Combate

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

João Gouveia - 201303988
João Mendonça - 201304605
Rui Gonçalves - 201201775

8 de Janeiro de 2017

Índice

1. Descrição informal do sistema e lista de requisitos

1.1 Descrição informal do sistema

Este trabalho foi realizado no âmbito da Unidade Curricular de Métodos Formais em Engenharia de Software, do 4º ano do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto.

O principal objetivo deste projeto é desenvolver, testar e documentar um modelo formal executável de um sistema de software de alta integridade, utilizando para isso a linguagem VDM++ e a ferramenta Overture para esse desenvolvimento. Desta forma, no final do trabalho, os estudantes serão dotados de uma capacidade de modelar sistemas formalmente em VDM++, e de demonstrar a consistência desse modelo.

Neste relatório iremos especificar e descrever o modelo formal, desenvolvido utilizando a linguagem referida acima, que implementa o jogo Combate, também conhecido como Stratego. Este terá a seguinte estrutura:

- Inicialmente, iremos descrever as regras do jogo Combate, seguidas da enumeração dos requisitos e restrições associadas, consideradas necessárias para a elaboração deste modelo.
- Seguidamente, iremos mostrar o Use Case Model, demonstrando os atores do jogo e os casos de uso, bem como o diagrama de classes, contendo as classes implementadas e as relações entre as mesmas, como forma de descrever a estrutura do modelo VDM++ implementado.
- Logo depois iremos transcrever o código VDM++, devidamente comentado e separado por classes.
- De seguida iremos descrever todos os testes e a sua implementação, bem como os resultados obtidos em cada teste.
- Depois será descrita a funcionalidade de análise de consistência do modelo apresentado.
- Por fim, antes da conclusão, em que iremos abordar alguns pontos a título de finalização, vamos descrever o processo de geração de código Java a partir da ferramenta Overture.

Combate é um jogo de tabuleiro lançado pela empresa Estrela em 1974, inicialmente com o nome Front, mas que foi alterado entretanto. Nos países de língua inglesa é conhecido sob o nome de Stratego. É jogado entre duas pessoas em que o objetivo é capturar a bandeira do adversário.

O tabuleiro é composto por 100 casas (10x10) e em que existem 2 lagos do mesmo tamanho, cada um ocupando 4 casas, em forma de quadrado. Estes ocupam as linhas 5 e 6

do tabuleiro, sendo que o primeiro lago ocupa a terceira e quarta colunas e o segundo lago ocupa a sétima e oitava colunas.

Cada jogador terá um exército à sua disposição de 39 peças para proteger a sua bandeira, e terá de colocar as 40 peças que tem (39 + bandeira) no seu campo inicial, que são as casas das linhas do tabuleiro separadas pelos lagos (linhas 1, 2, 3 e 4 para o primeiro exército e linhas 7, 8, 9 e 10 para o segundo).

As peças que cada jogador tem à disposição são de dois tipos: móveis e não móveis. As peças não móveis, que permanecem estáticas durante o jogo, podem ser:

- **1 Flag** (deverá ser escondida o melhor possível, de modo a que o adversário não a encontre)
- **6 Bombs** (derrotam qualquer outra, à exceção do Miner)

As peças móveis são usadas para invadir o território adversário, derrotar as suas peças e capturar a bandeira. Cada uma possui um determinado número, e quanto maior esse número, maior o poder dessa peça. Estas podem ser:

- **1 Spy** (S; quando ataca, tenta adivinhar qual é a peça adversária que está a atacar. Se acertar, derrota a peça adversária, senão é derrotado)
- **8 Scouts** (2 Pontos; podem mover-se mais que uma casa por vez)
- **5 Miners** (3; podem desarmar as bombas)
- **4 Sergeants** (4)
- **4 Lieutenants** (5)
- **4 Captains** (6)
- **3 Majors** (7)
- **2 Colonels** (8)
- **1 General** (9)
- **1 Marshall** (10)

Cada jogador apenas pode mover as peças do seu exército, que tem por objetivo, como já foi descrito acima, capturar a bandeira do exército inimigo, sendo também possível vencer deixando o adversário sem movimentos legais. À exceção do Scout e das peças não móveis, todas as outras apenas se podem movimentar uma casa de cada vez, na horizontal ou na vertical (nunca na diagonal).

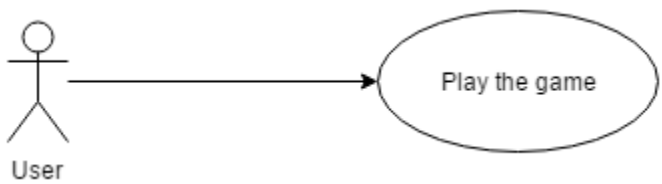
1.2 Lista de Requisitos

Id	Prioridade	Descrição
R1	Obrigatório	Cada jogador apenas pode efetuar uma jogada por turno.
R2	Obrigatório	Um jogador apenas pode mover peças da sua cor.
R3	Obrigatório	Um jogador apenas pode efetuar movimentos para casas que existam no tabuleiro.
R4	Obrigatório	Não é possível a um jogador movimentar peças para as casas que sejam um lago.
R5	Obrigatório	Um jogador não pode movimentar peças para casas ocupadas por peças da nossa cor.
R6	Obrigatório	Um jogador apenas pode efetuar movimentos na vertical ou na horizontal, nunca na diagonal.
R7	Obrigatório	Um jogador não pode movimentar peças do tipo Bomb e Flag.
R8	Obrigatório	Não é possível movimentar uma peça mais do que um espaço, à exceção da peça Scout.
R9	Obrigatório	Quando um jogador se movimentar para a casa onde se encontra uma bandeira adversária, este ganha o jogo.

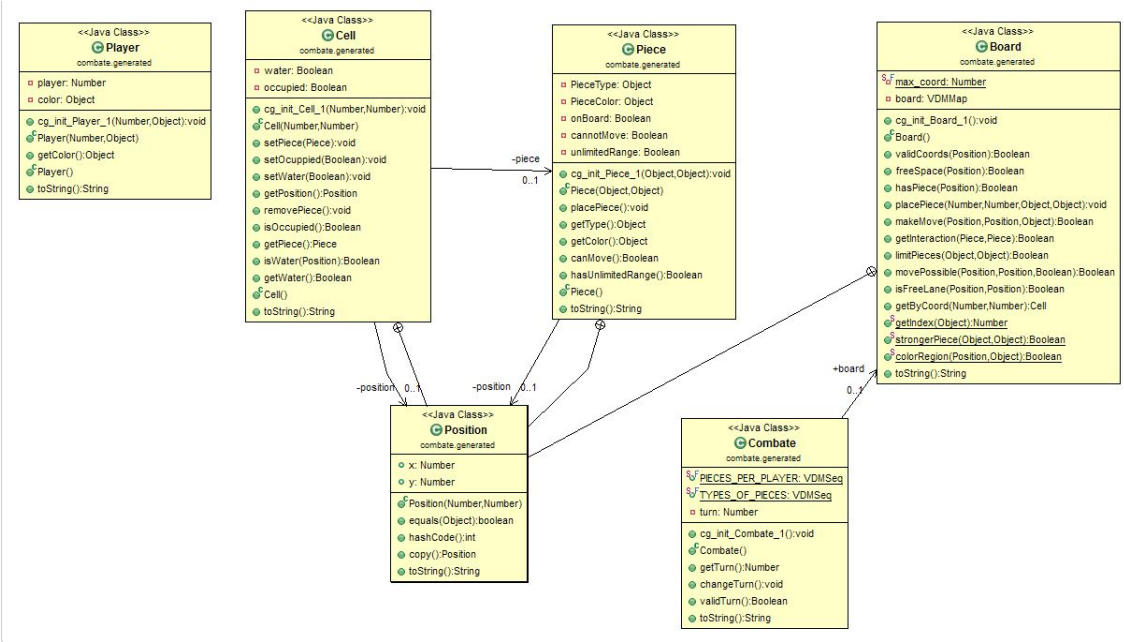
2. Modelo visual UML

2.1 Modelo de Casos de Uso

Como utilizador posso jogar o jogo com a interface criada pelos desenvolvedores (não existem outras opções)



2.2 Modelo de Classes



3. Modelo Formal VDM++

3.1 Classe Piece

```
class Piece
types
    public PType = <BOMB> | <MARSHALL> | <GENERAL> | <COLONEL> | <MAJOR> | <CAPTAIN> | <LIEUTENANT> | <SERGEANT> | <MINER> | <SCOUT> | <SPY> | <FLAG>;
    public PColor = <RED> | <BLUE>;

    public Position:: x:int y:int;
values
instance variables
    private PieceType : PType;
    private PieceColor : PColor;

    private position : Position;
    private onBoard : bool := false;
    private cannotMove : bool;
    private unlimitedRange : bool;
operations
    -- Construtor
    public Piece : PType*PColor ==> Piece
      Piece(type,color) ==
      (
        PieceType := type;
        PieceColor := color;

        position := mk_Position(-1,-1);
        cannotMove := PieceType = <BOMB> or PieceType = <FLAG>;
        unlimitedRange := PieceType = <SCOUT>;

        self.placePiece();
      )
      pre type <> nil and color <> nil;

    -- Função invocada no construtor que coloca a peça no tabuleiro
    public placePiece : () ==> ()
      placePiece() ==
      (
        onBoard := true;
      )
      post self.onBoard = true;

    -- Retorna o tipo da peça
    public pure getType : () ==> PType
      getType() == return PieceType;


    -- Retorna a cor da peça
    public pure getColor : () ==> PColor
      getColor() == return PieceColor;

    -- Verdadeiro se a peça se poder mexer, false otherwise
    public pure canMove : () ==> bool
      canMove() == return not cannotMove;

    -- Verdadeiro se a peça se poder mexer um número de casas maior que 1, false otherwise
    public pure hasUnlimitedRange : () ==> bool
      hasUnlimitedRange() == return unlimitedRange;

functions
traces
end Piece
```

3.2 Classe Cell

```
class Cell
types
  public Position :: x:nat y:nat;
values
instance variables
  private position : Position;
  private piece :  [Piece] := nil;
  private water : bool := false;
  private occupied : bool := false;

  inv occupied => piece <> nil;
  inv water => piece = nil and occupied = false;
operations
  -- Construtor
  public Cell : nat * nat ==> Cell
    Cell(x,y) ==
    (
      position := mk_Position(x,y);
      water := isWater(position);
    );

  -- Coloca uma peça numa casa
  public setPiece : Piece ==> ()
    setPiece(pie) ==
    (
      atomic(
        piece := pie;
        occupied := true;
        )
    )
    pre occupied = false and piece = nil and water = false
    post occupied = true and piece <> nil;

  -- Remove uma peça do tabuleiro
  public removePiece : () ==> ()
    removePiece() ==
    (
      atomic(
        piece := nil;
        occupied := false;
      )
    )
    pre occupied = true and piece <> nil and water = false
    post occupied = false and piece = nil;
```

```
44      post occupied = false and piece = nil;
45
46    -- Retorna a posição da casa
47    public getPosition : () ==> Position
48      getPosition() ==
49      (
50        return position;
51      );
52
53    -- Verdadeiro se a casa estiver ocupada ou for um lago, false otherwise
54    public pure isOccupied : () ==> bool
55      isOccupied() == return occupied or water;
56
57    -- Retorna a peça que está na casa
58    public pure getPiece : () ==> [Piece]
59      getPiece() == return piece;
60
61    -- Define as casas que são lagos
62    public pure isWater : Position ==> bool
63      isWater(pos) ==
64      (
65        return (pos.x = 2 or pos.x = 3 or pos.x = 6 or pos.x = 7) and (pos.y = 4 or pos.y = 5);
66      );
67
68    -- Verdadeiro se a casa for um lago, false otherwise
69    public pure getWater : () ==> bool
70      getWater() == return water;
71
72
73  functions
74  traces
75  end Cell
```

3.2 Classe Player

```
class Player
types
values
  public TOTAL_PIECES : nat = 40;
instance variables
  private player : nat := 0;

  private totalPieces : nat := TOTAL_PIECES;
  private unplayedPieces : nat := TOTAL_PIECES;

  private pieces : set of Piece := {};

  inv card pieces <= TOTAL_PIECES;

  inv player = 0 or player = 1;
operations
    -- Cosntrutor
    public Player : nat ==> Player
      Player(pl) ==
      (
        player := pl;
      )
      pre pl = 0 or pl = 1;

    -- O jogador adiciona uma peça sua ao tabuleiro
    public addPiece : Piece ==> ()
      addPiece(p) ==
      (
        pieces := {p}union pieces;
      )
      pre card pieces <= TOTAL_PIECES
      post card pieces <=TOTAL_PIECES;

    -- O jogador elimina uma peça sua do tabuleiro
    public eliminatePiece : Piece ==> ()
      eliminatePiece(p) ==
      (
        pieces := pieces \ {p};
      );

functions
traces
end Player
```

3.2 Classe Board

```
class Board

types
    public Position:: x:nat y:nat;

values
    private max_coord : nat = 9 --Board is 10*10

instance variables
  private board : map Position to Cell := {|->};

operations
-- Creates the board
public Board : () ==> Board
    Board() ==
      (
        for i=0 to max_coord by 1 do
          (
            for j=0 to max_coord by 1 do
              (
                board := board ++ {mk_Position(i,j)|-> new Cell(i,j)};
              )
          )
      );

  -- Returns the cell in the given coordinates
  public getByCoords : nat*nat ==>Cell
    getByCoords(i,j)==(
      return board(mk_Position(i,j))
    );

  -- Returns a position
  public createPosition:nat*nat ==>Position
    createPosition(i,j) ==
    (
      return mk_Position(i,j);
    );

  -- Acording to given max_coord, checks if given position is valid
  public pure validCoords : Position ==> bool
    validCoords(position) ==
    (
      return position.x <= max_coord and position.y <= max_coord;
    );
```

```
  ,,
45
46  --Checks if a given position is occupied
47  public pure freeSpace : Position ==> bool
48    freeSpace(position) ==
49    (
50      return not board(position).isOccupied();
51    )
52    pre validCoords(position);
53
54  -- Returns the piece that is in a given position
55  public pure hasPiece : Position ==> bool
56    hasPiece(position) == return board(position).getPiece() <> nil;
57
58  -- Creates and places a piece in a cell that is in the given coordinates
59  public placePiece : nat*nat*Piece`PType*Piece`PColor ==> ()
60    placePiece(i,j,type,color) ==
61    (
62      dcl piece : Piece := new Piece(type,color);
63      board(mk_Position(i,j)).setPiece(piece);
64    )
65    pre freeSpace(mk_Position(i,j)) and colorRegion(mk_Position(i,j),color) and limitPieces(type,color)
66    post hasPiece(mk_Position(i,j));
67
68  -- Makes the moves
69  public makeMove : Position*Position*Piece`PColor ==> bool
70    makeMove(origin,destination,color) ==
71    (
72      dcl pieceOrigin : [Piece] := board(origin).getPiece();
73      dcl pieceDestination : [Piece] := board(destination).getPiece();
74
75      if movePossible(origin,destination,pieceOrigin.hasUnlimitedRange())
76      then (
77        if pieceDestination = nil -- Piece can be moved, anything there
78          then (
79              board(origin).removePiece();
80              board(destination).setPiece(pieceOrigin);
81              return true;
82            )
83            else (
84              if  pieceOrigin.getColor() = pieceDestination.getColor()
85                then return false
86              else (
87                if pieceOrigin.getType() = pieceDestination.getType()
88                  then (
```

```
                then (
                  board(origin).removePiece();
                  board(destination).removePiece();
                  return true;
                )
                else (
                 if getInteraction(pieceOrigin,pieceDestination)
                  then (
                    board(origin).removePiece();
                    board(destination).removePiece();
                    board(destination).setPiece(pieceOrigin);
                    return true;
                  )
                  else (
                    board(origin).removePiece();
                    return true;
                  )
                )
            )
          )
    );
    return false;
  )pre validCoords(origin) and validCoords(destination) and hasPiece(origin) and board(origin).getPiece().getColor() = color and board(origin).getPiece().canMove()
  post hasPiece(origin) or not hasPiece(origin);

 --Returns true if 1st (origin) piece stronger, false otherwise
public pure getInteraction : Piece * Piece ==> bool
  getInteraction(origin,destination) == (
    dcl originType : Piece`PType := origin.getType();
    dcl destinationType : Piece`PType := destination.getType();
    cases originType :
    <MINER> -> return destinationType = <BOMB>,
    <SPY> -> return destinationType = <MARSHALL>,
    others -> strongerPiece(originType,destinationType)
    end
  );

 -- Verdadeiro se o número máximo de peças não foi ultrapassado, false otherwise
public pure limitPieces : Piece`PType*Piece`PColor ==> bool
  limitPieces(type,color) ==
  (
    dcl pieces : nat := 0;
    dcl index : nat := getIndex(type);


      for all cell in set rng board do
        (if cell.getPiece() <> nil
          then
            (if (cell.getPiece().getType() = type and cell.getPiece().getColor() = color)
              then pieces := pieces + 1));

      return pieces < Combate`PIECES_PER_PLAYER(index);
    );

  -- Checks if given move is possible given origin, destination and range
 public pure movePossible : Position*Position*bool ==> bool
    movePossible(origin,destination,range) ==
    (
      if range
        then return isFreeLane(origin,destination)
        else if origin.x = destination.x
          then return abs(origin.y - destination.y) = 1
          else return abs(origin.x - destination.x) = 1;
    )
    pre origin.x = destination.x or origin.y =destination.y;

  -- Checks if lane is free
 public pure isFreeLane : Position*Position ==> bool
    isFreeLane(origin,destination) ==
    (
      if origin.x = destination.x
        then
          for i = origin.y to destination.y by (if origin.y >= destination.y then -1 else 1) do
            (if (i <> destination.y and i <> origin.y) then if board(mk_Position(destination.x,i)).isOccupied() then return false;)
        else
          for i = origin.x to destination.x by (if origin.x >= destination.x then -1 else 1) do
            (if (i <> destination.x and i <> origin.x) then if board(mk_Position(i,destination.y)).isOccupied() then return false;);

        return true;
    );

 functions


  -- Get index of type of piece
  public getIndex : Piece`PType -> nat
    getIndex(type) ==
    (
      [i | i in set inds Combate`TYPES_OF_PIECES & Combate`TYPES_OF_PIECES(i) = type](1)
    );

  -- Verdadeiro se a força de um tipo for maior que a força de outro tipo
  public strongerPiece : Piece`PType*Piece`PType -> bool
    strongerPiece(originType,destinationType) == (
       getIndex(originType) > getIndex(destinationType)
    );

  -- Restringe o local onde se podem colocar as peças
  public colorRegion : Position*Piece`PColor -> bool
    colorRegion(position, color) ==
      if color = <BLUE> then position.y < 4 else position.y > 5;


traces
end Board
```

3.2 Classe Combate

```
class Combate
types

values
    public PIECES_PER_PLAYER : seq of nat1 = [6,1,1,2,3,4,4,4,5,8,1,1];
    public TYPES_OF_PIECES : seq of Piece`PType = [<BOMB>, <MARSHALL>, <GENERAL>, <COLONEL>, <MAJOR>, <CAPTAIN>, <LIEUTENANT>, <SERGEANT>, <MINER>, <SCOUT>, <SPY>, <FLAG>];
instance variables
    private board : Board;
    private turn : nat := 0;

    -- Invariables
    inv turn = 1 or turn = 0;
operations
    -- Construtor
    public Combate : () ==> Combate
    Combate() ==
    (
      board := new Board();
    )
    post self.validTurn();

    -- Retorna o turno atual
    public pure getTurn : () ==> nat
      getTurn() == return turn;

    -- Efetua a mudança de turno
    public changeTurn : () ==> ()
      changeTurn() ==
      (
        if turn = 0 then turn := 1 else turn := 0;
      )
    pre self.validTurn()
    post self.validTurn();

    -- Verdadeiro se o turno for válido
    public pure validTurn : () ==> bool
      validTurn() == return turn = 0 or turn = 1;

functions
traces
end Combate
```

4. Validação do Modelo

4.1 Classe CombateTest

```
class CombateTests

operations
    public static main : () ==> ()
        main() ==
        (
            new CombateTests().allTests();
        );

    public assertTrue : bool ==> ()
        assertTrue(arg) == return
        pre arg;

    -- Corre todos os testes implementados
    public allTests : () ==> ()
        allTests() ==
        (
            -- Testar se pode mover Bombs e Flags
            test_move_bombsAndFlags();

            -- Testar se um tipo é mais forte que outro
            test_strength();

            -- Testar se pode mover Captains (por exemplo)
            test_move_captain();

            -- Testar se um Scout pode mover mais que uma casa
            test_move_scout();

            -- Testar se pode mover Captains (por exemplo)
            test_move_miner();

            -- Testar se um casa é lago ou não
            test_water();

            -- Testar as funções do jogador
            test_player();

            -- Testar a alteração de turnos
            test_change_turns();
        );

    public test_move_bombsAndFlags : () ==> ()
        test_move_bombsAndFlags() ==
```

```
45          (
46              dcl bomb: Piece := new Piece(<BOMB>, <BLUE>);
47              dcl flag: Piece := new Piece(<FLAG>, <BLUE>);
48              dcl cell1: Cell := new Cell(0, 3);
49              dcl cell2: Cell := new Cell(7, 0);
50              cell1.setPiece(bomb);
51              cell2.setPiece(flag);
52
53              assertTrue(bomb.canMove() = false);
54              assertTrue(flag.canMove() = false);
55
56              assertTrue(flag.getType() = <FLAG>);
57              assertTrue(bomb.getColor() = <BLUE>);
58          );
59
60      public test_strength : () ==> ()
61          test_strength() ==
62          (
63              dcl b : Board := new Board();
64
65              dcl lieutenantBlue: Piece := new Piece(<LIEUTENANT>, <BLUE>);
66              dcl captainRed: Piece := new Piece(<CAPTAIN>, <RED>);
67
68              dcl minerBlue: Piece := new Piece(<MINER>, <BLUE>);
69              dcl bombRed: Piece := new Piece(<BOMB>, <RED>);
70
71              dcl spyBlue: Piece := new Piece(<SPY>, <BLUE>);
72              dcl marshallRed: Piece := new Piece(<MARSHALL>, <RED>);
73
74              assertTrue(b.getInteraction(captainRed, lieutenantBlue) = false);
75              assertTrue(b.getInteraction(minerBlue, bombRed));
76              assertTrue(b.getInteraction(spyBlue, marshallRed));
77          );
78
79      public test_move_captain : () ==> ()
80          test_move_captain() ==
81          (
82              dcl b : Board := new Board();
83
84              dcl lieutenantBlue: Piece := new Piece(<LIEUTENANT>, <BLUE>);
85              dcl captainRed: Piece := new Piece(<CAPTAIN>, <RED>);
86              dcl sergeantRed: Piece := new Piece(<CAPTAIN>, <RED>);
87
88              b.placePiece(4, 3, <CAPTAIN>, <BLUE>);
```

```
89          b.placePiece(0, 9, <FLAG>, <RED>);
90          b.getByCoords(5, 4).setPiece(lieutenantBlue);
91          b.getByCoords(4, 5).setPiece(captainRed);
92          b.getByCoords(5, 5).setPiece(sergeantRed);
93
94          assertTrue(b.limitPieces(<CAPTAIN>, <BLUE>));
95
96          -- Movimento para uma casa que não possui uma peça
97          assertTrue(b.makeMove(b.createPosition(4, 3), b.createPosition(4, 4), <BLUE>));
98
99          -- Movimento para uma casa que possui uma peça da mesma cor
100         assertTrue(b.makeMove(b.createPosition(4, 4), b.createPosition(5, 4), <BLUE>) = false);
101
102         -- Movimento para uma casa que possui uma peça da cor adversária inferior
103         --assertTrue(b.makeMove(b.createPosition(5, 4), b.createPosition(5, 5), <BLUE>));
104
105         -- Movimento para uma casa que possui uma peça da cor adversária igual
106         assertTrue(b.makeMove(b.createPosition(4, 4), b.createPosition(4, 5), <BLUE>));
107
108         -- Movimento para uma casa que possui uma peça da cor adversária
109         --assertTrue(b.makeMove(b.createPosition(4, 4), b.createPosition(4, 5), <BLUE>));
110     );
111
112 public test_move_scout : () ==> ()
113     test_move_scout() ==
114     (
115         dcl b : Board := new Board();
116
117         dcl scoutBlue: Piece := new Piece(<SCOUT>, <BLUE>);
118         b.getByCoords(8, 0).setPiece(scoutBlue);
119
120         assertTrue(b.movePossible(b.createPosition(8, 0), b.createPosition(8, 3), scoutBlue.hasUnlimitedRange()));
121         assertTrue(b.movePossible(b.createPosition(8, 0), b.createPosition(6, 0), scoutBlue.hasUnlimitedRange()));
122
123         b.placePiece(8, 2, <MAJOR>, <BLUE>);
124         b.placePiece(7, 0, <COLONEL>, <BLUE>);
125
126         assertTrue(b.makeMove(b.createPosition(8, 0), b.createPosition(8, 3), <BLUE>) = false);
127         assertTrue(b.movePossible(b.createPosition(8, 0), b.createPosition(6, 0), scoutBlue.hasUnlimitedRange()) = false);
128
129         assertTrue(b.movePossible(b.createPosition(8, 0), b.createPosition(8, 0), scoutBlue.hasUnlimitedRange()));
130         assertTrue(b.movePossible(b.createPosition(8, 0), b.createPosition(9, 0), scoutBlue.hasUnlimitedRange()));
131     );
132
```

```
132
133     public test_move_miner : () ==> ()
134         test_move_miner() ==
135         (
136             dcl b : Board := new Board();
137 
138             dcl minerBlue: Piece := new Piece(<MINER>, <BLUE>);
139             dcl bombRed: Piece := new Piece(<BOMB>, <RED>);
140             dcl sergeantRed: Piece := new Piece(<SERGEANT>, <RED>);
141             b.getByCoords(0, 5).setPiece(minerBlue);
142             b.getByCoords(0, 6).setPiece(bombRed);
143             b.getByCoords(1, 6).setPiece(sergeantRed);
144 
145             assertTrue(b.makeMove(b.createPosition(0, 5), b.createPosition(0, 6), <BLUE>));
146             assertTrue(b.makeMove(b.createPosition(0, 6), b.createPosition(1, 6), <BLUE>));
147             --assertTrue(b.makeMove(b.createPosition(8, 0), b.createPosition(6, 0), <BLUE>));
148         );
149 
150     public test_water : () ==> ()
151         test_water() ==
152         (
153             dcl b : Board := new Board();
154 
155             assertTrue(b.getByCoords(2, 4).isWater(b.getByCoords(2, 4).getPosition()));
156             assertTrue(b.getByCoords(2, 4).getWater());
157         );
158 
159     public test_player : () ==> ()
160         test_player() ==
161         (
162             dcl player1 : Player := new Player(0);
163             dcl player2 : Player := new Player(1);
164 
165             dcl bombBlue : Piece := new Piece(<BOMB>, <BLUE>);
166 
167             player1.addPiece(bombBlue);
168             player1.eliminatePiece(bombBlue);
169 
170             --assertTrue(player2.getPieces() = nil);
171         );
172 
173     public test_change_turns : () ==> ()
174         test_change_turns() ==
175         (
176             dcl combate : Combate := new Combate();
177 
178             combate.changeTurn();
179 
180             assertTrue(combate.getTurn() = 1);
181 
182             combate.changeTurn();
183 
184             assertTrue(combate.getTurn() = 0);
185         );
186 
187 
188 end CombateTests
189 
```

4.2 Cobertura dos testes

- Classe Piece:

```
class Piece
types
    public PType = <BOMB> | <MARSHALL> | <GENERAL> | <COLONEL> | <MAJOR> | <CAPTAIN> | <LIEUTENANT> | <SERGEANT> | <MINER> | <SCOUT> | <SPY> | <FLAG>;
    public PColor = <RED> | <BLUE>;

    public Position:: x:int y:int;
values
instance variables
    private PieceType : PType;
    private PieceColor : PColor;

    private position : Position;
    private onBoard : bool := false;
    private cannotMove : bool;
    private unlimitedRange : bool;
operations
    -- Construtor
    public Piece : PType*PColor ==> Piece
      Piece(type,color) ==
      (
        PieceType := type;
        PieceColor := color;

        position := mk_Position(-1,-1);
        cannotMove := PieceType = <BOMB> or PieceType = <FLAG>;
        unlimitedRange := PieceType = <SCOUT>;

        self.placePiece();
      )
      pre type <> nil and color <> nil;

    -- Função invocada no construtor que coloca a peça no tabuleiro
    public placePiece : () ==> ()
      placePiece() ==
      (
        onBoard := true;
      )
      post self.onBoard = true;

    -- Retorna o tipo da peça
    public pure getType : () ==> PType
      getType() == return PieceType;

    -- Retorna a cor da peça
    public pure getColor : () ==> PColor
      getColor() == return PieceColor;

    -- Verdadeiro se a peça se poder mexer, false otherwise
    public pure canMove : () ==> bool
      canMove() == return not cannotMove;

    -- Verdadeiro se a peça se poder mexer um número de casas maior que 1, false otherwise
    public pure hasUnlimitedRange : () ==> bool
      hasUnlimitedRange() == return unlimitedRange;

functions
traces
end Piece
```

- Classe Cell:

```
class Cell
types
  public Position :: x:nat y:nat;
values
instance variables
  private position : Position;
  private piece :  [Piece] := nil;
  private water : bool := false;
  private occupied : bool := false;

  inv occupied => piece <> nil;
  inv water => piece = nil and occupied = false;
operations
  -- Construtor
  public Cell : nat * nat ==> Cell
    Cell(x,y) ==
    (
      position := mk_Position(x,y);
      water := isWater(position);
    );

  -- Coloca uma peça numa casa
  public setPiece : Piece ==> ()
    setPiece(pie) ==
    (
      atomic(
        piece := pie;
        occupied := true;
        )
    )
    pre occupied = false and piece = nil and water = false
    post occupied = true and piece <> nil;

  -- Remove uma peça do tabuleiro
  public removePiece : () ==> ()
    removePiece() ==
    (
      atomic(
        piece := nil;
        occupied := false;
      )
    )
    pre occupied = true and piece <> nil and water = false
    post occupied = false and piece = nil;
```

```
45
46  -- Retorna a posição da casa
47  public getPosition : () ==> Position
48    getPosition() ==
49    (
50      return position;
51    );
52
53  -- Verdadeiro se a casa estiver ocupada ou for um lago, false otherwise
54  public pure isOccupied : () ==> bool
55    isOccupied() == return occupied or water;
56
57  -- Retorna a peça que está na casa
58  public pure getPiece : () ==> [Piece]
59    getPiece() == return piece;
60
61  -- Define as casas que são lagos
62  public pure isWater : Position ==> bool
63    isWater(pos) ==
64    (
65      return (pos.x = 2 or pos.x = 3 or pos.x = 6 or pos.x = 7) and (pos.y = 4 or pos.y = 5);
66    );
67
68  -- Verdadeiro se a casa for um lago, false otherwise
69  public pure getWater : () ==> bool
70    getWater() == return water;
71
72
73 functions
74 traces
75 end Cell
```

- Classe Player:

```
class Player
types
values
  public TOTAL_PIECES : nat = 40;
instance variables
  private player : nat := 0;

  private totalPieces : nat := TOTAL_PIECES;
  private unplayedPieces : nat := TOTAL_PIECES;

  private pieces : set of Piece := {};

  inv card pieces <= TOTAL_PIECES;

  inv player = 0 or player = 1;
operations
    -- Cosntrutor
    public Player : nat ==> Player
      Player(pl) ==
      (
        player := pl;
      )
      pre pl = 0 or pl = 1;

    -- O jogador adiciona uma peça sua ao tabuleiro
    public addPiece : Piece ==> ()
      addPiece(p) ==
      (
        pieces := {p}union pieces;
      )
      pre card pieces <= TOTAL_PIECES
      post card pieces <=TOTAL_PIECES;

    -- O jogador elimina uma peça sua do tabuleiro
    public eliminatePiece : Piece ==> ()
      eliminatePiece(p) ==
      (
        pieces := pieces \ {p};
      );

functions
traces
end Player
```

- Classe Board:

```
class Board

types
    public Position:: x:nat y:nat;

values
    private max_coord : nat = 9 --Board is 10*10

instance variables
  private board : map Position to Cell := {|->};

operations
-- Creates the board
public Board : () ==> Board
    Board() ==
      (
        for i=0 to max_coord by 1 do
          (
            for j=0 to max_coord by 1 do
              (
                board := board ++ {mk_Position(i,j)|-> new Cell(i,j)};
              )
          )
      );

  -- Returns the cell in the given coordinates
  public getByCoords : nat*nat ==>Cell
    getByCoords(i,j)==(
      return board(mk_Position(i,j))
    );

  -- Returns a position
  public createPosition:nat*nat ==>Position
    createPosition(i,j) ==
    (
      return mk_Position(i,j);
    );

  -- Acording to given max_coord, checks if given position is valid
  public pure validCoords : Position ==> bool
    validCoords(position) ==
    (
      return position.x <= max_coord and position.y <= max_coord;
    );


  --Checks if a given position is occupied
  public pure freeSpace : Position ==> bool
    freeSpace(position) ==
    (
      return not board(position).isOccupied();
    )
    pre validCoords(position);

  -- Returns the piece that is in a given position
  public pure hasPiece : Position ==> bool
    hasPiece(position) == return board(position).getPiece() <> nil;

  -- Creates and places a piece in a cell that is in the given coordinates
  public placePiece : nat*nat*Piece`PType*Piece`PColor ==> ()
    placePiece(i,j,type,color) ==
    (
      dcl piece : Piece := new Piece(type,color);
      board(mk_Position(i,j)).setPiece(piece);
    )
    pre freeSpace(mk_Position(i,j)) and colorRegion(mk_Position(i,j),color) and limitPieces(type,color)
    post hasPiece(mk_Position(i,j));

  -- Makes the moves
  public makeMove : Position*Position*Piece`PColor ==> bool
    makeMove(origin,destination,color) ==
    (
      dcl pieceOrigin : [Piece] := board(origin).getPiece();
      dcl pieceDestination : [Piece] := board(destination).getPiece();

      if movePossible(origin,destination,pieceOrigin.hasUnlimitedRange())
      then (
        if pieceDestination = nil -- Piece can be moved, anything there
          then (
              board(origin).removePiece();
              board(destination).setPiece(pieceOrigin);
              return true;
            )
            else (
              if  pieceOrigin.getColor() = pieceDestination.getColor()
                then return false
              else (
                if pieceOrigin.getType() = pieceDestination.getType()
                  then (
```

```
                board(origin).removePiece();
                board(destination).removePiece();
                return true;
              )
              else (
               if getInteraction(pieceOrigin,pieceDestination)
                then (
                  board(origin).removePiece();
                  board(destination).removePiece();
                  board(destination).setPiece(pieceOrigin);
                  return true;
                )
                else (
                  board(origin).removePiece();
                  return true;
                )
              )
            )
          )
    );
    return false;
  )pre validCoords(origin) and validCoords(destination) and hasPiece(origin) and board(origin).getPiece().getColor() = color and board(origin).getPiece().canMove()
  post hasPiece(origin) or not hasPiece(origin);

--Returns true if 1st (origin) piece stronger, false otherwise
public pure getInteraction : Piece * Piece ==> bool
  getInteraction(origin,destination) == (
    dcl originType : Piece`PType := origin.getType();
    dcl destinationType : Piece`PType := destination.getType();
    cases originType :
    <MINER> -> return destinationType = <BOMB>,
    <SPY> -> return destinationType = <MARSHALL>,
    others -> strongerPiece(originType,destinationType)
    end
  );

-- Verdadeiro se o número máximo de peças não foi ultrapassado, false otherwise
public pure limitPieces : Piece`PType*Piece`PColor ==> bool
  limitPieces(type,color) ==
  (
    dcl pieces : nat := 0;
    dcl index : nat := getIndex(type);

    for all cell in set rng board do

        then
          (if (cell.getPiece().getType() = type and cell.getPiece().getColor() = color)
            then pieces := pieces + 1));

    return pieces < Combate`PIECES_PER_PLAYER(index);
  );

-- Checks if given move is possible given origin, destination and range
public pure movePossible : Position*Position*bool ==> bool
  movePossible(origin,destination,range) ==
  (
    if range
      then return isFreeLane(origin,destination)
      else if origin.x = destination.x
        then return abs(origin.y - destination.y) = 1
        else return abs(origin.x - destination.x) = 1;
  )
  pre origin.x = destination.x or origin.y =destination.y;

-- Checks if lane is free
public pure isFreeLane : Position*Position ==> bool
  isFreeLane(origin,destination) ==
  (
    if origin.x = destination.x
      then
        for i = origin.y to destination.y by (if origin.y >= destination.y then -1 else 1) do
          (if (i <> destination.y and i <> origin.y) then if board(mk_Position(destination.x,i)).isOccupied() then return false;)
      else
        for i = origin.x to destination.x by (if origin.x >= destination.x then -1 else 1) do
          (if (i <> destination.x and i <> origin.x) then if board(mk_Position(i,destination.y)).isOccupied() then return false;);

      return true;
  );

functions

-- Get index of type of piece
public getIndex : Piece`PType -> nat
  getIndex(type) ==
  (
    [i | i in set inds Combate`TYPES_OF_PIECES & Combate`TYPES_OF_PIECES(i) = type](1)
  );


 -- Verdadeiro se a força de um tipo for maior que a força de outro tipo
 public strongerPiece : Piece`PType*Piece`PType -> bool
   strongerPiece(originType,destinationType) == (
      getIndex(originType) > getIndex(destinationType)
   );

 -- Restringe o local onde se podem colocar as peças
 public colorRegion : Position*Piece`PColor -> bool
   colorRegion(position, color) ==
     if color = <BLUE> then position.y < 4 else position.y > 5;


traces
end Board
```

- Classe Combate:

```
class Combate
types

values
    public PIECES_PER_PLAYER : seq of nat1 = [6,1,1,2,3,4,4,4,5,8,1,1];
    public TYPES_OF_PIECES : seq of Piece`PType = [<BOMB>, <MARSHALL>, <GENERAL>, <COLONEL>, <MAJOR>, <CAPTAIN>, <LIEUTENANT>, <SERGEANT>, <MINER>, <SCOUT>, <SPY>, <FLAG>];
instance variables
    private board : Board;
    private turn : nat := 0;

    -- Invariables
    inv turn = 1 or turn = 0;
operations
    -- Construtor
    public Combate : () ==> Combate
    Combate() ==
    (
      board := new Board();
    )
    post self.validTurn();

    -- Retorna o turno atual
    public pure getTurn : () ==> nat
      getTurn() == return turn;

    -- Efetua a mudança de turno
    public changeTurn : () ==> ()
      changeTurn() ==
      (
        if turn = 0 then turn := 1 else turn := 0;
      )
    pre self.validTurn()
    post self.validTurn();

    -- Verdadeiro se o turno for válido
    public pure validTurn : () ==> bool
      validTurn() == return turn = 0 or turn = 1;

functions
traces
end Combate
```

6. Geração de Código

O código java foi gerado com recurso à funcionalidade do overture para esse efeito.Tendo permitido sem grande dificuldade a implementação da GUI, além de um problema com o type Position que era criado em mais do que uma classe e depois a diferenciação na criação de objectos deste tipo.

7. Conclusões

Com este trabalho aprendemos a desenvolver e a testar modelos formais em VDM++ através do IDE Overture. Adquirimos a capacidade de demonstrar a consistência do modelo por nós criado bem como a capacidade de utilizar as ferramentas de geração de código java para permitir a criação de uma interface para a utilização do programa por parte do utilizador.

O nosso entendimento do VDM++ podia ser mais aprofundado, existem classes e “Types” que podiam ser implementados de maneira diferente que permitisse um modelo mais simples e mais reutilizável.

O trabalho não foi bem dividido, sendo que todos os membros participaram em cada parte do trabalho. Todos contribuíram igualmente para o sucesso do projeto.