Universidade do Porto

FEUP Faculdade de Engenharia

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

2013/2014

Relatório Final

T10 Cascade Game em VDM++

Turma: 4MIEIC04

Grupo: T4G10

Miao Sun – 200803743 – ei08162@fe.up.pt

Victor Filipe Carneiro Cerqueira – 201009027 – ei10055@fe.up.pt

08 de Dezembro de 2013

Índice

1. Descrição

Este trabalho foi realizado no âmbito da unidade curricular de Métodos Formais e Engenharia de Software. O objetivo é elaborar e aplicar os conhecimentos desenvolvidos para implementar um sistema de software em VDM++. O tema escolhido pelo grupo foi Cascade Game, que é um jogo de tabuleiro que contém 3 cores diferentes de blocos, o jogador tem que selecionar grupos que contém igual ou mais que 3 blocos da mesma cor para os eliminar, mais blocos eliminado duma vez mais pontuação recebe. Tem um bónus de uma bomba se conseguir eliminar igual ou mais que 15 blocos numa vez, se selecionar, a bomba elimina os blocos perto de si. O sistema insere uma nova linha de blocos com um intervalo constante. O jogador perde o jogo quando deixar os blocos exceder o tamanho do tabuleiro.

Foi usado o IDE Overture e editor de texto Microsoft Word para a realização da especificação formal do jogo, VDM++ Toolbox para validação de sintaxes e consistência da especificação, testar o funcionamento das funções e converter a especificação em código fonte Java, finalmente usado IDE Eclipse para a implementação da interface gráfica e umas funções que não podem ser realizado em VDM++.

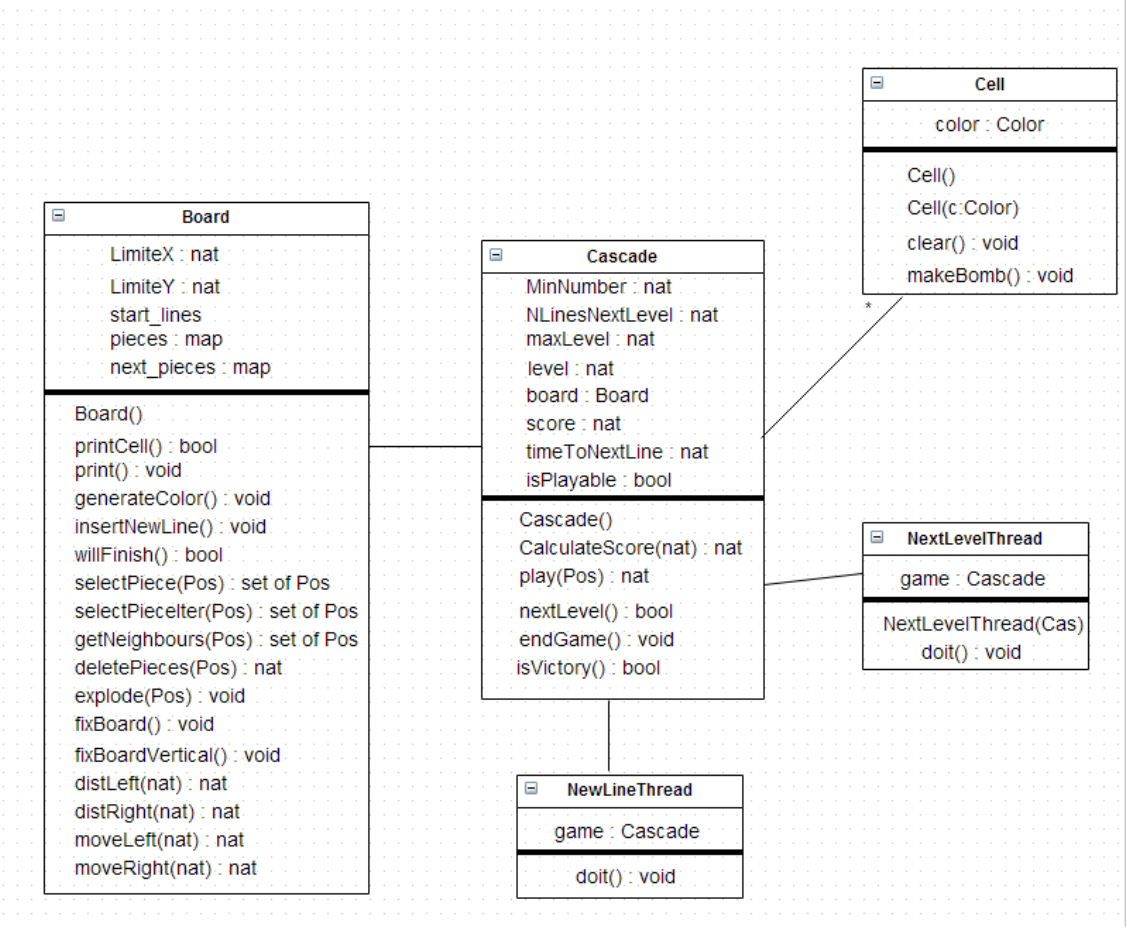
2. Lista de Requisitos

Código Identificador	**Requisitos**
REQ01	Os blocos só podem conter cores “red”, “yellow”, “green”, ou ser vazio ou conter uma bomba.
REQ02	Atualizar o tabuleiro após de cada jogada, de forma não deixar existir espaço entre blocos verticalmente e a primeira linha do tabuleiro horizontalmente.
REQ03	O sistema insere automaticamente uma linha de blocos num determinado intervalo de tempo.
REQ04	Atualizar o tabuleiro após de cada inserção de blocos feita automaticamente pelo sistema.
REQ05	Atribuir um bloco especial (bomba) se eliminar igual ou mais que 15 blocos numa só jogada.
REQ06	Calcula a pontuação de cada jogada, e atualizar a pontuação global do jogo após cada jogada.
REQ07	Terminar jogo se os blocos exceder o tamanho do tabuleiro verticalmente.
REQ08	O jogo tem vários níveis, entre automaticamente no próximo nível se o jogador conseguir manter jogar durante um determinado tempo.
REQ09	O jogador ganha o jogo se conseguir terminar o último nível do jogo.

3. Principais Restrições ao Funcionamento Correto

Código	Restrição
RES01	O tabuleiro tem um limite de tamanho determinado, os blocos não pode exceder o limite do tabuleiro.
RES02	O jogo começa com 6 linhas de blocos pré-gerados.
RES03	Na geração de blocos não pode gerar um bloco vazio nem gerar uma bomba, tem que conter uma cor ou “red” ou “yellow” ou “green”.
RES04	O sistema só insere automaticamente linhas de blocos quando a próxima iteração não termina o jogo.
RES05	Não pode tiver nenhum bloco exceder o limite vertical do tabuleiro, se tiver termina o jogo.
RES06	Não pode selecionar blocos vazios para obter vizinhos vazios ou eliminar grupo de blocos vazios.
RES07	Só pode eliminar grupos de blocos construído por 3 ou mais blocos de mesmo cor, exceto o bloco especial (bomba).
RES08	Só a bomba pode destruir blocos a volta, seja qualquer cor for.
RES09	Não pode existir espaço entre blocos verticalmente.
RES10	Não pode existir espaço entre blocos horizontalmente na linha base do tabuleiro.
RES11	O jogo começa no nível 1, e tem um limite de níveis determinados. Não pode exceder o limite de níveis.
RES12	O jogo só atribui pontuação positivos às jogadas que eliminam igual ou mais que 3 blocos.
RES13	O jogo só entre no próximo nível se o jogador conseguir manter jogar durante determinado tempo.

4. Diagrama de classes UML



5. Classes e Scripts de Teste

Classe TestCell

```
class TestCell

operations
	public AssertTrue : bool ==> ()
	AssertTrue(a) == return
	pre a;

	public runAllTestsCell : () ==> ()
	runAllTestsCell() ==
	(
		TestCreateCell();
		TestClearCell();
		TestMakeBomb();
	);

	public TestCreateCell : () ==> ()
	TestCreateCell() ==
	(
		dcl cell : Cell := new Cell(<red>);
		dcl cell2: Cell := new Cell();

		AssertTrue(cell.color = <red>);
		AssertTrue(cell2.color = <empty>);
	);

	public TestClearCell : () ==> ()
	TestClearCell() ==
	(
		dcl cell : Cell := new Cell(<red>);
		cell.clear();
		AssertTrue(cell.color = <empty>);
	);

	public TestMakeBomb : () ==> ()
	TestMakeBomb() ==
	(
		dcl cell : Cell := new Cell();
		cell.makeBomb();
		AssertTrue(cell.color = <bomb>);
	);

end TestCell
```

Classe TestBoard

```
class TestBoard

instance variables
	bd_h : Board := new Board();
	bd_v : Board := new Board();

operations
	public AssertTrue : bool ==> ()
	AssertTrue(a) == return
	pre a;

	public AuxBoard_H : () ==> ()
	AuxBoard_H() ==
	(
		bd_h.pieces := bd_h.pieces ++ {mk_Board`Position(6,6) |-> new Cell(<red>),
mk_Board`Position(6,7) |-> new Cell(<red>), mk_Board`Position(6,8) |-> new Cell(<red>),
mk_Board`Position(5,6) |-> new Cell(<green>), mk_Board`Position(5,7) |-> new Cell(<yellow>),
mk_Board`Position(5,8) |-> new Cell(<green>),   mk_Board`Position(7,6) |-> new Cell(<green>),
mk_Board`Position(7,7) |-> new Cell(<bomb>), mk_Board`Position(7,8) |-> new Cell(<green>),
mk_Board`Position(6,5) |-> new Cell(<green>), mk_Board`Position(6,9) |-> new Cell(<empty>)};
	);

	public AuxBoard_V : () ==> ()
	AuxBoard_V() ==
	(
		bd_v.pieces := bd_v.pieces ++ {mk_Board`Position(5, y) |-> new Cell() | y in set
{0,...,5}};
		bd_v.pieces := bd_v.pieces ++ {mk_Board`Position(6, y) |-> new Cell() | y in set
{0,...,5}};
	);

	public runAllTestsBoard : () ==> ()
	runAllTestsBoard () ==
	(
		TestCreateBoard();
		TestPrint();
		TestPrintCell();
		TestGenerateColor();
		TestInsertNewLine();
		TestWillFinish();
		TestSelectPiece();
		TestSelectPieceIter();
		TestGetNeighbours();
		TestDeletePieces();
		TestExplode();
		TestFixBoard();

		TestFixBoardVertical();
		TestDistLeft();
		TestDistRight();
		TestMoveLeft();
		TestMoveRight();
	);

	public TestCreateBoard : () ==> ()
	TestCreateBoard() ==
	(
		dcl board : Board := new Board();
```

```
            AssertTrue(board.pieces <> {|->});
    );

    public TestPrint : () ==> ()
    TestPrint() ==
    (
            dcl board : Board := new Board();
            AssertTrue(board.print() = true);
    );

    public TestPrintCell : () ==> ()
    TestPrintCell() ==
    (
            dcl c1 : Cell := new Cell();
            dcl c2 : Cell := new Cell(<red>);
            dcl c3 : Cell := new Cell(<green>);
            dcl c4 : Cell := new Cell(<yellow>);
            dcl c5 : Cell := new Cell(<bomb>);
            dcl io : IO := new IO();
            AssertTrue(Board`printCell(c1) = true);
            AssertTrue(Board`printCell(c2) = io.echo("R   "));
            AssertTrue(Board`printCell(c3) = io.echo("G   "));
            AssertTrue(Board`printCell(c4) = io.echo("Y   "));
            AssertTrue(Board`printCell(c5) = io.echo("#   "));
    );

    public TestGenerateColor : () ==> ()
    TestGenerateColor() ==
    (
            dcl board : Board := new Board();
            dcl c : Board`Color := board.generateColor();
            AssertTrue(c in set {<yellow>, <green>, <red>});
    );

    public TestInsertNewLine : () ==> ()
    TestInsertNewLine() ==
    (
            dcl board : Board := new Board();
            dcl oldPieces : map Board`Position to Cell := board.pieces;
            AssertTrue(oldPieces = board.pieces);
            board.insertNewLine();
            AssertTrue(oldPieces <> board.pieces);
    );

    public TestWillFinish : () ==> ()
    TestWillFinish() ==
    (
            dcl board : Board := new Board();
            AssertTrue(board.willFinish() = false);

            board.insertNewLine();
            board.insertNewLine();
            board.insertNewLine();
            board.insertNewLine();
            board.insertNewLine();
            board.insertNewLine();

            AssertTrue(board.willFinish() = true);
    );
```

```
	public TestSelectPiece : () ==> ()
	TestSelectPiece() ==
	(
		AuxBoard_H();
		AssertTrue(bd_h.selectPiece(mk_Board`Position(6,6)) =
bd_h.selectPieceIter(mk_Board`Position(6,6), {}));
		AssertTrue(bd_h.selectPiece(mk_Board`Position(6,6)) =
{mk_Board`Position(6,6), mk_Board`Position(6,7), mk_Board`Position(6,8)});
	);

	public TestSelectPieceIter : () ==> ()
	TestSelectPieceIter() ==
	(
		AuxBoard_H();
		AssertTrue(bd_h.selectPieceIter(mk_Board`Position(6,6), {}) =
{mk_Board`Position(6,6), mk_Board`Position(6,7), mk_Board`Position(6,8)});
				AssertTrue(mk_Board`Position(6,6) in set
bd_h.selectPieceIter(mk_Board`Position(6,6), {}));
	);

	public TestGetNeighbours : () ==> ()
	TestGetNeighbours() ==
	(
		AuxBoard_H();
		AssertTrue(bd_h.getNeighbours(mk_Board`Position(6,6)) =
{mk_Board`Position(6,6)} union {mk_Board`Position(6,7)});
		AssertTrue(mk_Board`Position(6,6) in set
bd_h.getNeighbours(mk_Board`Position(6,6)));
		AssertTrue(card bd_h.getNeighbours(mk_Board`Position(6,6)) = 2);
	);

	public TestDeletePieces : () ==> ()
	TestDeletePieces() ==
	(
		AuxBoard_H();
		AssertTrue(bd_h.deletePieces(mk_Board`Position(6,6))=3);
		AssertTrue(bd_h.pieces(mk_Board`Position(6,6)).color = <empty>);

		AuxBoard_H();
		bd_h.pieces(mk_Board`Position(6,8)).clear();
		AssertTrue(bd_h.deletePieces(mk_Board`Position(6,9)) = 0);
		AssertTrue(bd_h.deletePieces(mk_Board`Position(6,6)) = 2);

		bd_h.pieces := bd_h.pieces ++ {mk_Board`Position(x,5) |-> new Cell(<red>) | x
in set {0,...,15}};
		bd_h.pieces := bd_h.pieces ++ {mk_Board`Position(x,4) |-> new Cell(<green>) |
x in set {0,...,15}};

		AssertTrue(bd_h.deletePieces(mk_Board`Position(3,5)) = 18);
		AssertTrue(bd_h.pieces(mk_Board`Position(3,5)).color = <bomb>);

		AuxBoard_H();
		AssertTrue(bd_h.deletePieces(mk_Board`Position(7,7)) = 0);
	);

	public TestExplode : () ==> ()
	TestExplode() ==
	(
		AuxBoard_H();
		bd_h.explode(mk_Board`Position(7,7));
```

```
                AssertTrue(bd_h.pieces(mk_Board`Position(7,7)).color = <empty>);
                AssertTrue(bd_h.pieces(mk_Board`Position(7,8)).color = <empty>);
        );

        public TestFixBoard : () ==> ()
        TestFixBoard() ==
        (
                AuxBoard_H();
                bd_h.pieces(mk_Board`Position(6,7)).clear();
                bd_h.fixBoard();
                AssertTrue(bd_h.pieces(mk_Board`Position(6,7)).color = <red>);
                AssertTrue(bd_h.pieces(mk_Board`Position(6,8)).color = <empty>);
        );

        public TestFixBoardVertical : () ==> ()
        TestFixBoardVertical() ==
        (
                AuxBoard_V();
                bd_v.fixBoardVertical();
                AssertTrue(bd_v.pieces(mk_Board`Position(5,0)).color <> <empty>);
                AssertTrue(bd_v.pieces(mk_Board`Position(6,0)).color <> <empty>);
        );

        public TestDistLeft : () ==> ()
        TestDistLeft() ==
        (
                AuxBoard_V();
                AssertTrue(bd_v.distLeft(6) = 4);
                AssertTrue(bd_v.distLeft(8) = 8);
        );

        public TestDistRight : () ==> ()
        TestDistRight() ==
        (
                AuxBoard_V();
                AssertTrue(bd_v.distRight(5) = 7);
                AssertTrue(bd_v.distRight(8) = 8);
        );

        public TestMoveLeft : () ==> ()
        TestMoveLeft() ==
        (
                AuxBoard_V();
                AssertTrue(bd_v.moveLeft(10) = 72);
        );

        public TestMoveRight : () ==> ()
        TestMoveRight() ==
        (
                AuxBoard_V();
                AssertTrue(bd_v.moveRight(5) = 72);
        );

end TestBoard
```

Classe TestCascade

```
class TestCascade

operations

	public AssertTrue : bool ==> ()
	AssertTrue(a) == return
	pre a;

	public runAllTestsCascade : () ==> ()
	runAllTestsCascade() ==
	(
		TestCalculateScore();
		TestPlay();
		TestNextLevel();
		TestEndGame();
		TestIsVictory();
	);

	public TestCalculateScore : () ==> ()
	TestCalculateScore() ==
	(
		dcl cascade : Cascade := new Cascade();
		dcl n1 : nat := 3;
		dcl n2 : nat := 5;
		AssertTrue(cascade.CalculateScore(n1) = 50);
		AssertTrue(cascade.CalculateScore(n2) = 90);
	);

	public TestPlay : () ==> ()
	TestPlay() ==
	(
		dcl cascade : Cascade := new Cascade();
		AssertTrue(cascade.play(mk_Board`Position(6,6)) = 0);
		AssertTrue(cascade.play(mk_Board`Position(4,1)) = 50);
	);

	public TestNextLevel : () ==> ()
	TestNextLevel() ==
	(
		dcl cascade : Cascade := new Cascade();
		dcl b : bool := false;
		AssertTrue(cascade.nextLevel() = true);
		for all x in set {1,...,13} do
		b := cascade.nextLevel();
		AssertTrue(cascade.nextLevel() = false);
	);

	public TestEndGame : () ==> ()
	TestEndGame() ==
	(
		dcl cascade : Cascade := new Cascade();
		cascade.endGame();
		AssertTrue(cascade.isPlayable = false);
	);

	public TestIsVictory : () ==> ()
	TestIsVictory() ==
	(
		dcl cascade : Cascade := new Cascade();
```

```
            AssertTrue(cascade.isVictory() = false);
            cascade.level := 16;
            AssertTrue(cascade.isVictory() = true);
      )

end TestCascade
```

6. Matriz de Rastreabilidade

TestCell

	RES 01	RES 02	RES 03	RES 04	RES 05	RES 06	RES 07	RES 08	RES 09	RES 10	RES 11	RES 12	RES 13
TestCreateCell()													
TestClearCell()													
TestMakeBomb()													

TestBoard

	RES 01	RES 02	RES 03	RES 04	RES 05	RES 06	RES 07	RES 08	RES 09	RES 10	RES 11	RES 12	RES 13
TestCreateBoard ()		X											
TestPrint ()													
TestPrintCell ()													
TestGenerateColor()			X										
TestInsertNewLine()				X									
TestWillFinish()	X			X	X								
TestSelectPiece()						X							
TestSelectPieceIter()						X							
TestGetNeighbours()						X							
TestDeletePieces()							X						
TestExplode()								X					
TestFixBoard()									X				
TestFixBoardVertical()										X			
TestDistLeft()	X									X			
TestDistRight()	X									X			
TestMoveLeft()										X			
TestMoveRight()										X			

TestCascade

	RES 01	RES 02	RES 03	RES 04	RES 05	RES 06	RES 07	RES 08	RES 09	RES 10	RES 11	RES 12	RES 13
TestCalculateScore()												X	
TestPlay ()													
TestNextLevel ()											X		X
TestEndGame()	X				X								
TestIsVictory()													

7. Definição completa das classes

Classe Cell

Definição em VDM++ da classe Cell.

```
class Cell

types
	public Color = <yellow> | <green> | <red> | <empty> | <bomb>;

instance variables
	public color: Color := <empty>;

operations
	public Cell : () ==> Cell
	Cell () ==
	(
	)
	post self.color = <empty>;

	public Cell : Color ==> Cell
	Cell (c) ==
	(
		color := c;
	)
	post self.color = c;


	public clear : () ==> ()
	clear() ==
	(
		color := <empty>
	)
	post self.color = <empty>;

	public makeBomb : () ==> ()
	makeBomb() ==
	(
		color := <bomb>
	)
	pre self.color <> <bomb>
	post self.color = <bomb>;

end Cell
```

Classe Board

Definição em VDM++ da classe Board.

```
class Board
	values
		public LimiteX: nat = 15;
		public LimiteY: nat = 11;
		public start_lines: nat = 6;
		private io : IO = new IO();

	types
```

```
            public Color = Cell`Color;
            public X = nat
                  inv x == x in set {0,..., LimiteX};
            public Y = nat
                  inv y == y in set {0,..., LimiteY};
            public Position :: posX: X
                                                posY: Y;

      instance variables
            public pieces: map Position to Cell := {|->};

            public next_pieces: map X to Cell := {|->};
                  inv forall p in set rng next_pieces & p.color <>
<empty> and p.color <> <bomb>;

      functions

            public static printCell: Cell +> bool
            printCell(p) ==
            (
                  if p.color = <red> then io.echo("R   ")
                  else if p.color = <yellow> then io.echo("Y   ")
                  else if p.color = <green> then io.echo("G   ")
                  else if p.color = <bomb> then io.echo("#   ")
                  else io.echo(".    ")
            );


      Operations

            public Board : () ==> Board
            Board () ==
            (
                  new MATH().srand(1);
                  for all x in set {0,...,LimiteX} do
                  (
                        for all y in set {start_lines-1,...,LimiteY} do
                        (
                              pieces := pieces ++ {mk_Position(x,y)|-
>new Cell()}
                        );

                        for all y in set {0,...,start_lines-1} do
                        (
                              pieces := pieces ++ { mk_Position(x,y)|-
>new Cell(generateColor())}
                        );
                        next pieces := next pieces ++ {x|->new
Cell(generateColor())};
                  );
            )
            pre self.pieces = {|->}
            post self.pieces <> {|->};


            public generateColor : () ==> Color
            generateColor() ==
            (
                  dcl num: nat := (new MATH()).rand(3);
                  if(num = 0) then return <yellow>
                  else if(num = 1) then return <green>
```

```
                else return <red>
            )
      post RESULT in set {<yellow>, <green>,<red>};


            public insertNewLine : () ==> ()
            insertNewLine() ==
            (
                  dcl new_pieces : map Position to Cell :=
{mk_Position(x,y)|->new Cell() | x in set {0,..., LimiteX} , y in set
{0,..., LimiteY} };
                  dcl new_next_pieces : map X to Cell := {|->};

                  for all pos in set dom pieces do
                  (
                        if(pieces(pos).color <> <empty>) then
                        (
                              dcl new_pos : Position :=
mk_Position(pos.posX,pos.posY+1);
                              new_pieces :=new_pieces ++ {new_pos|->
pieces(pos)};
                        );
                  );

                  for all x in set{0,...,LimiteX} do
                  (
                        new_next_pieces := new_next_pieces ++ {x|->new
Cell(generateColor())};
                        new_pieces := new_pieces ++ {mk_Position(x,0)
|-> next_pieces(x)};
                  );
                  pieces := new_pieces;
                  next_pieces := new_next_pieces;
            )
            pre willFinish() = false
            post pieces <> pieces~;


            public willFinish : () ==> bool
            willFinish() ==
            (
                  return card{pos | pos in set dom pieces & pos.posY =
LimiteY and pieces(pos).color <> <empty>} > 0
            )
            post RESULT in set {true, false};


            public print : () ==> bool
            print() ==
            (
                  dcl res : bool;
                  for all y in set {0,...,LimiteY} do
                  (
                        for all x in set {-LimiteX,...,0} do
                        (
                              dcl p : Cell := pieces(mk_Position(-
x,y));
                              res := printCell(p);
                        );
                        res := io.echo("\n");
                  );
```

```
                for all x in set {0,...,LimiteX} do
                (
                      res := io.echo("----");
                );

                res := io.echo("\n");

                for all x in set {0,...,LimiteX} do
                (
                      dcl p : Cell := next_pieces(x);
                      res := printCell(p);
                );
                return res;
          );


          public selectPiece : Position ==> set of Position
          selectPiece(pos) ==
          (
                return selectPieceIter(pos,{});
          )
          pre pieces(pos).color <> <empty>;


          public selectPieceIter : Position * set of Position ==> set
of Position
          selectPieceIter(pos, analyzed) ==
          (
                dcl neighbours : set of Position :=
getNeighbours(pos);
                dcl res: set of Position := {};

                for all p in set neighbours do
                (
                      if (p not in set (analyzed union {pos})) then
                            res := res union selectPieceIter(p,
analyzed union {pos});
                );
                return res union {pos};
          )
          pre pieces(pos).color <> <empty>
          post pos in set RESULT and card RESULT >= 1;


          public getNeighbours : Position ==> set of Position
          getNeighbours(pos) ==
          (
                dcl res : set of Position := {pos};

                if(pos.posX > 0) then
                (
                      dcl n_pos : Position := mk_Position(pos.posX-
1,pos.posY);
                      if(pieces(pos).color = pieces(n_pos).color)
then
                            res := res union {n_pos};
                );

                if(pos.posY > 0) then
                (
```

```
                        dcl n_pos : Position :=
mk_Position(pos.posX,pos.posY-1);
                        if(pieces(pos).color = pieces(n_pos).color)
then
                              res := res union {n_pos};
                  );

                  if(pos.posX < LimiteX) then
                  (
                        dcl n_pos : Position :=
mk_Position(pos.posX+1,pos.posY);
                        if(pieces(pos).color = pieces(n_pos).color)
then
                              res := res union {n_pos};
                  );

                  if(pos.posY < LimiteY) then
                  (
                        dcl n_pos : Position :=
mk_Position(pos.posX,pos.posY+1);
                        if(pieces(pos).color = pieces(n_pos).color)
then
                              res := res union {n_pos};
                  );
                  return res;
            )
            pre pieces(pos).color <> <empty>
            post pos in set RESULT and card RESULT >= 1;


            public deletePieces : Position ==> nat
            deletePieces(pos) ==
            (
                  dcl pieces_to_delete : set of Position;
                  dcl n_pieces : nat;

                  if(pieces(pos).color = <empty>) then return 0;
                  if(pieces(pos).color = <bomb>) then
                  (
                        explode(pos);
                  fixBoard();
                        return 0;
                  );

                  pieces_to_delete := selectPiece(pos);
                  n_pieces := card pieces_to_delete;
                  if(n_pieces < 3) then return n_pieces;

                  for all p in set pieces_to_delete do
                  (
                        pieces(p).color := <empty>
                  );
                  if(n_pieces>=15) then pieces(pos).makeBomb();
                  fixBoard();
                  return n_pieces;
            )
            post RESULT >= 0;

            public explode : Position ==> ()
            explode(pos) ==
            (
```

```
                for all x in set {-3,...,3} do
                      for all y in set {-3,...,3} do
                      (
                            if(abs x + abs y <= 3 and pos.posX+x <=
LimiteX and pos.posX+x >= 0 and pos.posY+y <= LimiteY and pos.posY+y
>= 0) then
                                  pieces(mk_Position(pos.posX+x,
pos.posY+y)).color := <empty>
                      );
            )
            pre pieces(pos).color = <bomb>;
            --post pieces <> pieces~;


            public fixBoard : () ==> ()
            fixBoard() ==
            (
                  dcl count : nat := 0;
                  for all p in set dom pieces do
                  (
                        if(pieces(p).color <> <empty> and p.posY > 0
and pieces(mk_Position(p.posX,p.posY-1)).color = <empty>) then
                              (
                                    count := count+1;
                                    pieces(mk_Position(p.posX,p.posY-
1)).color := pieces(p).color;
                                    pieces(p).color := <empty>;
                              );
                  );


                  if(count > 0) then fixBoard();
                  --else fixBoardVertical();
            );
            --post pieces <> pieces~;


            public fixBoardVertical : () ==> ()
            fixBoardVertical() ==
            (
                  dcl count : nat := 0;
                  for all p in set {x|x in set dom pieces & x.posX >= 1
and x.posX < LimiteX} do
                  (
                        if(pieces(p).color = <empty> and p.posY = 0)
then
                        (
                              dcl distLeft : nat := distLeft(p.posX);
                              dcl distRight : nat := distRight(p.posX);
                              if distLeft < p.posX then
                              (
                                    count := count +
moveRight(distLeft);
            );
                              if distRight > p.posX then
                              (
                                    count := count +
moveLeft(distRight);
                              );
                        );
                  );
```

```
                  if(count > 0) then fixBoardVertical();
            );

            public distLeft : nat ==> nat
            distLeft(posx) ==
            (
                  dcl dm : seq of int := [x | x in set {-
posx+1,...,0}];
                  if(pieces(mk_Position(posx,0)).color = <empty>) then
                  while(len dm > 0) do
                  (
                        dcl h : int  := hd dm;
                        dm := tl dm;
                        if(pieces(mk_Position(-h,0)).color <> <empty>)
then return -h;
                  );
                  return posx;
            )
            pre posx >= 1
            post RESULT in set {0,...,LimiteX};


            public distRight : nat ==> nat
            distRight(posx) ==
            (
                  dcl dm : seq of nat := [x | x in set
{posx+1,...,LimiteX}];
                  if(pieces(mk_Position(posx,0)).color = <empty>) then
                  while(len dm > 0) do
                  (
                        dcl h : nat := hd dm;
                        dm := tl dm;
                        if(pieces(mk_Position(h,0)).color <> <empty>)
then return h;
                  );
                  return posx;
            )
            pre posx >= 0 and posx < LimiteX
            post RESULT in set {0,...,LimiteX};


            public moveLeft : nat ==> nat
            moveLeft(posx) ==
            (
                  dcl new_pieces: map Position to Cell := pieces;
                  dcl count : nat := 0;
                  for all x in set {-LimiteX,...,-posx} do
                  (
                        for all y in set {0,...,LimiteY} do
                        (
                              new_pieces(mk_Position(-x-1,y)).color :=
pieces(mk_Position(-x,y)).color;
                              new_pieces(mk_Position(-x,y)).color :=
<empty>;
                              count := count+1;
                        );
                  );
                  pieces := new_pieces;
                  return count;
            )
            pre posx>=1;
```

```
            public moveRight : nat ==> nat
            moveRight(posx) ==
            (
                  dcl new_pieces: map Position to Cell := pieces;
                  dcl count : nat := 0;
                  for all x in set {0,...,posx} do
                  (
                        for all y in set {0,...,LimiteY} do
                        (
                              new_pieces(mk_Position(x+1,y)).color :=
pieces(mk_Position(x,y)).color;
                              new_pieces(mk_Position(x,y)).color :=
<empty>;
                              count := count+1;
                        );
                  );
                  pieces := new_pieces;
                  return count;
            )
            pre posx<LimiteX;

end Board
```

Classe Cascade

Definição em VDM++ da classe Cascade.

```
class Cascade

values
      public MinNumber: nat = 3;
      public NLinesNextLevel : nat = 10;
      public maxLevel : nat = 15;

instance variables

      public board: Board := new Board();
      public level: nat := 1;
            inv level > 0 and level <= maxLevel+1;
      public score: nat := 0;
            inv score >= 0;
      public timeToNextLine : nat := 1000 * (8-level*0.5);
      public isPlayable : bool := true;

functions
      public CalculateScore : nat +> nat
      CalculateScore(n_blocks) ==
            50 + 20*(n_blocks - 3)
      pre n_blocks >= MinNumber
      post RESULT >= 50;

operations

      public Cascade : () ==> Cascade
      Cascade() ==
      (
            start(new NewLineThread(self));
```

```
            start(new NextLevelThread(self));

      );

      public play : Board`Position ==> nat
      play(pos) ==
      (
            dcl n_pieces : nat := board.deletePieces(pos);
            if(n_pieces>=3 and isPlayable) then
            (
                  score := score + CalculateScore(n_pieces);
                  return CalculateScore(n_pieces);
            )
            else return 0;
      );

      public nextLevel : () ==> bool
      nextLevel() ==
      (
            level := level+1;
            timeToNextLine := 1000 * (8-level*0.5);
            if level = maxLevel+1 then(
                  isPlayable := false;
                  return false;
            );
            return true;
      )
      pre isPlayable;

      public endGame : () ==> ()
      endGame() ==
      (
            isPlayable := false;
      )
      pre isPlayable;

      public isVictory : () ==> bool
      isVictory() ==
      (
            if(level = maxLevel+1) then return true
            else return false;
      )
      post isPlayable;

end Cascade
```

Classe NewLine Thread

Definição em VDM++ da classe NewLine Thread.

```
class NewLineThread

instance variables
      game : Cascade;

operations

      public NewLineThread : Cascade ==> NewLineThread
      NewLineThread(c) ==
```

```
	(
		game := c;
	);

	public doit : () ==> ()
	doit() ==
	(
		while game.isPlayable do
		(
			--wait (cannot be done in VDM++)
			game.board.insertNewLine();
			-- update display
		);
	);

sync
	per doit => #act(doit) <= 1

thread
	doit();

end NewLineThread
```

Classe NextLevelThread

Definição em VDM++ da classe NextLevelThread.

```
class NextLevelThread

instance variables
	game : Cascade;

operations

	public NextLevelThread : Cascade ==> NextLevelThread
	NextLevelThread(c) ==
	(
		game := c;
	);

	public doit : () ==> ()
	doit() ==
	(
		while game.isPlayable do
		(
			--wait (cannot be done in VDM++)
			if game.nextLevel() = false then game.isPlayable :=
false;
		);
	);

sync
	per doit => #act(doit) <= 1

thread
	doit();

end NextLevelThread
```

8. Cobertura de Testes

Classe Cell

name	*#calls*	*coverage*
Cell`clear	1	100%
Cell`Cell	2	100%
Cell`makeBomb	1	100%
Cell`Cell	2	100%
total		***100%***

Classe TestCell

name	*#calls*	*coverage*
TestCell`AssertTrue	4	100%
TestCell`TestMakeBomb	1	100%
TestCell`TestClearCell	1	100%
TestCell`TestCreateCell	1	100%
TestCell`runAllTestsCell	1	100%
total		***100%***

Classe Board

name	*#calls*	*coverage*
Board`Board	7	100%
Board`print	1	100%
Board`explode	2	100%
Board`distLeft	3	100%
Board`fixBoard	10	100%
Board`moveLeft	2	100%
Board`distRight	3	100%
Board`moveRight	2	100%
Board`printCell	213	100%
Board`willFinish	9	100%
Board`selectPiece	5	100%
Board`deletePieces	5	100%
Board`generateColor	897	100%
Board`getNeighbours	41	100%
Board`insertNewLine	7	100%
Board`selectPieceIter	38	100%
Board`fixBoardVertical	2	100%
total		***100%***

Classe TestBoard

name	*#calls*	*coverage*
TestBoard`TestPrint	1	100%
TestBoard`AssertTrue	38	100%
TestBoard`AuxBoard_H	8	100%
TestBoard`AuxBoard_V	5	100%
TestBoard`TestExplode	1	100%
TestBoard`TestDistLeft	1	100%
TestBoard`TestFixBoard	1	100%
TestBoard`TestMoveLeft	1	100%
TestBoard`TestDistRight	1	100%
TestBoard`TestMoveRight	1	100%
TestBoard`TestPrintCell	1	100%
TestBoard`TestWillFinish	1	100%
TestBoard`TestCreateBoard	1	100%
TestBoard`TestSelectPiece	1	100%
TestBoard`TestDeletePieces	1	100%
TestBoard`runAllTestsBoard	1	100%
TestBoard`TestGenerateColor	1	100%
TestBoard`TestGetNeighbours	1	100%
TestBoard`TestInsertNewLine	1	100%
TestBoard`TestSelectPieceIter	1	100%
TestBoard`TestFixBoardVertical	1	100%
total		***100%***

Classe Cascade

name	*#calls*	*coverage*
Cascade`play	2	100%
Cascade`Cascade	5	100%
Cascade`endGame	1	100%
Cascade`isVictory	2	100%
Cascade`nextLevel	15	100%
Cascade`CalculateScore	4	100%
total		***100%***

Classe TestCascade

name	*#calls*	*coverage*
TestCascade`TestPlay	1	100%
TestCascade`AssertTrue	9	100%
TestCascade`TestEndGame	1	100%
TestCascade`TestIsVictory	1	100%
TestCascade`TestNextLevel	1	100%
TestCascade`TestCalculateScore	1	100%
TestCascade`runAllTestsCascade	1	100%
total		***100%***

9. Consistência do Modelo

Segundo as regras observadas do jogo exemplo Cascade, podemos concluir que todas as regras do jogo estão devidamente implementadas. A lógica do jogo e as respetivas especificações do jogo foram implementadas em VDM++. Foram implementadas varias classes, nas quais 3 são principais, nomeadamente a classe Cell, a classe Board e a classe Cascade.

A classe Cell modela os blocos do tabuleiro, só tem um atributo que é *colo*, tem funções para criação do bloco, esvaziar bloco ou tornar-se um bomba.

A classe Board tem atributo *pieces* e next_*pieces*, ambas são um mapa de posições para um bloco, representa o tabuleiro do jogo e a linha que se insere automaticamente no jogo em cada determinado intervalo. Tem também muitas funções que são as operações necessários para garantir o correto funcionamento do jogo. Como *generateColor()* e *insertNewLine()* que gera e insere uma linha de blocos com cores sortidas, *selectPiece()* e *deletePieces()* que seleciona um grupo de blocos que é construído por 3 ou mais cores iguais e elimina-los, *fixBoard()* e *fixBoardVertical()* para garantir não existir espaços entre blocos verticalmente, e horizontalmente na linha mais abaixo do tabuleiro.

A classe Cascade tem atributos *board*, *level*, *score*, *timeToNextLine* e *isPlayable* que representa os componentes do jogo, tabuleiro, nível e pontuação do jogo, controlo de nível e a terminação do jogo. Tem funções play que faça a jogada (elimina os blocos selecionados) e atualiza a pontuação global do jogo, *nextLevel()*, *endGame()* e *isVictory()* para controlar o nível, terminação do jogo.

NewlineThread e NextLevelThread são 2 classes que fazem controlo da inserção de blocos e nível do jogo, precisa manipulação de tempo real do computador, como não conseguimos arranjar nenhuma maneira de realizar o tal em VDM++, foi desenvolvido a parte as funções em Java.

IO e MATH são 2 bibliotecas abordados para permitir a impressão no ecrã e fazer random para a geração de blocos.

Todas as classes mencionadas tem as regras implementadas para o correto funcionamento do jogo, tem invariantes, pré e pós condições para maior parte de atributos e funções.