	1	100%
Game`puttable	1	100%
Game`removable	1	100%
Game`isPhaseOne	3	100%

Game`isPhaseTwo	3	100%
Game`changePlayer	8	100%
Game`getCurrentPlayer	10	100%
Game`getCurrentPieceType	4	100%
Game`getUnplayedPieceCount	1	100%
total		***100%***

Classe Player

Definição em VDM++ da classe Player (com informação de Cobertura de Testes):

```
class Player
values
  public TOTAL_PIECES : nat = 9;

instance variables
  private pieceType : Board`PieceType := <WHITE>;
  private totalPieces : nat := TOTAL_PIECES;
  private unplayedPieces : nat := TOTAL_PIECES;
  private mills : set of set of seq of nat1 := {};
  inv pieceType <> <UNDEFINED>;
  inv totalPieces >= 0;
  inv unplayedPieces >= 0;

operations

  public Player : (Board`PieceType) ==> Player
  Player(type) ==
  (
    mills := {};
   pieceType := type;
   totalPieces := TOTAL_PIECES;
   unplayedPieces := TOTAL_PIECES;
  )
  pre type <> <UNDEFINED>;

  public setMills : set of set of seq of nat1 ==> ()
  setMills(m) ==
  (
    mills := m;
  );

  public getMills : () ==> set of set of seq of nat1
  getMills() ==
  (
    return mills;
  );

  public lost : () ==> bool
  lost() ==
  (
```

```
 return totalPieces < 3;
);

public reset : () ==> ()
reset() ==
(
 totalPieces := TOTAL_PIECES;
 unplayedPieces := TOTAL_PIECES;
);

public getPieceType : () ==> Board`PieceType
getPieceType() ==
(
 return pieceType;
)
pre pieceType <> <UNDEFINED>;

public decTotalPieces : () ==> ()
decTotalPieces() ==
(
 totalPieces := totalPieces - 1;
)
pre totalPieces - 1 >= 0;

public getTotalPieces : () ==> nat
getTotalPieces() ==
(
 return totalPieces;
);

public setTotalPieces : nat ==> ()
setTotalPieces(nPieces) ==
(
 totalPieces := nPieces;
)
pre nPieces >= 0;

public decUnplayedPieces : () ==> ()
decUnplayedPieces() ==
(
 unplayedPieces := unplayedPieces - 1;
)
pre unplayedPieces - 1 >= 0;

public getUnplayedPieces : () ==> nat
getUnplayedPieces() ==
(
 return unplayedPieces;
);

public setUnplayedPieces : nat ==> ()
setUnplayedPieces(nPieces) ==
(
 unplayedPieces := nPieces;
)
pre nPieces >= 0;

public setPieceType : Board`PieceType ==> ()
setPieceType(type) ==
(
 pieceType := type;
```

```
  )
  pre type <> <UNDEFINED>;
end Player
```

Cobertura de Testes

name	*#calls*	*coverage*
Player`lost	1	100%
Player`reset	1	100%
Player`Player	5	100%
Player`getMills	2	100%
Player`setMills	1	100%
Player`getPieceType	4	100%
Player`setPieceType	1	100%
Player`decTotalPieces	1	100%
Player`getTotalPieces	7	100%
Player`setTotalPieces	3	100%
Player`decUnplayedPieces	1	100%
Player`getUnplayedPieces	6	100%
Player`setUnplayedPieces	2	100%
total		***100%***

Classes de Teste Elaboradas

Classe TestBoard

Definição em VDM++ da classe TestBoard, para testar a classe Board:

```
class TestBoard

operations
 -- operacao auxiliar, que tira partido do facto do
 -- interpretador parar quando se viola uma precondicao
 public AssertTrue : bool ==> ()
 AssertTrue(a) == return
      pre a;

 public AssertFalse : bool ==> ()
 AssertFalse(a) == return
```

```
 pre not a;

 public TestBoardSetup : () ==> ()
 TestBoardSetup() ==
 (
  dcl b : Board := new Board();
  dcl p2 : map seq of nat1 to Board`PieceType := b.getBoard();
  b.reset();
  p2 := b.getBoard();
  b.put(<WHITE>, [1,1]);
  b.put(<BLACK>, [7,7]);
  b.put(<BLACK>, [5,4]);
  AssertTrue(b.getPiece([1,1]) = <WHITE>);
  AssertTrue(b.getPiece([7,7]) = <BLACK>);
  AssertTrue(b.getPiece([7,1]) = <UNDEFINED>);
  AssertFalse(b.puttable([1,1]));
  AssertFalse(b.puttable([7,7]));
  AssertFalse(b.puttable([5,4]));
  b.remove([1,1]);
  AssertTrue(b.puttable([1,1]));
  b.reset();
  AssertTrue(b.puttable([1,1]));
  AssertTrue(b.puttable([1,4]));
  AssertTrue(b.puttable([1,7]));
  AssertTrue(b.puttable([2,2]));
  AssertTrue(b.puttable([2,4]));
  AssertTrue(b.puttable([2,6]));
  AssertTrue(b.puttable([3,3]));
  AssertTrue(b.puttable([3,4]));
  AssertTrue(b.puttable([3,5]));
  AssertTrue(b.puttable([4,1]));
  AssertTrue(b.puttable([4,2]));
  AssertTrue(b.puttable([4,3]));
  AssertTrue(b.puttable([4,5]));
  AssertTrue(b.puttable([4,6]));
  AssertTrue(b.puttable([4,7]));
  AssertTrue(b.puttable([5,3]));
  AssertTrue(b.puttable([5,4]));
  AssertTrue(b.puttable([5,5]));
  AssertTrue(b.puttable([6,2]));
  AssertTrue(b.puttable([6,4]));
  AssertTrue(b.puttable([6,6]));
  AssertTrue(b.puttable([7,1]));
  AssertTrue(b.puttable([7,4]));
  AssertTrue(b.puttable([7,7]));
        AssertTrue(p2 = {
     [1,1] |-> <UNDEFINED>, [1,4] |-> <UNDEFINED>, [1,7] |->
<UNDEFINED>,
     [2,2] |-> <UNDEFINED>, [2,4] |-> <UNDEFINED>, [2,6] |->
<UNDEFINED>,
     [3,3] |-> <UNDEFINED>, [3,4] |-> <UNDEFINED>, [3,5] |->
<UNDEFINED>,
     [4,1] |-> <UNDEFINED>, [4,2] |-> <UNDEFINED>, [4,3] |->
<UNDEFINED>,
     [4,5] |-> <UNDEFINED>, [4,6] |-> <UNDEFINED>, [4,7] |->
<UNDEFINED>,
     [5,3] |-> <UNDEFINED>, [5,4] |-> <UNDEFINED>, [5,5] |->
<UNDEFINED>,
     [6,2] |-> <UNDEFINED>, [6,4] |-> <UNDEFINED>, [6,6] |->
<UNDEFINED>,
```

```
    [7,1] |-> <UNDEFINED>, [7,4] |-> <UNDEFINED>, [7,7] |->
<UNDEFINED>
   });
        return
);

public TestMovable : () ==> ()
TestMovable() ==
(
 dcl b : Board := new Board();
 b.put(<WHITE>, [1,1]);
       b.put(<BLACK>, [7,7]);
 b.put(<BLACK>, [2,4]);
 b.put(<BLACK>, [4,3]);
 b.put(<WHITE>, [4,6]);
       AssertTrue(b.movable(<WHITE>, [1,1], [1,4]));
       AssertTrue(b.movable(<WHITE>, [1,1], [4,1]));
       AssertFalse(b.movable(<WHITE>, [1,1], [1,7]));
       AssertFalse(b.movable(<WHITE>, [1,1], [7,1]));
       AssertFalse(b.movable(<BLACK>, [7,7], [3,4]));
       AssertTrue(b.movable(<BLACK>, [7,7], [7,4]));
 AssertTrue(b.movable(<BLACK>, [7,7], [4,7]));
 AssertFalse(b.movable(<BLACK>, [7,7], [1,1]));
 AssertTrue(b.removable(<WHITE>, [7,7]));
 AssertFalse(b.removable(<BLACK>, [7,7]));
 AssertTrue(b.movable(<BLACK>,[2,4], [2,2]));
       AssertTrue(b.movable(<BLACK>, [2,4], [1,4]));
 AssertTrue(b.movable(<BLACK>, [4,3], [4,2]));
 AssertTrue(b.movable(<BLACK>, [4,3], [3,3]));
 AssertTrue(b.movable(<WHITE>, [4,6], [2,6]));
 AssertTrue(b.movable(<WHITE>, [4,6], [4,5]));
       return
);

public TestMills : () ==> ()
TestMills() ==
(
 dcl b : Board := new Board();
 b.put(<WHITE>, [5,4]);
 b.put(<WHITE>, [6,4]);
 b.put(<WHITE>, [7,4]);
 b.put(<BLACK>, [1,7]);
 b.put(<BLACK>, [4,7]);
 b.put(<BLACK>, [7,7]);
 AssertTrue(b.hasMills(<WHITE>));
 AssertTrue(b.hasMills(<BLACK>));
 return
      );
      public TestMills2 : () ==> ()
TestMills2() ==
(
 dcl b : Board := new Board();
 b.put(<WHITE>, [4,1]);
 b.put(<WHITE>, [4,2]);
 b.put(<WHITE>, [4,3]);
 b.put(<WHITE>, [4,5]);
 b.put(<WHITE>, [4,6]);
 b.put(<WHITE>, [4,7]);
 b.put(<WHITE>, [3,3]);
 b.put(<BLACK>, [1,1]);
 b.put(<BLACK>, [1,4]);
```

```
 b.put(<BLACK>, [1,7]);
 AssertTrue(b.removable(<BLACK>, [3,3]));
 AssertFalse(b.removable(<BLACK>, [4,3]));
 AssertTrue(b.removable(<WHITE>, [1,1]));
 return
);

public TestAnotherPossibleMills : () ==> ()
TestAnotherPossibleMills() ==
(
 dcl b : Board := new Board();
 b.put(<BLACK>, [2,2]);
       b.put(<BLACK>, [2,4]);
 b.put(<BLACK>, [2,6]);
 b.put(<BLACK>, [3,3]);
 b.put(<BLACK>, [3,4]);
 b.put(<BLACK>, [3,5]);
 b.put(<BLACK>, [5,3]);
 b.put(<BLACK>, [5,4]);
 b.put(<BLACK>, [5,5]);
 b.put(<WHITE>, [6,2]);
 b.put(<WHITE>, [6,4]);
 b.put(<WHITE>, [6,6]);
 b.put(<BLACK>, [7,1]);
 b.put(<BLACK>, [7,4]);
 b.put(<BLACK>, [7,7]);
 b.put(<BLACK>, [1,1]);
 b.put(<WHITE>, [1,4]);
 AssertTrue(b.removable(<WHITE>, [1,1]));
 AssertTrue(b.removable(<BLACK>, [1,4]));
 return
);
      public TestAnotherPossibleMills2 : () ==> ()
TestAnotherPossibleMills2() ==
(
 dcl b : Board := new Board();
 b.put(<BLACK>, [2,2]);
 b.put(<BLACK>, [4,2]);
 b.put(<BLACK>, [6,2]);
 b.put(<BLACK>, [3,3]);
 b.put(<BLACK>, [4,3]);
 b.put(<BLACK>, [5,3]);
 b.put(<BLACK>, [3,5]);
 b.put(<BLACK>, [4,5]);
 b.put(<BLACK>, [5,5]);
 b.put(<WHITE>, [2,6]);
 b.put(<WHITE>, [4,6]);
 b.put(<WHITE>, [6,6]);
 b.put(<BLACK>, [1,1]);
 b.put(<BLACK>, [4,1]);
 b.put(<BLACK>, [7,1]);
 b.put(<BLACK>, [7,7]);
 b.put(<WHITE>, [5,4]);
 AssertTrue(b.removable(<WHITE>, [7,7]));
 AssertTrue(b.removable(<BLACK>, [5,4]));
 return
);
      public TestAnotherPossibleMills3 : () ==> ()
TestAnotherPossibleMills3() ==
(
 dcl b : Board := new Board();
```

```
  b.put(<BLACK>, [1,7]);
  b.put(<BLACK>, [4,7]);
  b.put(<BLACK>, [7,7]);
  b.put(<BLACK>, [1,4]);
  b.put(<BLACK>, [2,4]);
  b.put(<BLACK>, [3,4]);
  b.put(<WHITE>, [5,4]);
  b.put(<WHITE>, [6,4]);
  b.put(<WHITE>, [7,4]);
  b.put(<BLACK>, [1,1]);
  b.put(<WHITE>, [6,2]);
  AssertTrue(b.removable(<WHITE>, [1,1]));
  AssertTrue(b.removable(<BLACK>, [6,2]));
  return
       );

end TestBoard
```

Cobertura de Testes

name	***#calls***	***coverage***
TestBoard`TestMills	1	100%
TestBoard`AssertTrue	50	100%
TestBoard`TestMills2	1	100%
TestBoard`AssertFalse	9	100%
TestBoard`TestMovable	1	100%
TestBoard`TestBoardSetup	1	100%
TestBoard`TestAnotherPossibleMills	1	100%
TestBoard`TestAnotherPossibleMills2	1	100%
TestBoard`TestAnotherPossibleMills3	1	100%
total		***100%***

Classe TestGame

Definição em VDM++ da classe TestGame, para testar a classe Game:

```
class TestGame

operations
       -- operacao auxiliar, que tira partido do facto do
 -- interpretador parar quando se viola uma pre-condicao
 public AssertTrue : bool ==> ()
 AssertTrue(a) == return
      pre a;
```

```
public AssertFalse : bool ==> ()
AssertFalse(a) == return
pre not a;

public TestGameSetup : () ==> ()
TestGameSetup() ==
(
 dcl g : Game := new Game();
 dcl p : Player := new Player(<WHITE>);
 AssertTrue(g.getCurrentPieceType() = <WHITE>);
 AssertTrue(g.isPhaseOne());
 AssertFalse(g.isPhaseTwo());
 p := g.getCurrentPlayer();
 p.setUnplayedPieces(0);
 g.changePlayer();
 p := g.getCurrentPlayer();
 AssertTrue(g.isPhaseOne());
 AssertFalse(g.isPhaseTwo());
 p.setUnplayedPieces(0);
 AssertTrue(g.getCurrentPieceType() = <BLACK>);
 AssertFalse(g.isPhaseOne());
 AssertTrue(g.isPhaseTwo());
 g.changePlayer();
 AssertTrue(g.getCurrentPieceType() = <WHITE>);
 g.reset();
 AssertTrue(g.getCurrentPieceType() = <WHITE>);
 return
);

public TestPieceColocation : () ==> ()
TestPieceColocation() ==
(
 dcl g : Game := new Game();
 dcl b : Board := new Board();
 AssertTrue(g.puttable([1,1]));
 AssertFalse(g.put([1,1]));
 AssertFalse(g.put([1,4]));
 AssertTrue(g.put([1,7]));
 g.remove([1,7]);
 b := g.getBoard();
 AssertFalse(b.hasMills(<WHITE>));
 g.changePlayer();
 AssertFalse(g.put([7,7]));
 g.changePlayer();
 AssertTrue(g.removable([7,7]));
 return
);

public TestGameMovement : () ==> ()
TestGameMovement() ==
(
 dcl g : Game := new Game();
 dcl p : map Board`PieceType to nat;  AssertFalse(g.put([1,1]));
 AssertTrue(g.movable([1,1],[1,4]));
 g.changePlayer();
 AssertFalse(g.put([1,4]));
 g.changePlayer();
 AssertFalse(g.movable([1,1], [1,4]));
 AssertFalse(g.move([1,1], [4,1]));
       p := g.getUnplayedPieceCount();
```

```
        AssertTrue(p = {<BLACK> |-> 9, <WHITE> |-> 9} );
  AssertTrue(card dom p = 2);
  return
 );


end TestGame
```

Cobertura de Testes

name	*#calls*	*coverage*
TestGame`AssertTrue	13	100%
TestGame`AssertFalse	11	100%
TestGame`TestGameSetup	1	100%
TestGame`TestGameMovement	1	100%
TestGame`TestPieceColocation	1	100%
total		***100%***

Classe TestPlayer

Definição em VDM++ da classe TestPlayer, para testar a classe Player:

```
class TestPlayer

operations
 -- operacao auxiliar, que tira partido do facto do
 -- interpretador parar quando se viola uma pre-condicao
 public AssertTrue : bool ==> ()
 AssertTrue(a) == return
      pre a;

 public AssertFalse : bool ==> ()
 AssertFalse(a) == return
 pre not a;

 public TestInitialPieces : () ==> ()
 TestInitialPieces() ==
 (
  dcl p: Player := new Player(<WHITE>);
        AssertTrue(p.getTotalPieces() = 9);
  AssertTrue(p.getUnplayedPieces() = 9);
  p.decTotalPieces();
  AssertTrue(p.getTotalPieces() = 8);
  p.setTotalPieces(5);
  AssertTrue(p.getTotalPieces() = 5);
  p.setUnplayedPieces(3);
  AssertTrue(p.getUnplayedPieces() = 3);
  p.decUnplayedPieces();
  AssertTrue(p.getUnplayedPieces() = 2);
```

```
        return
      );


 public TestReset : () ==> ()
 TestReset() ==
 (
        dcl p : Player := new Player(<WHITE>);
  AssertTrue(p.getTotalPieces() = 9);
  AssertTrue(p.getUnplayedPieces() = 9);
        p.setTotalPieces(5);
        p.setUnplayedPieces(5);
  AssertTrue(p.getTotalPieces() = 5);
  AssertTrue(p.getUnplayedPieces() = 5);
  p.reset();
  AssertTrue(p.getTotalPieces() = Player`TOTAL_PIECES);
        AssertTrue(p.getUnplayedPieces() = Player`TOTAL_PIECES);
  return
 );

 public TestHasLost : () ==> ()
 TestHasLost() ==
 (
  dcl p : Player := new Player(<BLACK>);
        AssertTrue(p.getTotalPieces() = 9);
        p.setTotalPieces(2);
  AssertTrue(p.lost());
  return
 );

 public TestPieceType : () ==> ()
 TestPieceType() ==
 (
  dcl p : Player := new Player(<BLACK>);
  AssertFalse(p.getPieceType() = <UNDEFINED>);
  AssertTrue(p.getPieceType() = <BLACK>);
  p.setPieceType(<WHITE>);
        AssertTrue(p.getPieceType() = <WHITE>);
        return
 );


 public TestPlayerMills : () ==> ()
 TestPlayerMills() ==
 (
  dcl p : Player := new Player(<BLACK>);
  dcl m : set of set of seq of nat1 := {{[5,4] , [6,4], [7,4] }};
  AssertTrue(p.getPieceType() = <BLACK>);
  AssertTrue(p.getMills() = {});
  p.setMills(m);
  AssertTrue(p.getMills() = {{[5,4] , [6,4], [7,4] }});
  return
 );



end TestPlayer
```

Cobertura de Testes

name	***#calls***	***coverage***
TestPlayer`TestReset	1	100%
TestPlayer`AssertTrue	19	100%
TestPlayer`AssertFalse	1	100%
TestPlayer`TestHasLost	1	100%
TestPlayer`TestPieceType	1	100%
TestPlayer`TestPlayerMills	1	100%
TestPlayer`TestInitialPieces	1	100%
total		***100%***

Ficheiros de Teste

TestAnotherPossibleMills.arg

```
new TestBoard().TestAnotherPossibleMills()
```

TestAnotherPossibleMills.arg.exp

```
(no return value)
```

TestAnotherPossibleMills2.arg

```
new TestBoard().TestAnotherPossibleMills2()
```

TestAnotherPossibleMills2.arg.exp

```
(no return value)
```

TestAnotherPossibleMills3.arg

```
new TestBoard().TestAnotherPossibleMills3()
```

TestAnotherPossibleMills3.arg.exp

```
(no return value)
```

TestBoardSetup.arg

```
new TestBoard().TestBoardSetup()
```

TestBoardSetup.arg.exp

```
(no return value)
```

TestGameMovement.arg

```
new TestGame().TestGameMovement()
```

TestGameMovement.arg.exp

```
(no return value)
```

TestGameSetup.arg

```
new TestGame().TestGameSetup()
```

TestGameSetup.arg.exp

```
(no return value)
```

TestHasLost.arg

```
new TestPlayer().TestHasLost()
```

TestHasLost.arg.exp

```
(no return value)
```

TestInitialPieces.arg

```
new TestPlayer().TestInitialPieces()
```

TestInitialPieces.arg.exp

```
(no return value)
```

TestMills.arg

```
new TestBoard().TestMills()
```

TestMills.arg.exp

```
(no return value)
```

TestMills2.arg

```
new TestBoard().TestMills2()
```

TestMills2.arg.exp

```
(no return value)
```

TestMovable.arg

```
new TestBoard().TestMovable()
```

TestMovable.arg.exp

```
(no return value)
```

TestPieceColocation.arg

```
new TestGame().TestPieceColocation()
```

TestPieceColocation.arg.exp

```
(no return value)
```

TestPieceType.arg

```
new TestPlayer().TestPieceType()
```

TestPieceType.arg.exp

```
(no return value)
```

TestPlayerMills.arg

```
new TestPlayer().TestPlayerMills()
```

TestPlayerMills.arg.exp

```
(no return value)
```

TestReset.arg

```
new TestPlayer().TestReset()
```

TestReset.arg.exp

```
(no return value)
```

vdmloop.bat

```
@echo off
rem Runs a collection of VDM++ test examples
rem Assumes specification is in Word RTF files

set S1=Player.rtf
set S2=TestPlayer.rtf
set S3=Board.rtf
set S4=TestBoard.rtf
set S5=Game.rtf
set S6=TestGame.rtf

"D:\VDM++Toolbox_v8.1.1b\bin\vppde" -p -R vdm.tc %S1% %S2% %S3% %S4%
%S5% %S6%
for /R %%f in (*.arg) do call vdmtest "%%f"
```

vdmtest.bat

```
@echo off

rem Tests the date book specification for one test case (argument)
rem -- Output the argument to stdout (for redirect) and "con" (for
user feedback)
echo VDM Test: '%1' > con
echo VDM Test: '%1'

rem short names for specification files in Word RTF Format
set S1=Player.rtf
set S2=TestPlayer.rtf
set S3=Board.rtf
set S4=TestBoard.rtf
set S5=Game.rtf
set S6=TestGame.rtf


rem -- Calls the interpreter for this test case
"D:\VDM++Toolbox_v8.1.1b\bin\vppde" -i -D -I -P -Q -R vdm.tc -O %1.res
%1 %S1% %S2% %S3% %S4% %S5% %S6%
```

```
rem -- Check for difference between result of execution and expected
result.
if EXIST %1.exp fc /w %1.res %1.exp

:end
```

Matriz de Rastreabilidade dos Testes

Teste	Criação do Jogo	Colocação de Peças	Movimentação de Peças	Remoção de Peças	Formação de Mills	Mudança de Turno	Verificação do Vencedor
TestInitialPieces	X	X					
TestPieceType	X						
TestReset	X						
TestHasLost	X						X
TestPlayerMills	X				X		
TestGameSetup	X					X	
TestGameMovement	X		X			X	
TestPieceColocation	X			X		X	
TestBoardSetup	X	X		X			
TestMovable	X	X	X	X			
TestMills	X	X			X		
TestMills2	X	X		X	X		
TestAnotherPossibleMills	X	X		X	X		
TestAnotherPossibleMills2	X	X		X	X		
TestAnotherPossibleMills3	X	X		X	X		

Análise de Consistência do Modelo

Esta análise foi feita recorrendo à funcionalidade *Integrity Check* da ferramenta *VDM++ Toolbox*. Todos os possíveis problemas levantados pela execução dessa funcionalidade sobre a especificação realizada foram resolvidos, exceto nos casos em que a integridade é garantida por outros elementos da especificação. Na tabela abaixo, é possível ver os problemas não resolvidos, qual a causa deles, e a justificação pela qual o grupo não considera que os mesmos sejam problemas de integridade, ou a forma como a mesma é garantida.

Classe	**Tipo de Inconsistência Detetado**	**Causa/Local onde Ocorre**	**Solução encontrada para garantir não ocorre problema**
Player	*State Invariants*	Atribuição de valores.	Valores corretos são assegurados por pré-condições.
	Subtype	Atribuição de valores com invariantes.	Valores limite assegurados por pré-condições.
Board	*Sequence Application*	Utilização de índices para acesso a Sequências.	Pré-condições chamam a função *validCoords*, que não só assegura a validade das coordenadas como também que os índices usados são válidos.
	Post Conditions	Verificação do conteúdo da posição (funções *put/remove*).	Pós-condição verifica que o conteúdo da posição foi corretamente alterado.
	Map Enumeration	Atribuição de valores *hard-coded* em maps.	Valores são introduzidos manualmente, portanto garante-se que são os corretos.
	Map Application	Utilização de chaves para acesso a maps.	Pré-condições chamam a função *validCoords*, que assegura a validade das coordenadas/chaves usadas.
	Function Application	Parâmetros passados a funções.	Todas as funções asseguram a validade dos parâmetros que lhe são passados através das suas pré-condições.
Game	*Post Conditions*	Verificação dos valores de retorno de funções.	Pós-condições verificam se resultados gerados através de ciclos estão de acordo com o que seria esperado, a nível de conteúdo e estrutura.
	State Invariants	Atribuição de valores.	Valores são atribuídos explicitamente, caso contrário pré e pós-condições verificam se os valores são os esperados.
	Function Application	Parâmetros passados a funções.	Parâmetros corretos assegurados por pré-condições da função “pai”, e também das funções chamadas.

Geração Automática de Código

Após a especificação, recorreu-se à funcionalidade de geração de código Java da ferramenta *VDM++ Toolbox*, gerando assim automaticamente todas as funções relacionadas com a lógica de jogo. Esse código gerado foi integrado com uma interface gráfica desenvolvida com recurso a Swing, e exportado em formato *.jar*, formando um executável possível de encontrar na pasta enviada para a submissão do trabalho. Todo o código Java resultante foi também submetido. De seguida, podemos ver alguns *screenshots* da aplicação final:

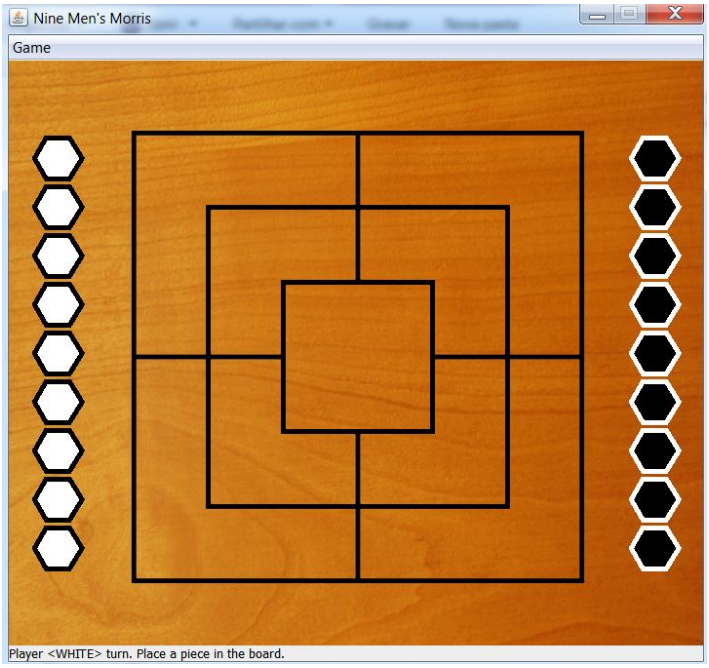


Fig.3 – Início de um novo jogo.

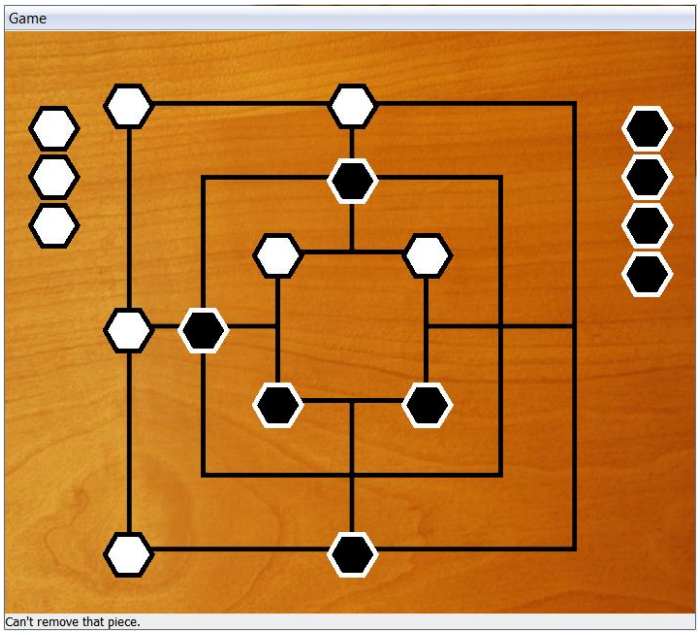


Fig.4 – Formação de um "mill" ainda na primeira fase do jogo.

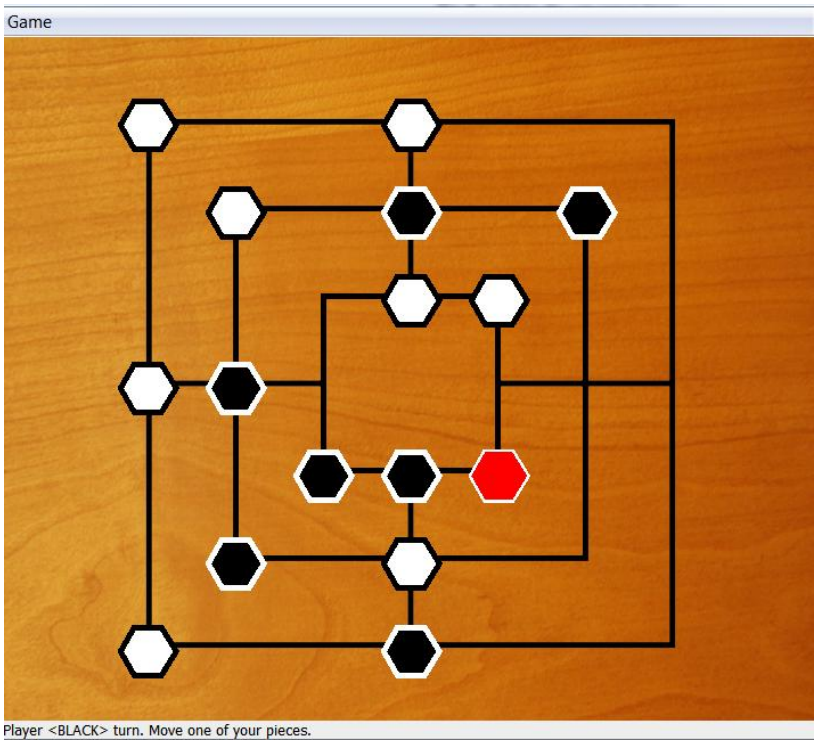


Fig.5 – Movimentação de uma peça.

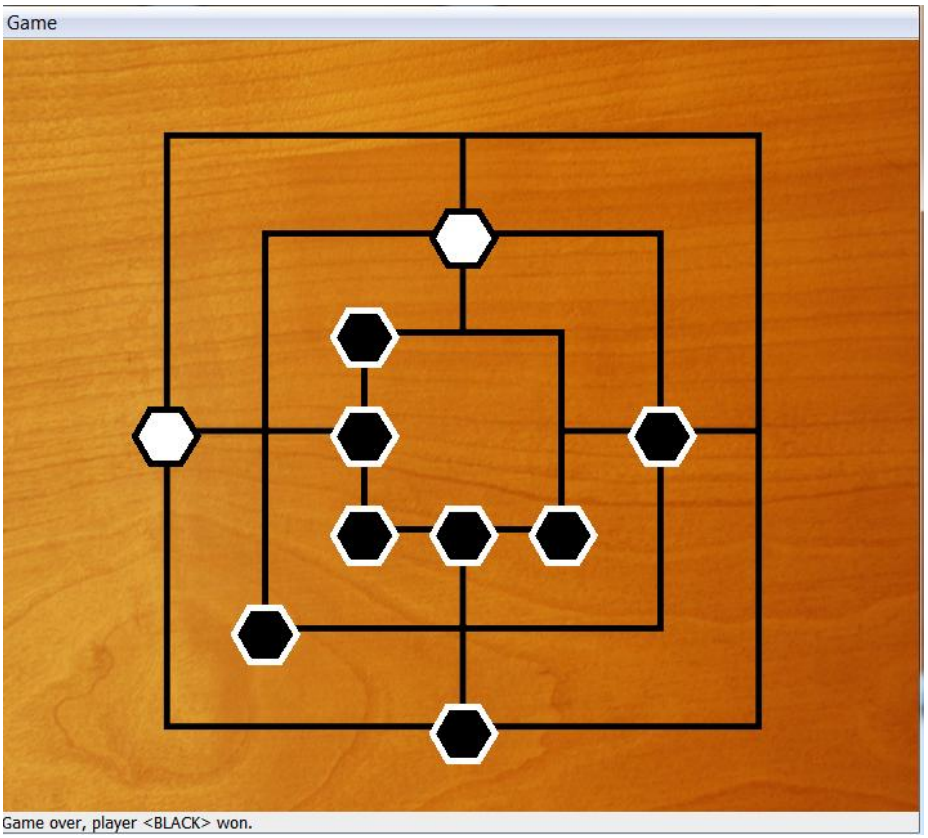


Fig.6 – Final do jogo.