U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Formal Modeling of the chinese board game Xo Dou Qi in VDM++

Mestrado Integrado em Engenharia Informática e Computação

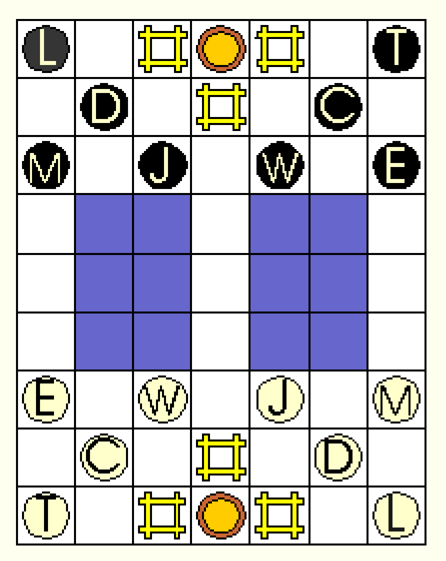
Métodos Formais em Engenharia de Software

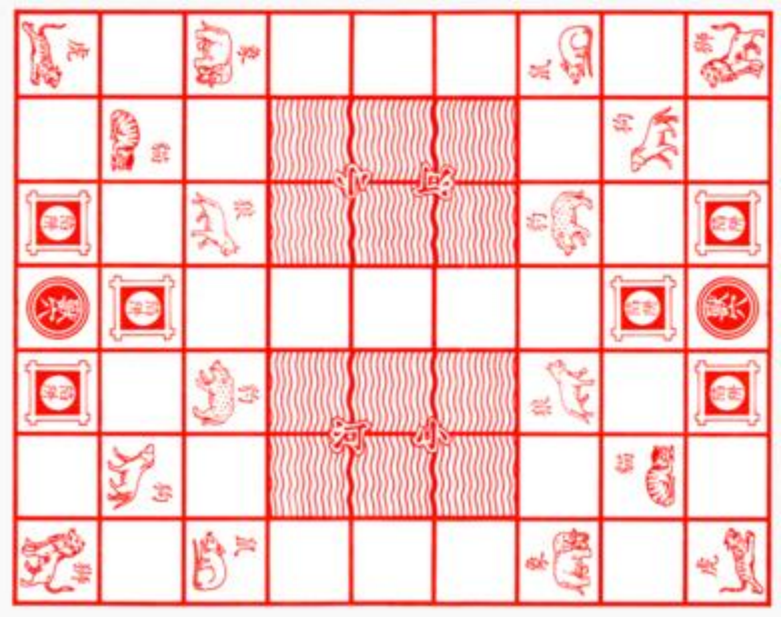
Francisco de Sousa Gomes Ferreira do Couto

Contents

1. Informal system description and list of requirements

1.1 Informal system description





The main goal for this project was to develop a replica of the Chinese board game called Xo Dou Qi. This game, played on a 7×9 board, is also known as The Jungle Game, Jungle Chess, or Animals Chess, and is sometimes called Oriental Chess or Children's Chess. Two different players play it. The goal of the game is to move a piece onto a special square, the den, on the opponent's side of the board.

There are eight different types of pieces available for each player namely with different values which represent their strength:

- 8 - Elephant
- 7 - Lion
- 6 - Tiger
- 5 - Leopard
- 4 - Wolf
- 3 - Dog
- 2 - Cat
- 1 – Rat

Each piece moves one square horizontally or vertically (not diagonally). If a stronger animal moves to a tile occupied by a weaker animal, this last one is captured. The only exception lies between the elephant and the rat since both can capture each other. Animals with the same strength are also mutual preys. In the middle of the board there are special squares which represent the river. These squares can’t be transposed by any animals except the rat who may walk on them freely. There’s also a special case in this situation since the lions and the tigers may jump over the river squares either horizontally or vertically, but never diagonally. The rat can’t capture the elephant coming from the river.

Surrounding both lairs of each players are three traps. If one animal falls on one of these it may be captured by any enemy animal even if it is weaker. The player who reaches the enemy's lair first wins.

As there are many variants of the game the rules above cited were all based on a single implementation described here in Portuguese.

As to meet all the requirements above mentioned the project contemplated two different types of implementations. The first one was a VDM++ implementation by formally modelling all the rules using Overture. A file named GameTest was developed to achieve a test coverage of 100% to validate all the movements, the board and game states and the winning condition. The second stage of implementation relied on the JAVA code generated by Overture. Since all the logic of the game was already completely developed there was only the need to draw the board and create a recursive gaming function to allow each player to input their desired movements until the game reached an end state and one of them won.

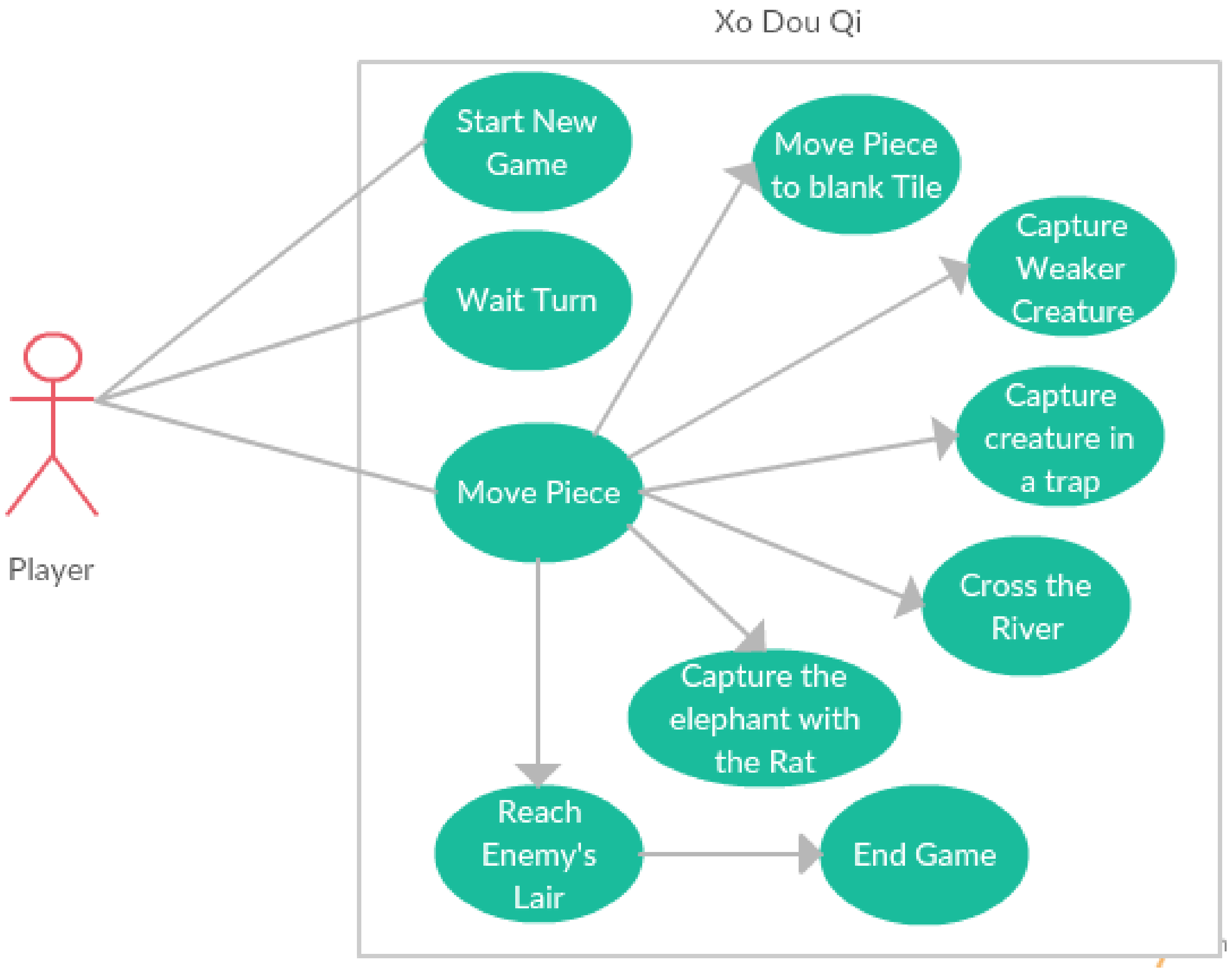
1.2 List of requirements

Id	Priority	Description
R1	Mandatory	The game must be playable for 2 people at the same time, on different turns.
R2	Mandatory	Each player must be able to move their own pieces horizontally or vertically, but never diagonally.
R3	Mandatory	Lions and tigers must be able to cross rivers horizontally or diagonally
R4	Mandatory	In case a stronger piece reaches the tile where a weaker piece lies, the latter must be captured.
R5	Mandatory	The rat and the elephant must be able to mutually capture each other.
R6	Mandatory	The rat must be the only piece to be able to walk along the rivers.
R7	Mandatory	If a stronger animal falls into a trap it must be vulnerable even to weaker creatures.
R8	Mandatory	Any player must win the game if it reaches the opponent's lair.

These requirements are directly translated onto use cases as shown next.

2. Visual UML model

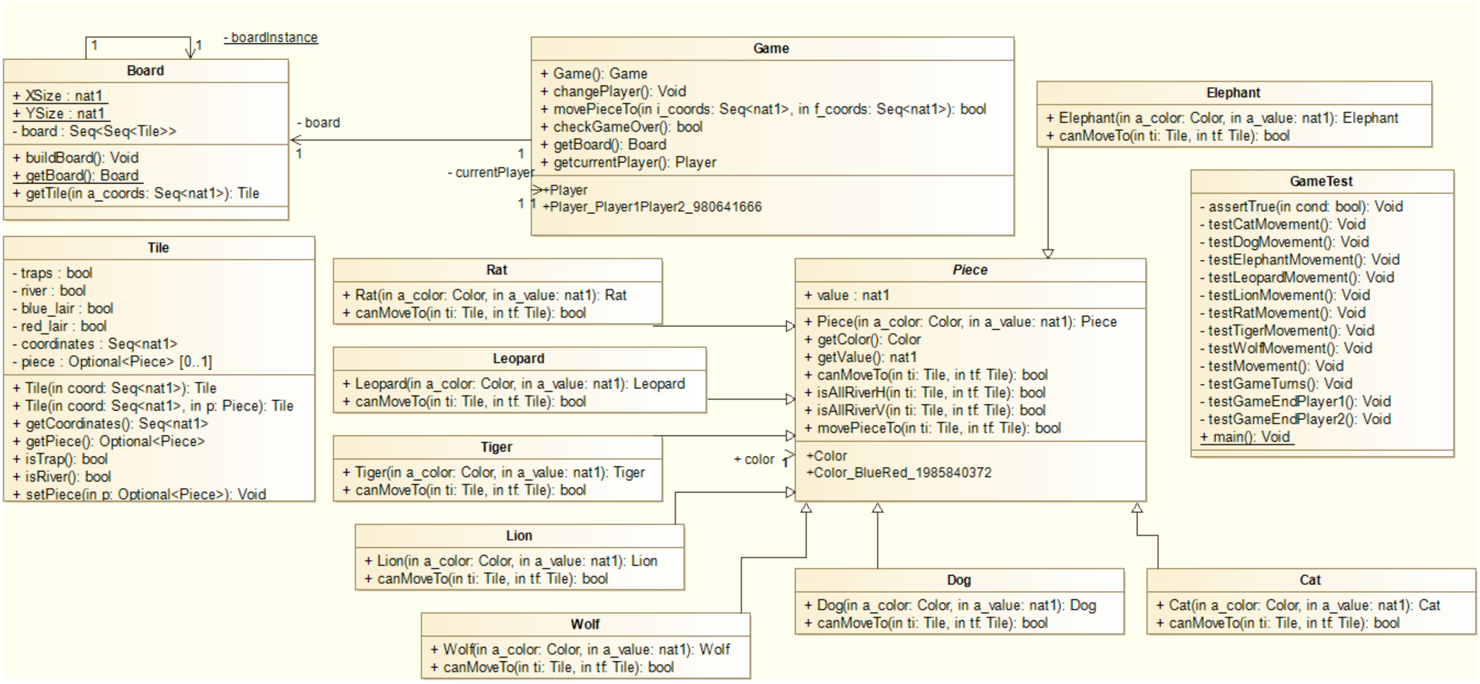
2.1 Use case model



Scenario	**Start New Game**
Description	Normal scenario to start playing the game developing a new board and with the pieces in their initial positions
Pre-conditions	
Post-conditions	1. The board is 7x9 2. Each player has 8 pieces 3. The initial player is Player1 4. No player has won the match yet
Scenario	**Wait Turn**
Description	Normal Scenario to wait its turn to be able to play again
Pre-conditions	1. It's the other player's turn to play
Post-conditions	1. Its player's turn to play now
Steps	1. The other player makes a valid play 2. The other player does not achieve a winning state

Scenario	**Move Piece**
Description	Normal scenario to make a play, moving a certain piece
Pre-conditions	1. No winning state has yet been achieved 2. The piece to be played exists in the selected tile 3. It's the player's turn to play
Post-conditions	1. The initial tile is empty 2. The chosen tile has the piece selected 3. It's the other's player to play or it was achieved a winning state *(final system state)*
Steps	1. The player selects the tile with the piece he wishes to move 2. The player selects the destination tile 3. The move is validated 4. The piece changes position

2.2 Class model



- boardInstance
1
1
Board
+ XSize : nat1
+ YSize : nat1
- board : Seq<Seq<Tile>>
+ buildBoard(): Void
+ getBoard(): Board
+ getTile(in a_coords: Seq<nat1>): Tile
- board
1
1
- currentPlayer
1 1
Game
+ Game(): Game
+ changePlayer(): Void
+ movePieceTo(in i_coords: Seq<nat1>, in f_coords: Seq<nat1>): bool
+ checkGameOver(): bool
+ getBoard(): Board
+ getcurrentPlayer(): Player
+Player
+Player_Player1Player2_980641666
Elephant
+ Elephant(in a_color: Color, in a_value: nat1): Elephant
+ canMoveTo(in ti: Tile, in tf: Tile): bool
GameTest
- assertTrue(in cond: bool): Void
- testCatMovement(): Void
- testDogMovement(): Void
- testElephantMovement(): Void
- testLeopardMovement(): Void
- testLionMovement(): Void
- testRatMovement(): Void
- testTigerMovement(): Void
- testWolfMovement(): Void
- testMovement(): Void
- testGameTurns(): Void
- testGameEndPlayer1(): Void
- testGameEndPlayer2(): Void
+ main(): Void
Tile
- traps : bool
- river : bool
- blue_lair : bool
- red_lair : bool
- coordinates : Seq<nat1>
- piece : Optional<Piece> [0..1]
+ Tile(in coord: Seq<nat1>): Tile
+ Tile(in coord: Seq<nat1>, in p: Piece): Tile
+ getCoordinates(): Seq<nat1>
+ getPiece(): Optional<Piece>
+ isTrap(): bool
+ isRiver(): bool
+ setPiece(in p: Optional<Piece>): Void
Rat
+ Rat(in a_color: Color, in a_value: nat1): Rat
+ canMoveTo(in ti: Tile, in tf: Tile): bool
Leopard
+ Leopard(in a_color: Color, in a_value: nat1): Leopard
+ canMoveTo(in ti: Tile, in tf: Tile): bool
Tiger
+ Tiger(in a_color: Color, in a_value: nat1): Tiger
+ canMoveTo(in ti: Tile, in tf: Tile): bool
+ color
1
Piece
+ value : nat1
+ Piece(in a_color: Color, in a_value: nat1): Piece
+ getColor(): Color
+ getValue(): nat1
+ canMoveTo(in ti: Tile, in tf: Tile): bool
+ isAllRiverH(in ti: Tile, in tf: Tile): bool
+ isAllRiverV(in ti: Tile, in tf: Tile): bool
+ movePieceTo(in ti: Tile, in tf: Tile): bool
+Color
+Color_BlueRed_1985840372
Lion
+ Lion(in a_color: Color, in a_value: nat1): Lion
+ canMoveTo(in ti: Tile, in tf: Tile): bool
Dog
+ Dog(in a_color: Color, in a_value: nat1): Dog
+ canMoveTo(in ti: Tile, in tf: Tile): bool
Cat
+ Cat(in a_color: Color, in a_value: nat1): Cat
+ canMoveTo(in ti: Tile, in tf: Tile): bool
Wolf
+ Wolf(in a_color: Color, in a_value: nat1): Wolf
+ canMoveTo(in ti: Tile, in tf: Tile): bool

Class	Description
Piece	Defines a piece the player can play with in the board and the core characteristics amongst all. It's the superclass.
Rat	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Rat.
Leopard	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Leopard.
Tiger	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Tiger.
Lion	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Lion.
Wolf	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Wolf.

Dog	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Dog.
Cat	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Cat.
Elephant	Subclass of piece. Defines one of the pieces' types on the board and its attributes – the Elephant.
Tile	Defines each square in the game board either if it exists a piece or not.
Board	Main class regarding the board definition as an aggregate of tiles which might or not contain pieces, its size, and other characteristics.
Game	Core model; defines the state variables and operations available to the players.
GameTest	Defines the test/usage scenarios and test cases for the game, including movements for each of the pieces defined and its attributes.

3. Formal VDM++ model

3.1 Class Piece

```
class Piece
/*
class responsible for the variables and operations of all the subclasses(types of pieces
in the game)
*/
types
      public Color = <Blue> | <Red>; -- Defines the player to whom that piece belong
values
-- Correspond to global variables such as PI
instance variables
      public color: Color;
      public value: nat1;

operations --Correspond to operations in UML, methods in Java, and memberfunctions in
C++

      --Piece constructor
      public Piece : Color * nat1 ==> Piece
      Piece(a_color, a_value) ==(
            color := a_color;
            value := a_value;
      );

      --returns the color of the piece
      pure public getColor : () ==> Color
      getColor() ==
            return self.color;

      --returns the color of the piece
      pure public getValue : () ==> nat1
      getValue() ==
            return self.value;

      --checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
            == is subclass responsibility;
      --checks special types of movements across rivers horizontally (lions and tigers)
      public isAllRiverH: Tile * Tile ==> bool
      isAllRiverH(ti, tf) == (

             if(ti.getCoordinates()(2) = tf.getCoordinates()(2))
                          then(
                          dcl tempx: nat1 := ti.getCoordinates()(1) + 1;

                          while (tempx < tf.getCoordinates()(1)) do(
                                dcl tempT: Tile := new Tile([tempx,
ti.getCoordinates()(2)]);
                                if(tempT.isRiver() = false or is_Rat(tempT.getPiece()))
                                      then (return false;)
                                else
                                      tempx := tempx + 1;
                                );

                                return true;
                          )
             else return false;
      );

      --checks special types of movements across rivers vertically (lions and tigers)
      public isAllRiverV: Tile * Tile ==> bool
      isAllRiverV(ti, tf) == (

                   if(ti.getCoordinates()(1) = tf.getCoordinates()(1))
                          then(

                          dcl tempy: nat1 := ti.getCoordinates()(2) + 1;

                          while (tempy < tf.getCoordinates()(2)) do(
                                dcl tempT: Tile := new Tile([ti.getCoordinates()(1),
tempy]);
                                if((tempT.isRiver() = false) or
(is_Rat(tempT.getPiece())))
                                      then (return false;)
                                else
                                      tempy := tempy + 1;
                                );

                                return true;

                          )
                   else return false;
      );

    --method used to, according all restrictions, change a piece from place
    public movePieceTo: Tile * Tile ==> bool
    movePieceTo(ti, tf) == (
             if(ti.getPiece().canMoveTo(ti,tf))
                   then (
                          tf.setPiece(ti.getPiece());
                          ti.setPiece(nil);
                          return true;
                          )
             else return false;
    )
    pre ti.getPiece() <> nil;
    end Piece
```

3.2 Class Rat

```
class Rat is subclass of Piece
/*
class responsible for the pieces of type Rat
*/
operations
        public Rat : Color * nat1 ==> Rat
                        Rat(a_color, a_value) ==(
                                Piece(a_color, a_value);
                        );

        --checks if a piece in a certain tile can move to another one
        public canMoveTo : Tile * Tile  ==> bool
        canMoveTo(ti, tf)
                ==  if(
                                        (
                                        (abs(ti.getCoordinates()(1) -
tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                                        (abs(ti.getCoordinates()(2) -
tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0)
                                        ))
                                       then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue())
                                       or (is_Rat(ti.getPiece()) and
is_Elephant(tf.getPiece()) and (ti.isRiver() = false))
                                       or (tf.isTrap() = true);)
                                else (return false;)
        pre ti <> tf;


end Rat
```

3.3 Class Leopard

```
class Leopard is subclass of Piece
/*
class responsible for the pieces of type Leopard
*/
operations
        public Leopard : Color * nat1 ==> Leopard
                        Leopard(a_color, a_value) ==(
                                Piece(a_color, a_value);
                        );

--checks if a piece in a certain tile can move to another one
        public canMoveTo : Tile * Tile  ==> bool
        canMoveTo(ti, tf)
                ==  if(
                        ( (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and
ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
```

```
                   (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and
ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0) )
                   and tf.isRiver() = false)
                   then (return (tf.getPiece() = nil or ti.getPiece().getValue() >=
tf.getPiece().getValue() or tf.isTrap() = true ))
                 else (return false;)
      pre ti <> tf;



 end Leopard
```

3.4 Class Tiger

```
class Tiger is subclass of Piece
/*
class responsible for the pieces of type Tiger
*/
operations
      public Tiger : Color * nat1 ==> Tiger
                   Tiger(a_color, a_value) ==(
                          Piece(a_color, a_value);
                   );

      --checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
             ==  (
                          if(
                           ( (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1
and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                          (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and
ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0))and tf.isRiver() = false)
                              then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue() or tf.isTrap() = true ))
                          else if(isAllRiverH(ti,tf) or isAllRiverV(ti,tf))
                              then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue()))
                          else
                                return false;
                          )
      pre ti <> tf;

end Tiger
```

3.5 Class Lion

```
class Lion is subclass of Piece
/*
class responsible for the pieces of type Lion
*/
operations
      public Lion : Color * nat1 ==> Lion
                   Lion(a_color, a_value) ==(
                          Piece(a_color, a_value);
                   );

      --checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
             ==  (
                          if(
                                        (
                                        (abs(ti.getCoordinates()(1) -
tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                                        (abs(ti.getCoordinates()(2) -
tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0)
                                        )
                                       and tf.isRiver() = false)
                                             then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue() or tf.isTrap() = true))
                          else if(isAllRiverH(ti,tf) or isAllRiverV(ti,tf))
                                             then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue()))
                          else
                                       return false;
                          )
      pre ti <> tf;


end Lion
```

3.6 Class Wolf

```
class Wolf is subclass of Piece
/*
class responsible for the pieces of type Wolf
*/
operations
      public Wolf : Color * nat1 ==> Wolf
                   Wolf(a_color, a_value) ==(
                          Piece(a_color, a_value);
                   );

--checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
             ==  if(
                                        (
                                        (abs(ti.getCoordinates()(1) -
tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                                        (abs(ti.getCoordinates()(2) -
tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0)
                                        )
```

```
                                      and tf.isRiver() = false)
                                            then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue() or tf.isTrap() = true ))
                         else (return false;)
      pre ti <> tf;

end Wolf
```

3.7 Class Dog

```
class Dog is subclass of Piece
/*
class responsible for the pieces of type Dog
*/
operations
      public Dog : Color * nat1 ==> Dog
                   Dog(a_color, a_value) ==(
                         Piece(a_color, a_value);
                   );

--checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
            ==  if(
                                       (
                                       (abs(ti.getCoordinates()(1) -
tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                                       (abs(ti.getCoordinates()(2) -
tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0)
                                       )
                                      and tf.isRiver() = false)
                                            then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue() or tf.isTrap() = true ))
                         else (return false;)
      pre ti <> tf;

end Dog
```

3.8 Class Cat

```
class Cat is subclass of Piece
/*
class responsible for the pieces of type Cat
*/
operations
      public Cat : Color * nat1 ==> Cat
            Cat(a_color, a_value) ==(
                   Piece(a_color, a_value);
            );

      --checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
            ==  if(
                                       (
                                        (abs(ti.getCoordinates()(1) -
tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                                        (abs(ti.getCoordinates()(2) -
tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0)
                                        )
                                      and tf.isRiver() = false)
                                            then (return (tf.getPiece() = nil or
ti.getPiece().getValue() >= tf.getPiece().getValue() or tf.isTrap() = true ))
                          else (return false;)
      pre ti <> tf;

end Cat
```

3.9 Class Elephant

```
class Elephant is subclass of Piece
/*
class responsible for the pieces of type Elephant
*/
operations
      public Elephant : Color * nat1 ==> Elephant
                   Elephant(a_color, a_value) ==(
                          Piece(a_color, a_value);
                   );

--checks if a piece in a certain tile can move to another one
      public canMoveTo : Tile * Tile  ==> bool
      canMoveTo(ti, tf)
             ==  if(
                                        (
                                        (abs(ti.getCoordinates()(1) -
tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.getCoordinates()(2) = 0) or
                                        (abs(ti.getCoordinates()(2) -
tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.getCoordinates()(1) = 0)
                                        )
                                      and tf.isRiver() = false)
                                            then (return true)
                          else (return false;)
      pre ti <> tf;

end Elephant
```

3.10 Class Tile

```
class Tile
/*
class responsible for the variables and operations of all the tiles that together form
the game board
*/
instance variables
             private traps: bool; --this tile has a trap
             private river: bool; --this tile has water
             private blue_lair: bool;
             private red_lair: bool;
             private coordinates: seq of nat1; --tile coordinates, integrity check with
board coordinates
             private piece: [Piece]; --Piece on tile, can be null if there is none
             inv coordinates(1) in set {1, ..., Board`XSize} and
                          coordinates(2) in set {1, ..., Board`YSize}

operations

             --Tile constuctor
             public Tile: seq of nat1 ==> Tile
             Tile(coord) == (
                          coordinates := coord;
                          piece := nil;
                          traps := ((coord(1) = 3 and coord(2) = 1) or (coord(1) = 5 and
coord(2) = 1) or (coord(1) = 4 and coord(2) = 2)
                                                                 or (coord(1) = 3 and
coord(2) = 9) or (coord(1) = 5 and coord(2) = 9) or (coord(1) = 4 and coord(2) = 8));
                          river := ((coord(1) = 2 and coord(2) = 4) or (coord(1) = 3 and
coord(2) = 4) or (coord(1) = 5 and coord(2) = 4) or (coord(1) = 6 and coord(2) = 4)
                                                    or (coord(1) = 2 and coord(2) = 5) or
(coord(1) = 3 and coord(2) = 5) or (coord(1) = 5 and coord(2) = 5) or (coord(1) = 6 and
coord(2) = 5)
                                                    or (coord(1) = 2 and coord(2) = 6) or
(coord(1) = 3 and coord(2) = 6) or (coord(1) = 5 and coord(2) = 6) or (coord(1) = 6 and
coord(2) = 6));
                          blue_lair := (coord(1) = 4 and coord(2) = 1);
                          red_lair := (coord(1) = 4 and coord(2) = 9);
                          return self;
             );

             --Tile constructor with piece
             public Tile: seq of nat1 * Piece ==> Tile
                          Tile(coord, p) == (
                          coordinates := coord;
                          traps := ((coord(1) = 3 and coord(2) = 1) or (coord(1) = 5 and
coord(2) = 1) or (coord(1) = 4 and coord(2) = 2)
                                                                 or (coord(1) = 3 and
coord(2) = 9) or (coord(1) = 5 and coord(2) = 9) or (coord(1) = 4 and coord(2) = 8));
                          river := ((coord(1) = 2 and coord(2) = 4) or (coord(1) = 3 and
coord(2) = 4) or (coord(1) = 5 and coord(2) = 4) or (coord(1) = 6 and coord(2) = 4)
                                                    or (coord(1) = 2 and coord(2) = 5) or
(coord(1) = 3 and coord(2) = 5) or (coord(1) = 5 and coord(2) = 5) or (coord(1) = 6 and
coord(2) = 5)
                                                    or (coord(1) = 2 and coord(2) = 6) or
(coord(1) = 3 and coord(2) = 6) or (coord(1) = 5 and coord(2) = 6) or (coord(1) = 6 and
coord(2) = 6));
                          blue_lair := (coord(1) = 4 and coord(2) = 1);
                          red_lair := (coord(1) = 4 and coord(2) = 9);
                          piece := p;
                          return self
             );

             --returns the coordinates of that tile
             pure public getCoordinates: () ==> seq of nat1
             getCoordinates() == return coordinates;

             --returns the piece in that tile
             pure public getPiece: () ==> [Piece]
             getPiece() == return piece;

             --returns whether on that tile lies a trap or not
             pure public isTrap: () ==> bool
             isTrap() == return traps;

             --returns whether that trap is a river or not
             pure public isRiver: () ==> bool
             isRiver() == return river;

             --method to set the piece that lies on the tile
             public setPiece: [Piece] ==> ()
             setPiece(p) == piece := p;

end Tile
```

3.11 Class Board

```
class Board
/*
class responsible for the definition of the game Board composed of all the Tiles and
subsequently the pieces in each one
*/
values --Board is a 7x9 rectangle = 63 tiles
             public XSize: nat1 = 7;
             public YSize: nat1 = 9;
instance variables
             private board: seq of seq of Tile := [[]];
             private static boardInstance:Board := new Board(); --Board singleton
operations

             --function responsible for building the game board with all the pieces in
place and their corresponding values
             public buildBoard: () ==> ()
             buildBoard() == (

                    board := [
                          [new Tile([1,1], new Lion(<Blue>, 7)), new Tile([2,1]), new
Tile([3,1]), new Tile([4,1]), new Tile([5,1]), new Tile([6,1]), new Tile([7,1], new
Tiger(<Blue>, 6))], ---first row

                          [new Tile([1,2]), new Tile([2,2], new Dog(<Blue>, 3)), new
Tile([3,2]), new Tile([4,2]), new Tile([5,2]), new Tile([6,2], new Cat(<Blue>, 2)), new
Tile([7,2])], ---second row

                          [new Tile([1,3], new Rat(<Blue>, 1)), new Tile([2,3]), new
Tile([3,3], new Leopard(<Blue>, 5)), new Tile([4,3]), new Tile([5,3], new Wolf(<Blue>,
4)), new Tile([6,3]), new Tile([7,3], new Elephant(<Blue>, 8))], --third row

                          [new Tile([1,4]), new Tile([2,4]), new Tile([3,4]), new
Tile([4,4]), new Tile([5,4]), new Tile([6,4]), new Tile([7,4])], --fourth row
                          [new Tile([1,5]), new Tile([2,5]), new Tile([3,5]), new
Tile([4,5]), new Tile([5,5]), new Tile([6,5]), new Tile([7,5])], --fifth row
                          [new Tile([1,6]), new Tile([2,6]), new Tile([3,6]), new
Tile([4,6]), new Tile([5,6]), new Tile([6,6]), new Tile([7,6])], --sixth row

                          [new Tile([1,7], new Elephant(<Red>, 8)), new Tile([2,7]), new
Tile([3,7], new Wolf(<Red>, 4)), new Tile([4,7]), new Tile([5,7], new Leopard(<Red>,
5)), new Tile([6,7]), new Tile([7,7], new Rat(<Red>, 1))], --seventh row
```

```
                          [new Tile([1,8]), new Tile([2,8], new Cat(<Red>, 2)), new
Tile([3,8]), new Tile([4,8]),new Tile([5,8]), new Tile([6,8], new Dog(<Red>, 3)), new
Tile([7,8])], ---eighth row

                          [new Tile([1,9], new Tiger(<Red>, 6)), new Tile([2,9]), new
Tile([3,9]), new Tile([4,9]), new Tile([5,9]), new Tile([6,9]), new Tile([7,9], new
Lion(<Red>, 7))] ---ninth row
                          ];
             );

             --returns the board singleton
             public static getBoard: () ==> Board
             getBoard() == return boardInstance;

             --get method for a board tile
             pure public getTile: seq of nat1 ==> Tile
             getTile(a_coords) == (
                          return board(a_coords(2))(a_coords(1));
             );

end Board
```

3.12 Class Game

```
class Game
/*
class responsible for game logic - Piece Movement and Player Turns
*/
types
      public Player = <Player1> | <Player2>;

instance variables
      private board : Board := new Board();
      private currentPlayer: Player;

operations

             --Game constructor
             public Game: () ==> Game
    Game() == (
                   board := Board`getBoard();
                   board.buildBoard();
                   currentPlayer := <Player1>;
    );

    --method used to change the player that will play this turn
    public changePlayer: () ==> ()
    changePlayer() == (
      if currentPlayer = <Player1> then currentPlayer := <Player2>
      else currentPlayer := <Player1>;

    )
             post currentPlayer <> currentPlayer~;

             --method used to, according all restrictions, change a piece from place
             public movePieceTo: seq of nat1 * seq of nat1 ==> bool
    movePieceTo(i_coords, f_coords) == (
      if(board.getTile(i_coords).getPiece() <> nil
                   and ((board.getTile(i_coords).getPiece().getColor() = <Blue> and
currentPlayer = <Player1>) or (board.getTile(i_coords).getPiece().getColor() = <Red> and
currentPlayer = <Player2>)))
             then return
board.getTile(i_coords).getPiece().movePieceTo(board.getTile(i_coords),
board.getTile(f_coords))
      else return false;
    );

             --method to check if any player won already
             public checkGameOver: () ==> bool
             checkGameOver() == (
                   if((board.getTile([4,1]).getPiece() <> nil and
board.getTile([4,1]).getPiece().getColor() = <Red>) or
                            (board.getTile([4,9]).getPiece() <> nil and
board.getTile([4,9]).getPiece().getColor() = <Blue>))
                          then return true;
                   return false;
             );

      --returns the actual gaming board
             pure public getBoard : () ==> Board
      getBoard() ==
      (
             return board;
      );

      --returns the player whose turn it is to play
      pure public getcurrentPlayer : () ==> Player
      getcurrentPlayer() ==
      (
             return currentPlayer;
      );

end Game
```

As it is visible, due to some bug in the colouring plug-in from overture some parenthesis were not coloured green during the coverage testing. To verify if it was due to some error or just some flaw in the IDE, percentages of coverage were generated automatically using another overture method and all stated a 100% coverage ratio for all the classes previously shown.

4. Model validation

4.1 Class GameTest

```
class GameTest

operations
      private assertTrue: bool ==> ()
                assertTrue(cond) == return
                pre cond;

  private testCatMovement: () ==> ()
                testCatMovement() ==
                (
                       dcl c: Piece := new Cat(<Red>, 2);
                       dcl r: Piece := new Rat(<Blue>, 1);
                       dcl l: Piece := new Lion(<Blue>, 7);
                       dcl t1:Tile := new Tile([1,1],c);
                       dcl t2:Tile  := new Tile([1,2]);
                       dcl t3:Tile  := new Tile([2,1]);
                       dcl t4:Tile  := new Tile([2,2]);
                       dcl t5:Tile  := new Tile([1,2],r);
                       dcl t6:Tile  := new Tile([1,2],l);


      assertTrue(c.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(c.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(c.canMoveTo(t1, t4) = false); -- diagonal movement


      assertTrue(c.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature

      assertTrue(c.canMoveTo(t1, t6) = false); -- horizontal movement but with a
stronger creature
                );

  private testDogMovement: () ==> ()
                testDogMovement() ==
                (
                        dcl d: Piece := new Dog(<Red>, 3);
                        dcl r: Piece := new Rat(<Blue>, 1);
                        dcl l: Piece := new Lion(<Blue>, 7);
                        dcl t1:Tile := new Tile([1,1],d);
                        dcl t2:Tile  := new Tile([1,2]);
                        dcl t3:Tile  := new Tile([2,1]);
                        dcl t4:Tile  := new Tile([2,2]);
                        dcl t5:Tile  := new Tile([1,2],r);
                        dcl t6:Tile  := new Tile([1,2],l);


      assertTrue(d.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(d.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(d.canMoveTo(t1, t4) = false); -- diagonal movement
      assertTrue(d.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature

      assertTrue(d.canMoveTo(t1, t6) = false); -- horizontal movement but with a
stronger creature

             );

   private testElephantMovement: () ==> ()
             testElephantMovement() ==
             (
             dcl e: Piece := new Elephant(<Red>, 8);
             dcl r: Piece := new Rat(<Blue>, 1);
             dcl l: Piece := new Lion(<Blue>, 7);

             dcl t1:Tile := new Tile([1,1],e);
             dcl t2:Tile := new Tile([1,2]);
             dcl t3:Tile := new Tile([2,1]);
             dcl t4:Tile := new Tile([2,2]);

             dcl t5:Tile := new Tile([1,2],r);
             dcl t6:Tile := new Tile([1,2],l);


      assertTrue(e.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(e.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(e.canMoveTo(t1, t4) = false); -- diagonal movement


      assertTrue(e.canMoveTo(t1, t5)); -- elephant can eat rats too

      assertTrue(e.canMoveTo(t1, t6)); -- horizontal movement eating a weaker creature
  );

   private testLeopardMovement: () ==> ()
      testLeopardMovement() ==
       (
             dcl leo: Piece := new Leopard(<Red>, 5);
             dcl r: Piece := new Rat(<Blue>, 1);
             dcl l: Piece := new Lion(<Blue>, 7);

             dcl t1:Tile := new Tile([1,1],leo);
             dcl t2:Tile := new Tile([1,2]);
             dcl t3:Tile := new Tile([2,1]);
             dcl t4:Tile := new Tile([2,2]);

             dcl t5:Tile := new Tile([1,2],r);
             dcl t6:Tile := new Tile([1,2],l);


      assertTrue(leo.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(leo.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(leo.canMoveTo(t1, t4) = false); -- diagonal movement
      assertTrue(leo.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature

      assertTrue(leo.canMoveTo(t1, t6) = false); -- horizontal movement but with a
stronger creature

      );

    private testLionMovement: () ==> ()
      testLionMovement() ==
       (
            dcl l: Piece := new Lion(<Red>, 7);
            dcl r: Piece := new Rat(<Blue>, 1);
            dcl e: Piece := new Elephant(<Blue>, 8);

            dcl t1:Tile := new Tile([2,3],l); ---lion near the river vertically
            dcl t8:Tile := new Tile([1,4],l); ---lion near the river horizontally
            dcl t10:Tile := new Tile([1,5],l); ---lion near the river horizontally but
unable to jump

            dcl t2:Tile := new Tile([2,2]);
            dcl t3:Tile := new Tile([3,3]);
            dcl t4:Tile := new Tile([1,4]);

            dcl t5:Tile := new Tile([2,7]);

            dcl t6:Tile := new Tile([3,3],r);
            dcl t7:Tile := new Tile([3,3],e);

            dcl t9:Tile := new Tile([4,4]);

            dcl t11:Tile := new Tile([4,5], r);


      assertTrue(l.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(l.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(l.canMoveTo(t1, t4) = false); -- diagonal movement


      assertTrue(l.canMoveTo(t1, t5)); --- jump the river vertically

      assertTrue(l.canMoveTo(t8, t9)); --- jump the river horizontally


      assertTrue(l.canMoveTo(t10, t11)); --- jump the river horizontally eating a rat


      assertTrue(l.canMoveTo(t1, t6)); -- horizontal movement eating a weaker creature

      assertTrue(l.canMoveTo(t1, t7) = false); -- horizontal movement but with a
stronger creature

      );

       private testRatMovement: () ==> ()
            testRatMovement() ==
            (
                   dcl r1: Piece := new Rat(<Red>, 1);
                   dcl r2: Piece := new Rat(<Red>, 1);

                   dcl e: Piece := new Elephant(<Blue>, 8);
                   dcl l: Piece := new Lion(<Blue>, 7);

                   dcl t1:Tile := new Tile([2,3],r1); ---rat near the river
                   dcl t2:Tile := new Tile([2,2]);
                   dcl t3:Tile := new Tile([3,3]);
                   dcl t4:Tile := new Tile([1,4]);

                   dcl t5:Tile := new Tile([2,4], r2); ---rat in the river
                   dcl t6:Tile := new Tile([2,3], e);

                   dcl t7:Tile := new Tile([3,3],e);
                   dcl t8:Tile := new Tile([3,3],l);

                   dcl t9:Tile := new Tile([2,5]); --- Tile in the river


      assertTrue(r1.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(r1.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(r1.canMoveTo(t1, t4) = false); -- diagonal movement


      assertTrue(r2.canMoveTo(t5, t6) = false); --- can't eat the elephant from the
river

      assertTrue(r2.canMoveTo(t5, t9)); --- rat can move through the river


      assertTrue(r1.canMoveTo(t1, t7)); -- horizontal movement eating an elephant

      assertTrue(r1.canMoveTo(t1, t8) = false); -- horizontal movement but with a
stronger creature

      );

        private testTigerMovement: () ==> ()
             testTigerMovement() ==
              (
                   dcl t: Piece := new Tiger(<Red>, 6);
                   dcl r: Piece := new Rat(<Blue>, 1);
                   dcl e: Piece := new Elephant(<Blue>, 8);

                   dcl t1:Tile := new Tile([2,3],t); ---lion near the river vertically
                   dcl t8:Tile := new Tile([1,4],t); ---lion near the river horizontally
                   dcl t10:Tile := new Tile([1,5],t); ---lion near the river
horizontally but unable to jump

                   dcl t2:Tile := new Tile([2,2]);
                   dcl t3:Tile := new Tile([3,3]);
                   dcl t4:Tile := new Tile([1,4]);

                   dcl t5:Tile := new Tile([2,7]);

                   dcl t6:Tile := new Tile([3,3],r);
                   dcl t7:Tile := new Tile([3,3],e);

                   dcl t9:Tile := new Tile([4,4]);

                   dcl t11:Tile := new Tile([4,5], r);

                   dcl t12:Tile := new Tile([3,4]); ----river horizontal movement
                   dcl t13:Tile := new Tile([2,5]); ----river vertical movement


      assertTrue(t.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(t.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(t.canMoveTo(t1, t4) = false); -- diagonal movement


      assertTrue(t.canMoveTo(t1, t5)); --- jump the river vertically

      assertTrue(t.canMoveTo(t1, t13)); --- jump the river vertically but fall in water

      assertTrue(t.canMoveTo(t8, t9)); --- jump the river horizontally

      assertTrue(t.canMoveTo(t8, t12)); --- jump the river horizontally but fall in
water


      assertTrue(t.canMoveTo(t10, t11)); --- jump the river horizontally eating a rat


      assertTrue(t.canMoveTo(t1, t6)); -- horizontal movement eating a weaker creature

      assertTrue(t.canMoveTo(t1, t7) = false); -- horizontal movement but with a
stronger creature

      );

          private testWolfMovement: () ==> ()
             testWolfMovement() ==
             (
                   dcl w: Piece := new Wolf(<Red>, 4);
                   dcl r: Piece := new Rat(<Blue>, 1);
                   dcl l: Piece := new Lion(<Blue>, 7);

                   dcl t1:Tile := new Tile([1,1],w);
                   dcl t2:Tile := new Tile([1,2]);
                   dcl t3:Tile := new Tile([2,1]);
                   dcl t4:Tile := new Tile([2,2]);

                   dcl t5:Tile := new Tile([1,2],r);
                   dcl t6:Tile := new Tile([1,2],l);


      assertTrue(w.canMoveTo(t1, t2)); -- vertical movement

      assertTrue(w.canMoveTo(t1, t3)); -- horizontal movement

      assertTrue(w.canMoveTo(t1, t4) = false); -- diagonal movement
```

```
        assertTrue(w.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature

        assertTrue(w.canMoveTo(t1, t6) = false); -- horizontal movement but with a
stronger creature

        );

               private testMovement: () ==> ()
                testMovement() ==
                (

                dcl p: Piece := new Lion(<Blue>, 7);
                dcl p2: Piece := new Rat(<Red>, 1);
                dcl p3: Piece := new Wolf(<Red>, 4);

                dcl p4: Piece := new Lion(<Blue>, 7);


                dcl tl1: Tile := new Tile([1,1], p);
                dcl tl2: Tile := new Tile([2,1]);

                dcl tl3: Tile := new Tile([1,1], p2);
                dcl tl4: Tile := new Tile([2,1]);

                dcl tl5: Tile := new Tile([1,1], p3);
                dcl tl6: Tile := new Tile([2,2]);
                dcl tl7: Tile := new Tile([1,2]);

                dcl tl8: Tile := new Tile([2,3], p4);
                dcl tl9: Tile := new Tile([2,7]);
                dcl tl12: Tile := new Tile([2,8]);

                dcl tl10: Tile := new Tile([1,4], p4);
                dcl tl11: Tile := new Tile([4,4]);
                dcl tl13: Tile := new Tile([5,4]);

                assertTrue(p.movePieceTo(tl1, tl2));
                assertTrue((tl2.getPiece() = nil) = false);
                assertTrue(tl1.getPiece() = nil);

                assertTrue(p2.movePieceTo(tl3, tl4));
                assertTrue((tl4.getPiece() = nil) = false);
                assertTrue(tl3.getPiece() = nil);

                assertTrue(p3.movePieceTo(tl5, tl6) = false); ----extra testing for wrong
movement

                assertTrue(p3.movePieceTo(tl5, tl7));
                assertTrue((tl7.getPiece() = nil) = false);
                assertTrue(tl5.getPiece() = nil);
                assertTrue(p3.movePieceTo(tl7, tl6));
                assertTrue((tl6.getPiece() = nil) = false);
                assertTrue(tl7.getPiece() = nil);


                ----extra testing for river movements
                assertTrue(p4.isAllRiverV(tl8, tl9));
```

```
             assertTrue(p4.isAllRiverV(tl8, tl12) = false);

             assertTrue(p4.isAllRiverH(tl10, tl11));
             assertTrue(p4.isAllRiverH(tl10, tl13) = false);


      );

      private testGameTurns: () ==> ()
             testGameTurns() ==
              (

                   dcl g: Game := new Game();

                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.getcurrentPlayer() = <Player1>);
                   g.changePlayer();
                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.getcurrentPlayer() = <Player2>);
                   g.changePlayer();
                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.getcurrentPlayer() = <Player1>);


                   -----extra test for bad movement
                   assertTrue(g.movePieceTo([2,1],[2,2]) = false);


      );

      private testGameEndPlayer1: () ==> ()
             testGameEndPlayer1() ==
             (

                   dcl g: Game := new Game();

                   assertTrue(isofclass(Leopard,
g.getBoard().getTile([3,3]).getPiece()));

                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.movePieceTo([3,3],[4,3]));
                   g.changePlayer(); g.changePlayer();

                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.movePieceTo([4,3],[4,4]));
                   g.changePlayer(); g.changePlayer();

                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.movePieceTo([4,4],[4,5]));
                   g.changePlayer(); g.changePlayer();

                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.movePieceTo([4,5],[4,6]));
                   g.changePlayer(); g.changePlayer();

                   assertTrue(g.checkGameOver() = false);
                   assertTrue(g.movePieceTo([4,6],[4,7]));
                   g.changePlayer(); g.changePlayer();
```

```
                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,7],[4,8]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,8],[4,9]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver());


    );

    private testGameEndPlayer2: () ==> ()
          testGameEndPlayer2() ==
          (

                dcl g: Game := new Game();

                g.changePlayer();

                assertTrue(isofclass(Wolf, g.getBoard().getTile([3,7]).getPiece()));

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([3,7],[4,7]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,7],[4,6]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,6],[4,5]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,5],[4,4]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,4],[4,3]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,3],[4,2]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver() = false);
                assertTrue(g.movePieceTo([4,2],[4,1]));
                g.changePlayer(); g.changePlayer();

                assertTrue(g.checkGameOver());

);
```

```
    public static main: () ==> ()
            main() ==
            (

            ------------- Movement Tests -------------------------------------------
                    new GameTest().testCatMovement();
                    new GameTest().testDogMovement();
                    new GameTest().testElephantMovement();
                    new GameTest().testTigerMovement();
                    new GameTest().testRatMovement();
                    new GameTest().testLionMovement();
                    new GameTest().testLeopardMovement();
                    new GameTest().testWolfMovement();

                    new GameTest().testMovement();

            ------------- Game Tests --------------------------------------------


                    new GameTest().testGameTurns();
                    new GameTest().testGameEndPlayer1();
                    new GameTest().testGameEndPlayer2();
            );


end GameTest
```

In annex, it can be found the generated content in overture regarding the code implemented and the coverage tables for each of the classes and its functions.

5. Conclusions

After a thorough analysis of the requirements set in the beginning and the sources consulted, we can say that the final product meets all the expectations and ultimately the final goal: a correct formal model of the Chinese game board called Xo Dou Qi. Taking advantage of the potentialities of the Overture IDE, besides the test coverage analysis, UML models (shown above) and JAVA code were also generated. The latter, regarding a console implementation of the game, is 100% playable and meets all the rules previously stated. It served to assert that, after a thorough implementation using VDM++, without much effort, we can have a simple yet concise implementation of the final product in a well-known language as is JAVA.

The project took approximately 8 to 10 hours to develop and does not present any known nor visible bugs. As it was not required, no GUI was developed, but as a future improvement it could be something to think about and ultimately improve the final product.

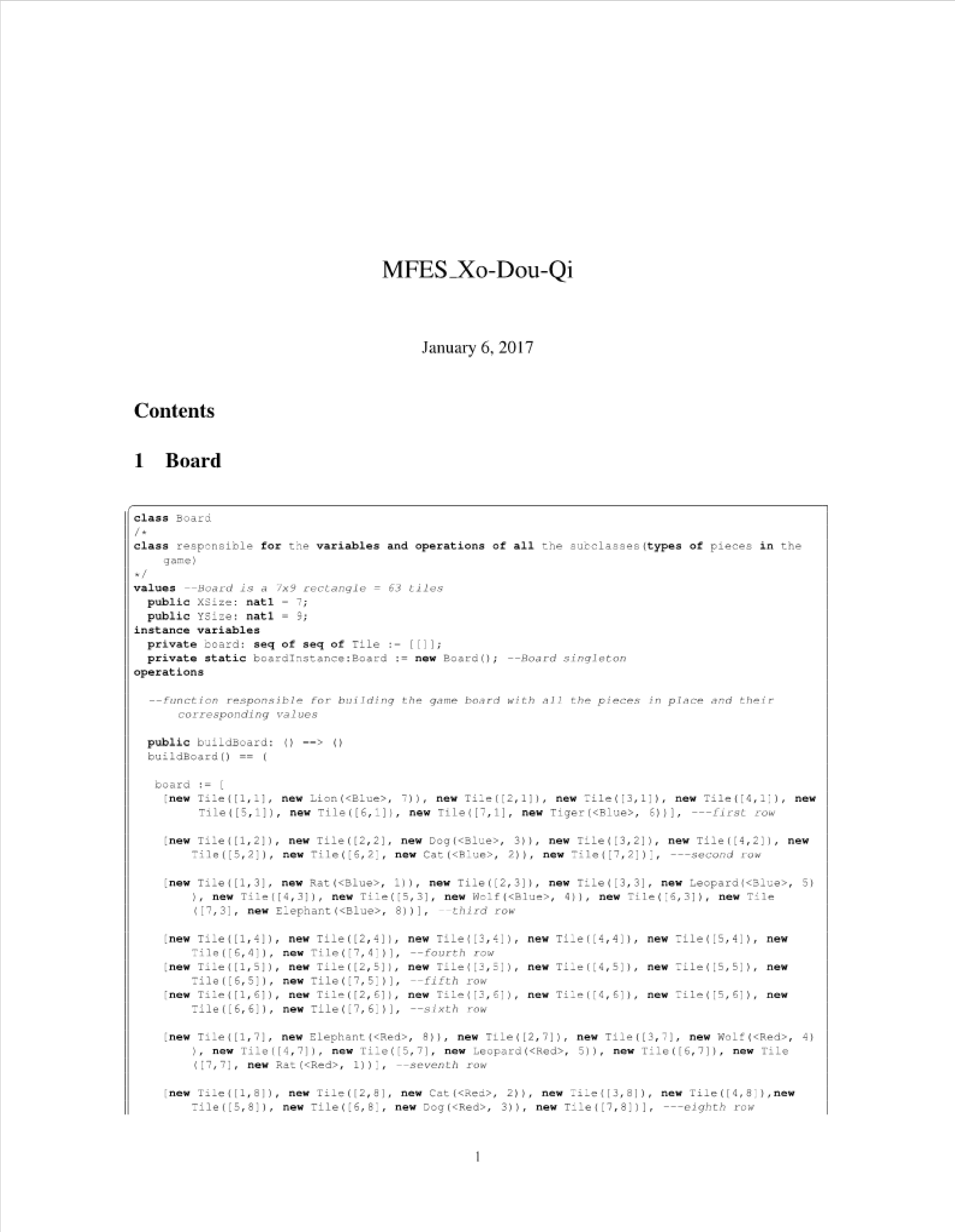
6. References

1. Game rules, https://pt.wikipedia.org/wiki/Xo_Dou_Qi
2. Extra game information, http://www.chessvariants.com/other.dir/animal.html

3. http://overturetool.org
4. Larsen, P.G. et. Al., 2016. VDM-10 Language Manual. 10th ed.
5. Faria, J.P. (s.d.). Formal Modeling of a Vending Machine in VDM++

7. Annex

Bellow is, as previously stated, the .pdf document generated by Overture regarding the coverage of each class and all their functions.



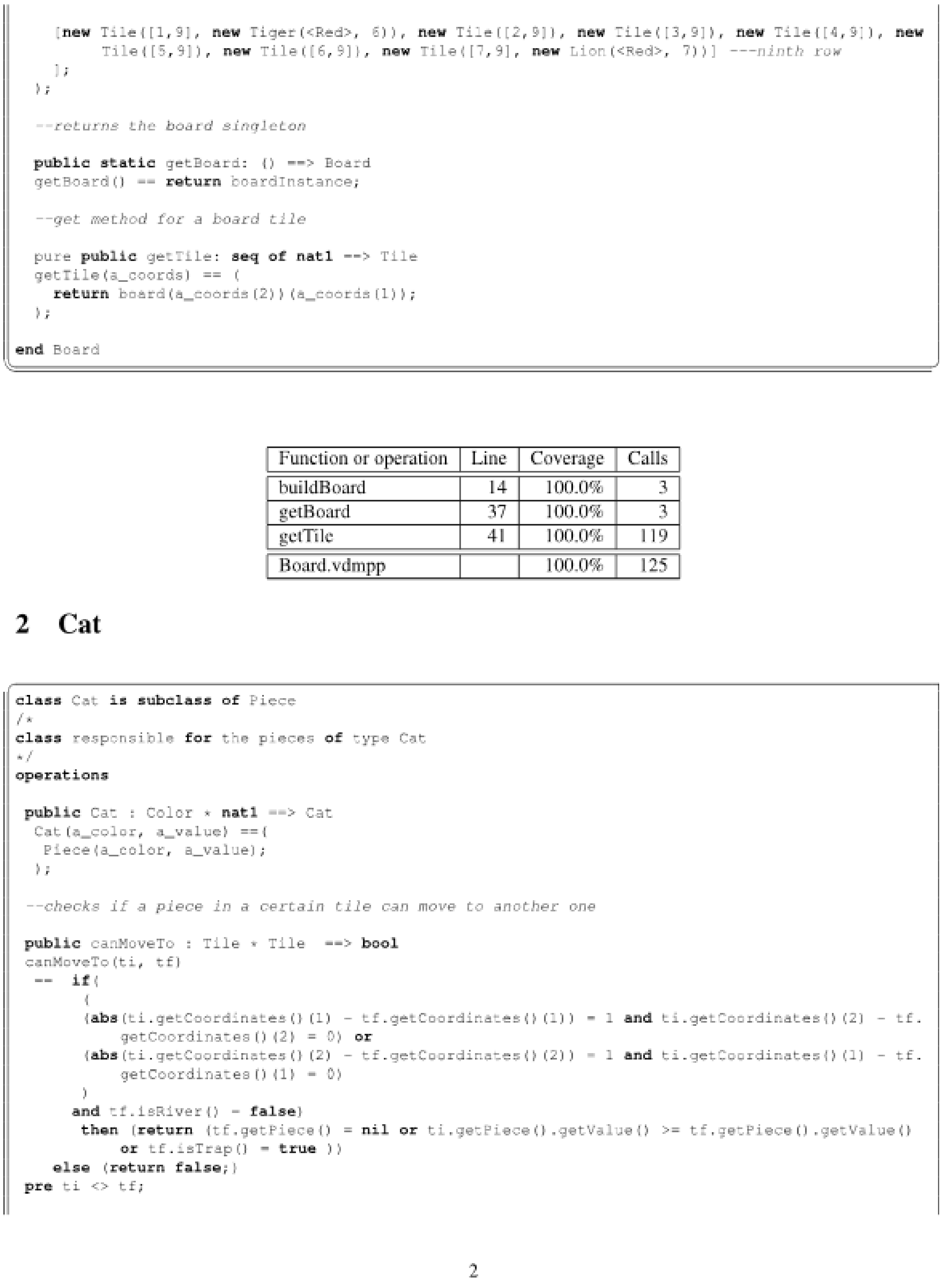
MFES_Xo-Dou-Qi

January 6, 2017

Contents

1 Board

```
class Board
/*
class responsible for the variables and operations of all the subclasses(types of pieces in the
    game)
*/
values --Board is a 7x9 rectangle = 63 tiles
  public XSize: nat1 = 7;
  public YSize: nat1 = 9;
instance variables
  private board: seq of seq of Tile := [[]];
  private static boardInstance:Board := new Board(); --Board singleton
operations

  --function responsible for building the game board with all the pieces in place and their
      corresponding values

  public buildBoard: () ==> ()
  buildBoard() == (

   board := [
    [new Tile([1,1], new Lion(<Blue>, 7)), new Tile([2,1]), new Tile([3,1]), new Tile([4,1]), new
         Tile([5,1]), new Tile([6,1]), new Tile([7,1], new Tiger(<Blue>, 6))], ---first row

    [new Tile([1,2]), new Tile([2,2], new Dog(<Blue>, 3)), new Tile([3,2]), new Tile([4,2]), new
        Tile([5,2]), new Tile([6,2], new Cat(<Blue>, 2)), new Tile([7,2])], ---second row

    [new Tile([1,3], new Rat(<Blue>, 1)), new Tile([2,3]), new Tile([3,3], new Leopard(<Blue>, 5)
        ), new Tile([4,3]), new Tile([5,3], new Wolf(<Blue>, 4)), new Tile([6,3]), new Tile
        ([7,3], new Elephant(<Blue>, 8))], --third row

    [new Tile([1,4]), new Tile([2,4]), new Tile([3,4]), new Tile([4,4]), new Tile([5,4]), new
        Tile([6,4]), new Tile([7,4])], --fourth row
    [new Tile([1,5]), new Tile([2,5]), new Tile([3,5]), new Tile([4,5]), new Tile([5,5]), new
        Tile([6,5]), new Tile([7,5])], --fifth row
    [new Tile([1,6]), new Tile([2,6]), new Tile([3,6]), new Tile([4,6]), new Tile([5,6]), new
        Tile([6,6]), new Tile([7,6])], --sixth row

    [new Tile([1,7], new Elephant(<Red>, 8)), new Tile([2,7]), new Tile([3,7], new Wolf(<Red>, 4)
        ), new Tile([4,7]), new Tile([5,7], new Leopard(<Red>, 5)), new Tile([6,7]), new Tile
        ([7,7], new Rat(<Red>, 1))], --seventh row

    [new Tile([1,8]), new Tile([2,8], new Cat(<Red>, 2)), new Tile([3,8]), new Tile([4,8]),new
        Tile([5,8]), new Tile([6,8], new Dog(<Red>, 3)), new Tile([7,8])], ---eighth row
```

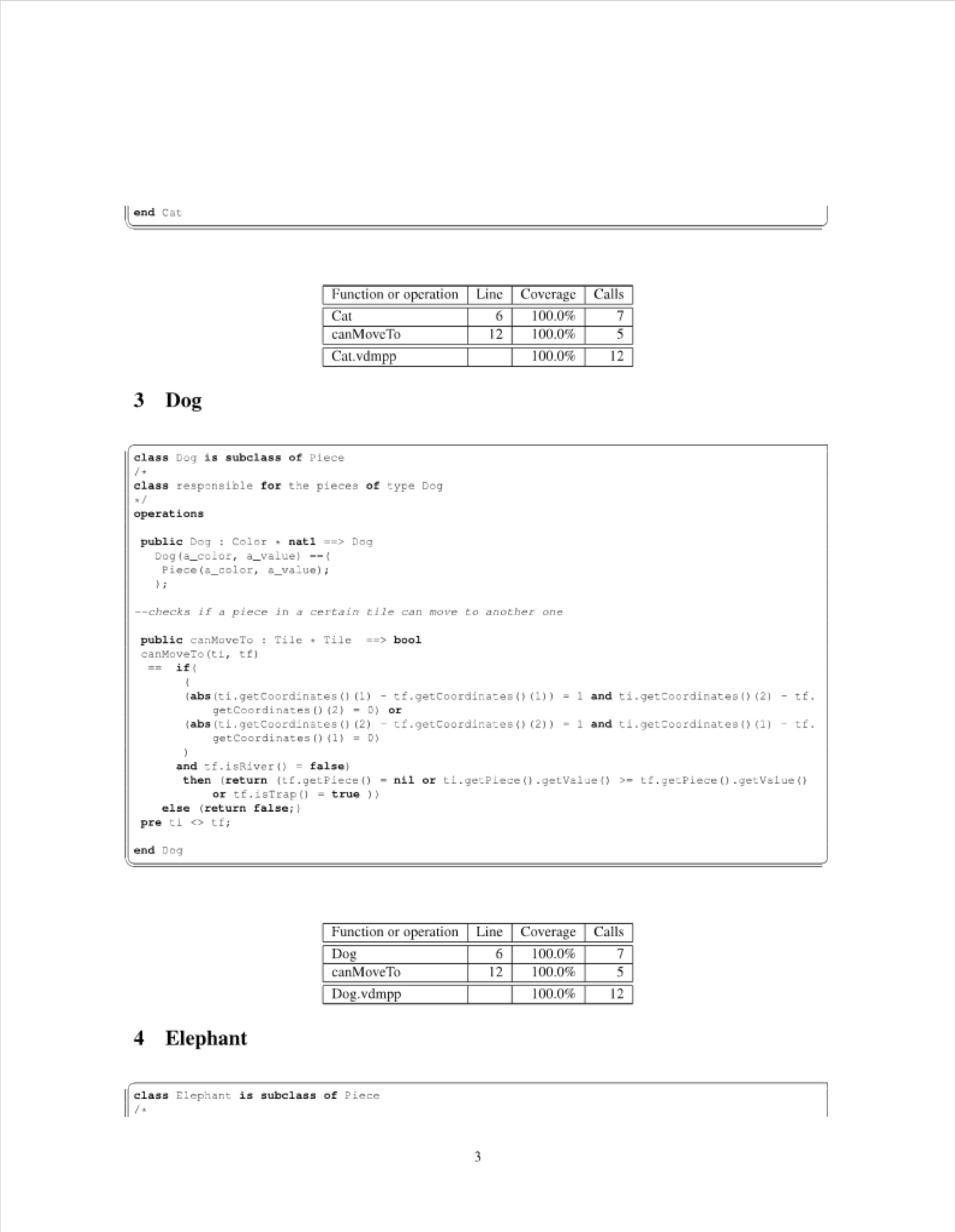


```
    [new Tile([1,9], new Tiger(<Red>, 6)), new Tile([2,9]), new Tile([3,9]), new Tile([4,9]), new
         Tile([5,9]), new Tile([6,9]), new Tile([7,9], new Lion(<Red>, 7))] ---ninth row
    ];
  );

  --returns the board singleton

  public static getBoard: () ==> Board
  getBoard() == return boardInstance;

  --get method for a board tile

  pure public getTile: seq of nat1 ==> Tile
  getTile(a_coords) == (
    return board(a_coords(2))(a_coords(1));
  );

end Board
```

Function or operation	Line	Coverage	Calls
buildBoard	14	100.0%	3
getBoard	37	100.0%	3
getTile	41	100.0%	119
Board.vdmpp		100.0%	125

2 Cat

```
class Cat is subclass of Piece
/*
class responsible for the pieces of type Cat
*/
operations

 public Cat : Color * nat1 ==> Cat
  Cat(a_color, a_value) ==(
   Piece(a_color, a_value);
  );

 --checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  --  if(
       (
       (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
           getCoordinates()(2) = 0) or
       (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
           getCoordinates()(1) = 0)
       )
      and tf.isRiver() = false)
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue()
           or tf.isTrap() = true ))
    else (return false;)
 pre ti <> tf;
```



```
end Cat
```

Function or operation	Line	Coverage	Calls
Cat	6	100.0%	7
canMoveTo	12	100.0%	5
Cat.vdmpp		100.0%	12

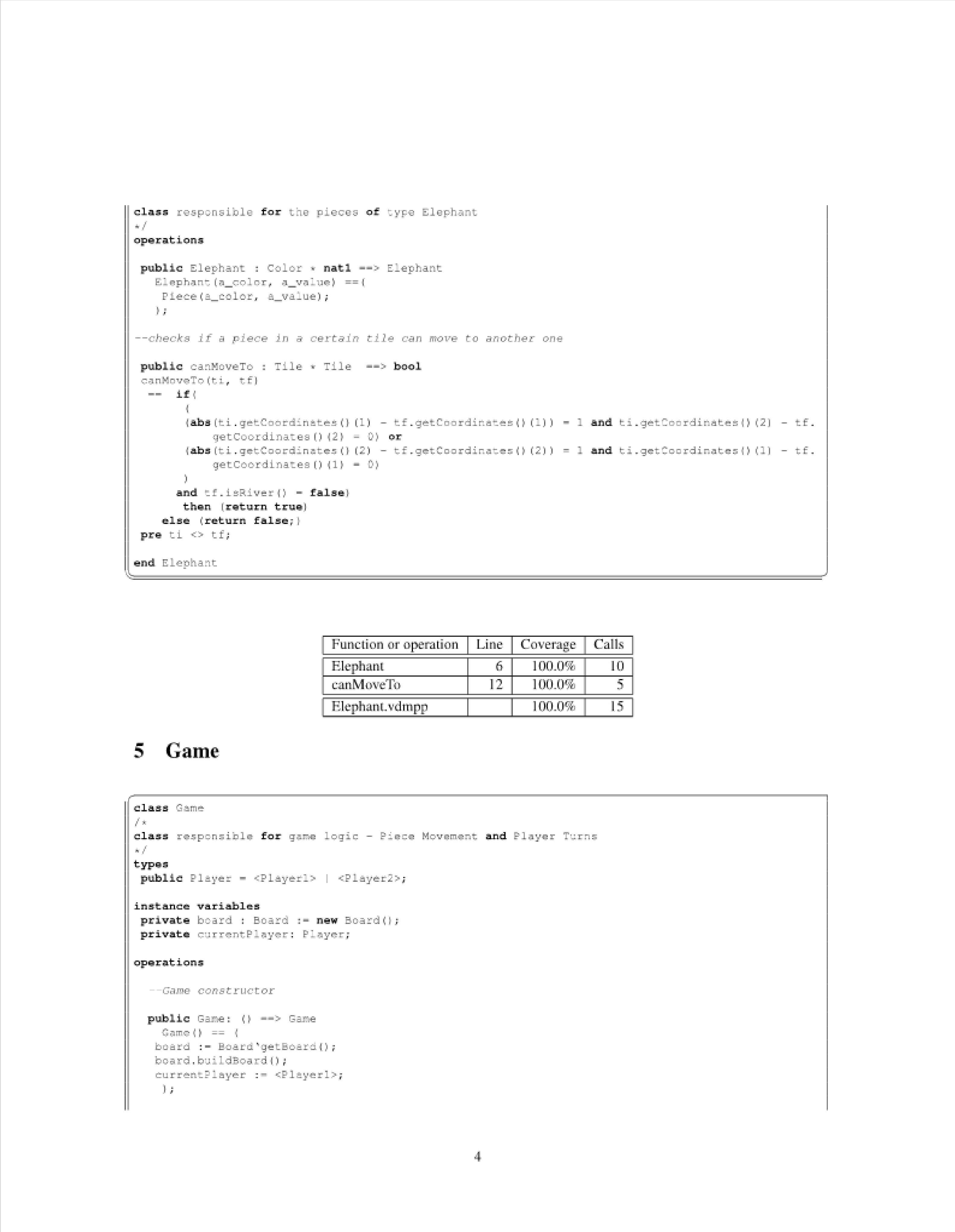
3 Dog

```
class Dog is subclass of Piece
/*
class responsible for the pieces of type Dog
*/
operations

 public Dog : Color * natl ==> Dog
   Dog(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

--checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  ==  if(
       (
       (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
           getCoordinates()(2) = 0) or
       (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
           getCoordinates()(1) = 0)
       )
      and tf.isRiver() = false)
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue()
           or tf.isTrap() = true ))
    else (return false;)
 pre ti <> tf;

end Dog
```

Function or operation	Line	Coverage	Calls
Dog	6	100.0%	7
canMoveTo	12	100.0%	5
Dog.vdmpp		100.0%	12

4 Elephant

```
class Elephant is subclass of Piece
/*
```

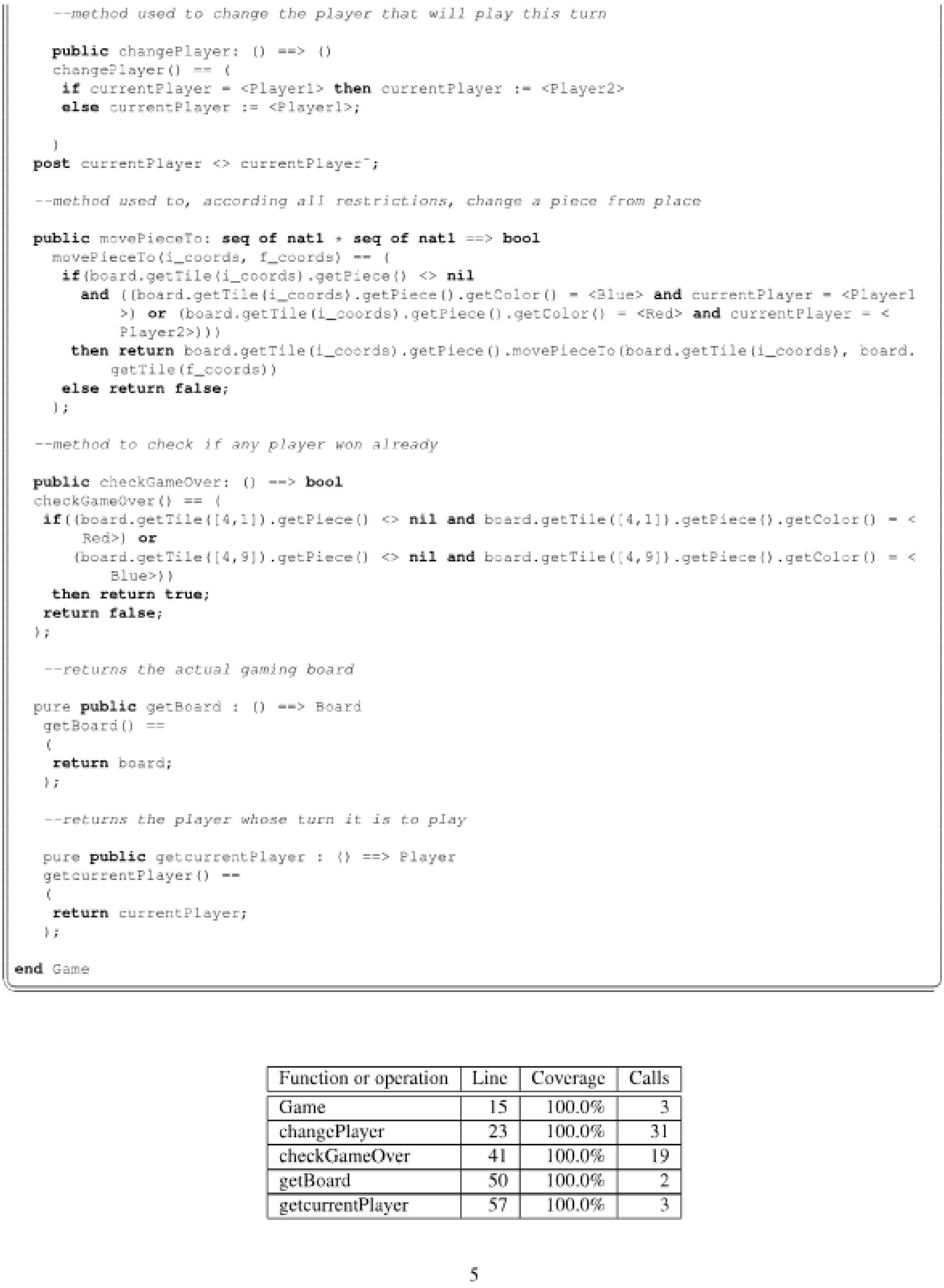


```
class responsible for the pieces of type Elephant
*/
operations

 public Elephant : Color * nat1 ==> Elephant
   Elephant(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

--checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  --  if(
        (
        (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
            getCoordinates()(2) = 0) or
        (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
            getCoordinates()(1) = 0)
        )
      and tf.isRiver() = false)
       then (return true)
    else (return false;)
 pre ti <> tf;

end Elephant
```

Function or operation	Line	Coverage	Calls
Elephant	6	100.0%	10
canMoveTo	12	100.0%	5
Elephant.vdmpp		100.0%	15

5 Game

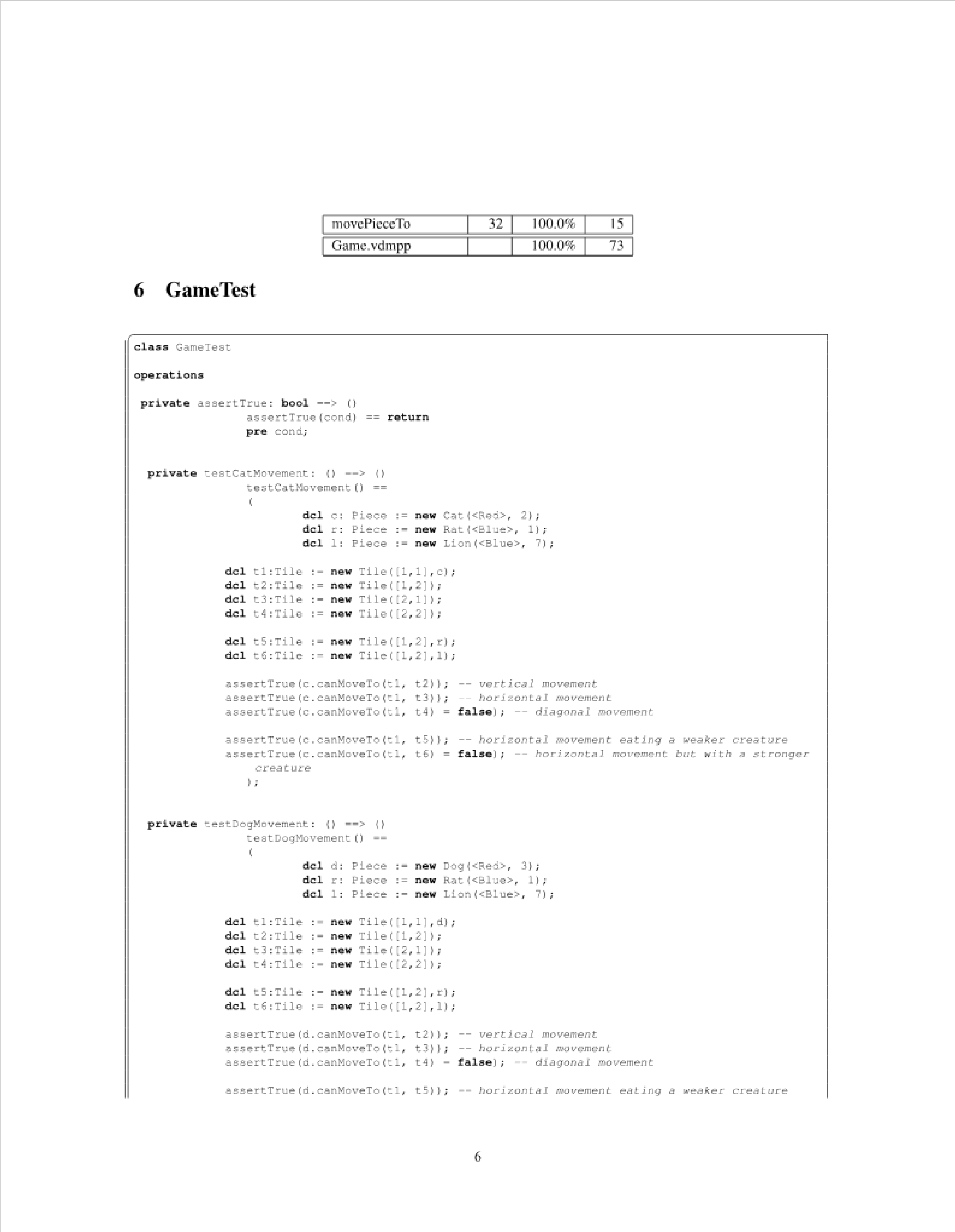
```
class Game
/*
class responsible for game logic - Piece Movement and Player Turns
*/
types
 public Player = <Player1> | <Player2>;

instance variables
 private board : Board := new Board();
 private currentPlayer: Player;

operations

  --Game constructor

  public Game: () ==> Game
    Game() == (
   board := Board`getBoard();
   board.buildBoard();
   currentPlayer := <Player1>;
    );
```



```
   --method used to change the player that will play this turn

   public changePlayer: () ==> ()
   changePlayer() == (
    if currentPlayer = <Player1> then currentPlayer := <Player2>
    else currentPlayer := <Player1>;

   )
 post currentPlayer <> currentPlayer~;

 --method used to, according all restrictions, change a piece from place

 public movePieceTo: seq of nat1 * seq of nat1 ==> bool
   movePieceTo(i_coords, f_coords) == (
    if(board.getTile(i_coords).getPiece() <> nil
      and ((board.getTile(i_coords).getPiece().getColor() = <Blue> and currentPlayer = <Player1
          >) or (board.getTile(i_coords).getPiece().getColor() = <Red> and currentPlayer = <
          Player2>)))
     then return board.getTile(i_coords).getPiece().movePieceTo(board.getTile(i_coords), board.
         getTile(f_coords))
    else return false;
   );

 --method to check if any player won already

 public checkGameOver: () ==> bool
 checkGameOver() == (
  if((board.getTile([4,1]).getPiece() <> nil and board.getTile([4,1]).getPiece().getColor() = <
      Red>) or
     (board.getTile([4,9]).getPiece() <> nil and board.getTile([4,9]).getPiece().getColor() = <
         Blue>))
   then return true;
  return false;
 );

  --returns the actual gaming board

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

  --returns the player whose turn it is to play

  pure public getcurrentPlayer : () ==> Player
  getcurrentPlayer() ==
  (
   return currentPlayer;
  );

end Game
```

Function or operation	Line	Coverage	Calls
Game	15	100.0%	3
changePlayer	23	100.0%	31
checkGameOver	41	100.0%	19
getBoard	50	100.0%	2
getcurrentPlayer	57	100.0%	3

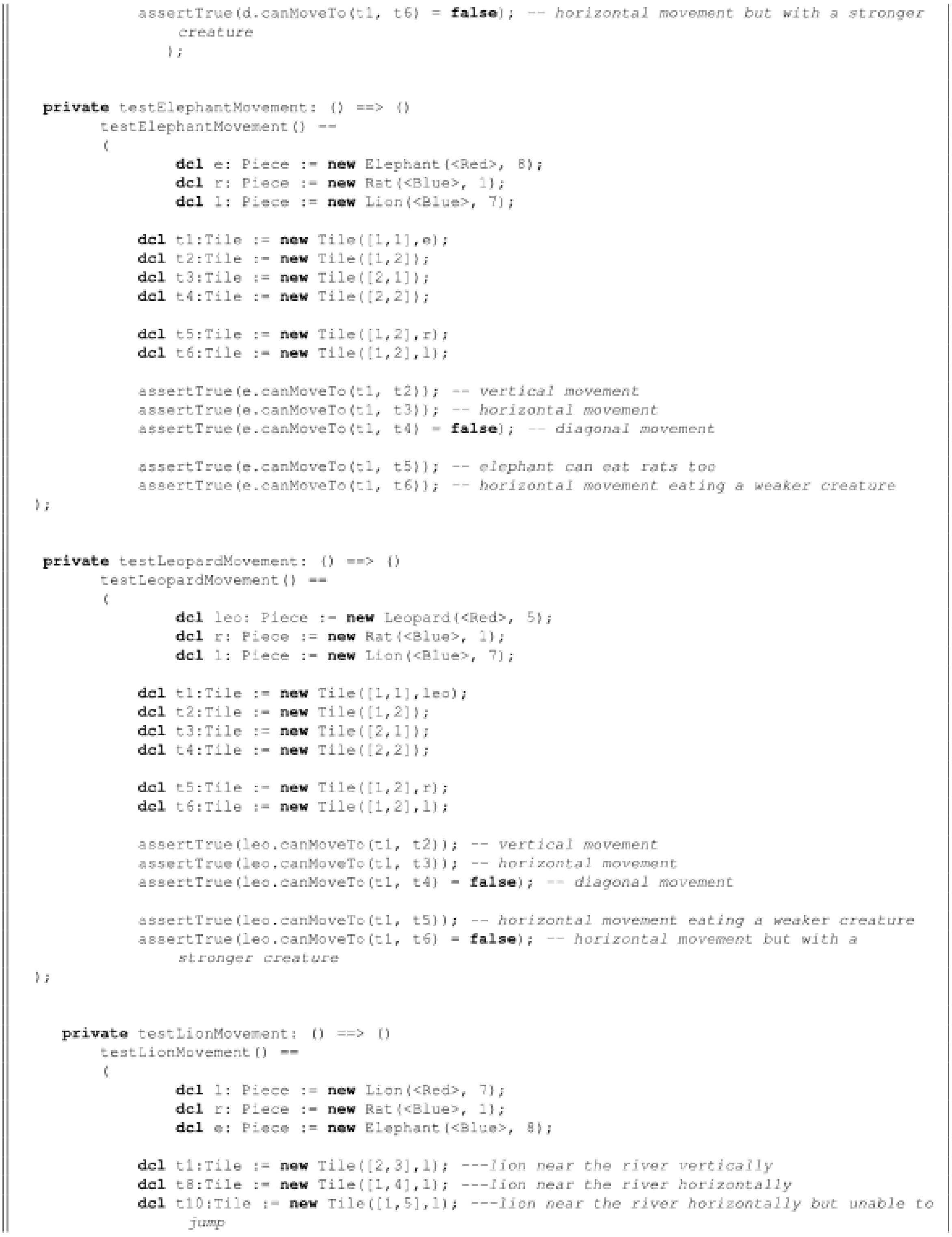
5



movePieceTo	32	100.0%	15
Game.vdmpp		100.0%	73

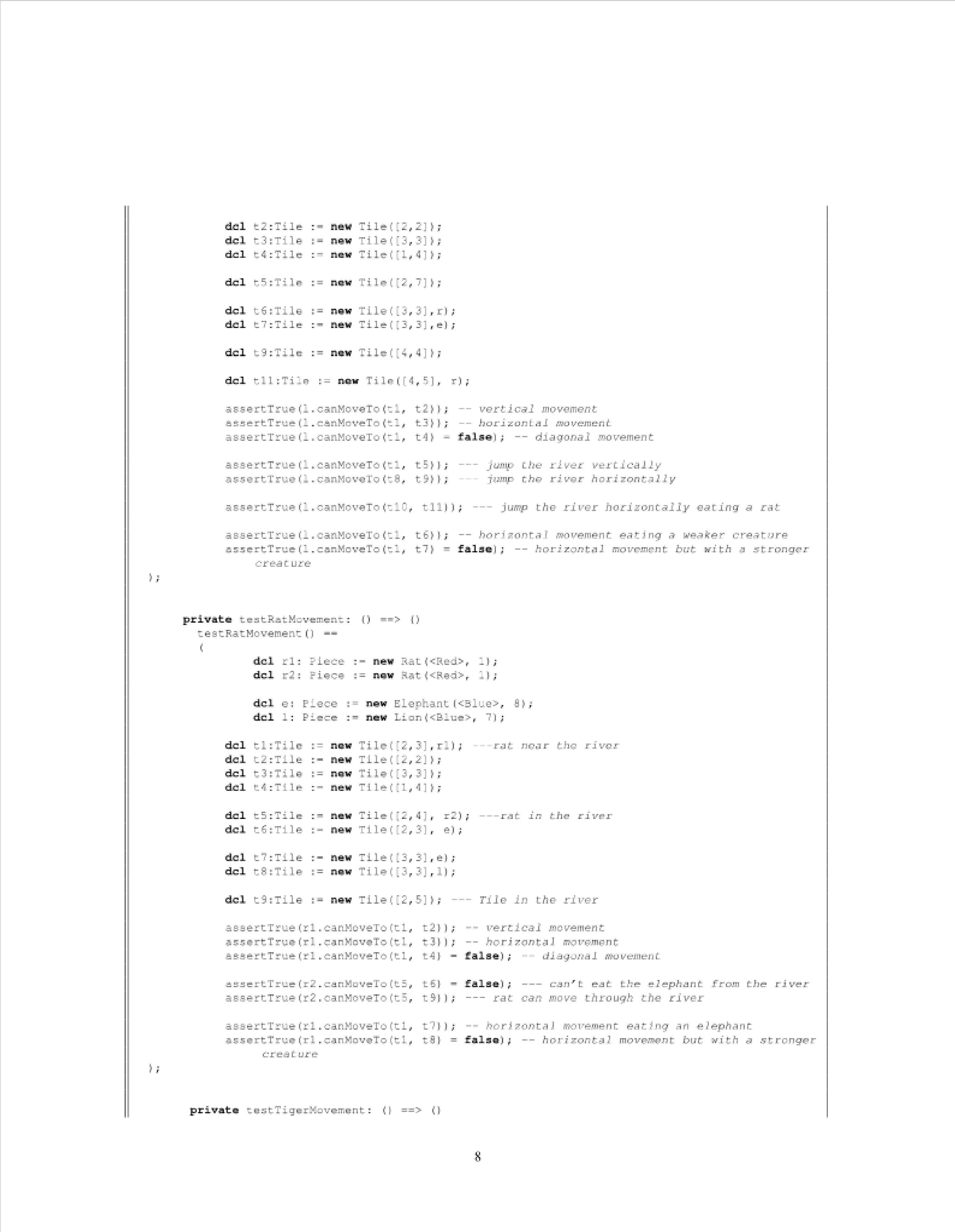
6 GameTest

```
class GameTest

operations

 private assertTrue: bool ==> ()
               assertTrue(cond) == return
               pre cond;


  private testCatMovement: () ==> ()
               testCatMovement() ==
               (
                       dcl c: Piece := new Cat(<Red>, 2);
                       dcl r: Piece := new Rat(<Blue>, 1);
                       dcl l: Piece := new Lion(<Blue>, 7);

            dcl t1:Tile := new Tile([1,1],c);
            dcl t2:Tile := new Tile([1,2]);
            dcl t3:Tile := new Tile([2,1]);
            dcl t4:Tile := new Tile([2,2]);

            dcl t5:Tile := new Tile([1,2],r);
            dcl t6:Tile := new Tile([1,2],l);

            assertTrue(c.canMoveTo(t1, t2)); -- vertical movement
            assertTrue(c.canMoveTo(t1, t3)); -- horizontal movement
            assertTrue(c.canMoveTo(t1, t4) = false); -- diagonal movement

            assertTrue(c.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature
            assertTrue(c.canMoveTo(t1, t6) = false); -- horizontal movement but with a stronger
                creature
               );


  private testDogMovement: () ==> ()
               testDogMovement() ==
               (
                       dcl d: Piece := new Dog(<Red>, 3);
                       dcl r: Piece := new Rat(<Blue>, 1);
                       dcl l: Piece := new Lion(<Blue>, 7);

            dcl t1:Tile := new Tile([1,1],d);
            dcl t2:Tile := new Tile([1,2]);
            dcl t3:Tile := new Tile([2,1]);
            dcl t4:Tile := new Tile([2,2]);

            dcl t5:Tile := new Tile([1,2],r);
            dcl t6:Tile := new Tile([1,2],l);

            assertTrue(d.canMoveTo(t1, t2)); -- vertical movement
            assertTrue(d.canMoveTo(t1, t3)); -- horizontal movement
            assertTrue(d.canMoveTo(t1, t4) = false); -- diagonal movement

            assertTrue(d.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature
```

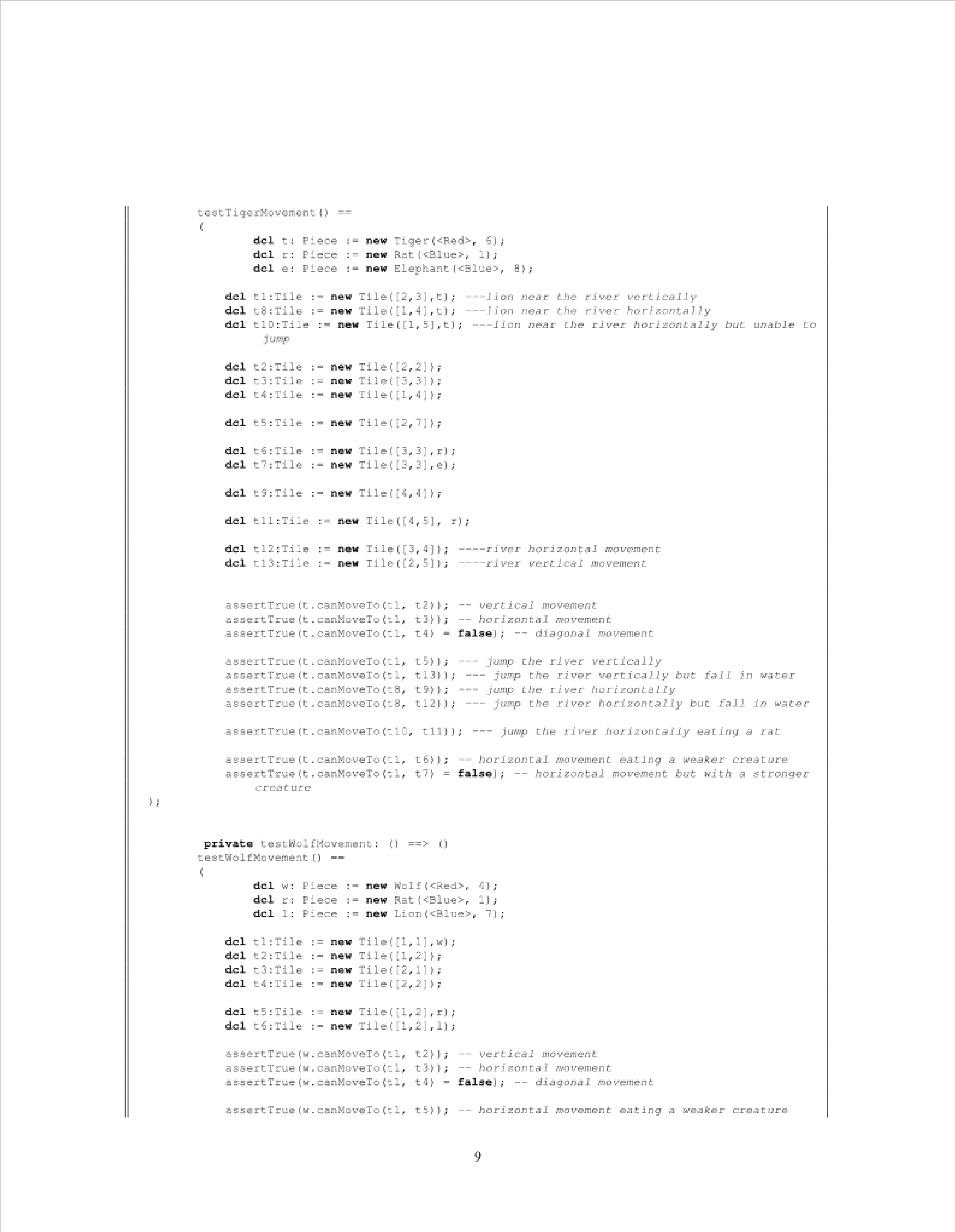


```
          assertTrue(d.canMoveTo(t1, t6) = false); -- horizontal movement but with a stronger
              creature
             );


 private testElephantMovement: () ==> ()
       testElephantMovement() ==
       (
              dcl e: Piece := new Elephant(<Red>, 8);
              dcl r: Piece := new Rat(<Blue>, 1);
              dcl l: Piece := new Lion(<Blue>, 7);

          dcl t1:Tile := new Tile([1,1],e);
          dcl t2:Tile := new Tile([1,2]);
          dcl t3:Tile := new Tile([2,1]);
          dcl t4:Tile := new Tile([2,2]);

          dcl t5:Tile := new Tile([1,2],r);
          dcl t6:Tile := new Tile([1,2],l);

          assertTrue(e.canMoveTo(t1, t2)); -- vertical movement
          assertTrue(e.canMoveTo(t1, t3)); -- horizontal movement
          assertTrue(e.canMoveTo(t1, t4) = false); -- diagonal movement

          assertTrue(e.canMoveTo(t1, t5)); -- elephant can eat rats too
          assertTrue(e.canMoveTo(t1, t6)); -- horizontal movement eating a weaker creature
);


 private testLeopardMovement: () ==> ()
       testLeopardMovement() ==
       (
              dcl leo: Piece := new Leopard(<Red>, 5);
              dcl r: Piece := new Rat(<Blue>, 1);
              dcl l: Piece := new Lion(<Blue>, 7);

          dcl t1:Tile := new Tile([1,1],leo);
          dcl t2:Tile := new Tile([1,2]);
          dcl t3:Tile := new Tile([2,1]);
          dcl t4:Tile := new Tile([2,2]);

          dcl t5:Tile := new Tile([1,2],r);
          dcl t6:Tile := new Tile([1,2],l);

          assertTrue(leo.canMoveTo(t1, t2)); -- vertical movement
          assertTrue(leo.canMoveTo(t1, t3)); -- horizontal movement
          assertTrue(leo.canMoveTo(t1, t4) = false); -- diagonal movement

          assertTrue(leo.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature
          assertTrue(leo.canMoveTo(t1, t6) = false); -- horizontal movement but with a
              stronger creature
);


   private testLionMovement: () ==> ()
       testLionMovement() ==
       (
              dcl l: Piece := new Lion(<Red>, 7);
              dcl r: Piece := new Rat(<Blue>, 1);
              dcl e: Piece := new Elephant(<Blue>, 8);

          dcl t1:Tile := new Tile([2,3],l); ---lion near the river vertically
          dcl t8:Tile := new Tile([1,4],l); ---lion near the river horizontally
          dcl t10:Tile := new Tile([1,5],l); ---lion near the river horizontally but unable to
               jump
```

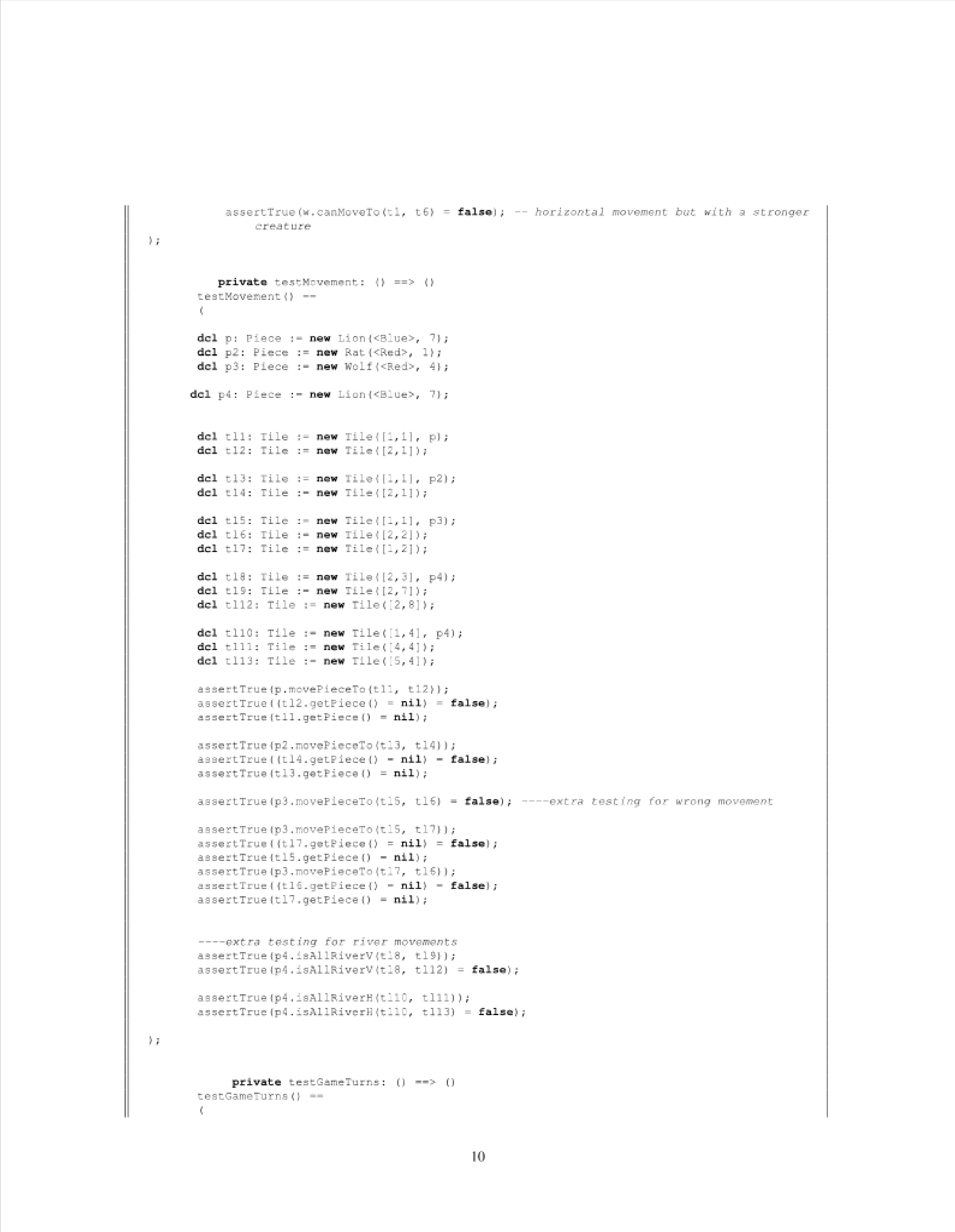
7



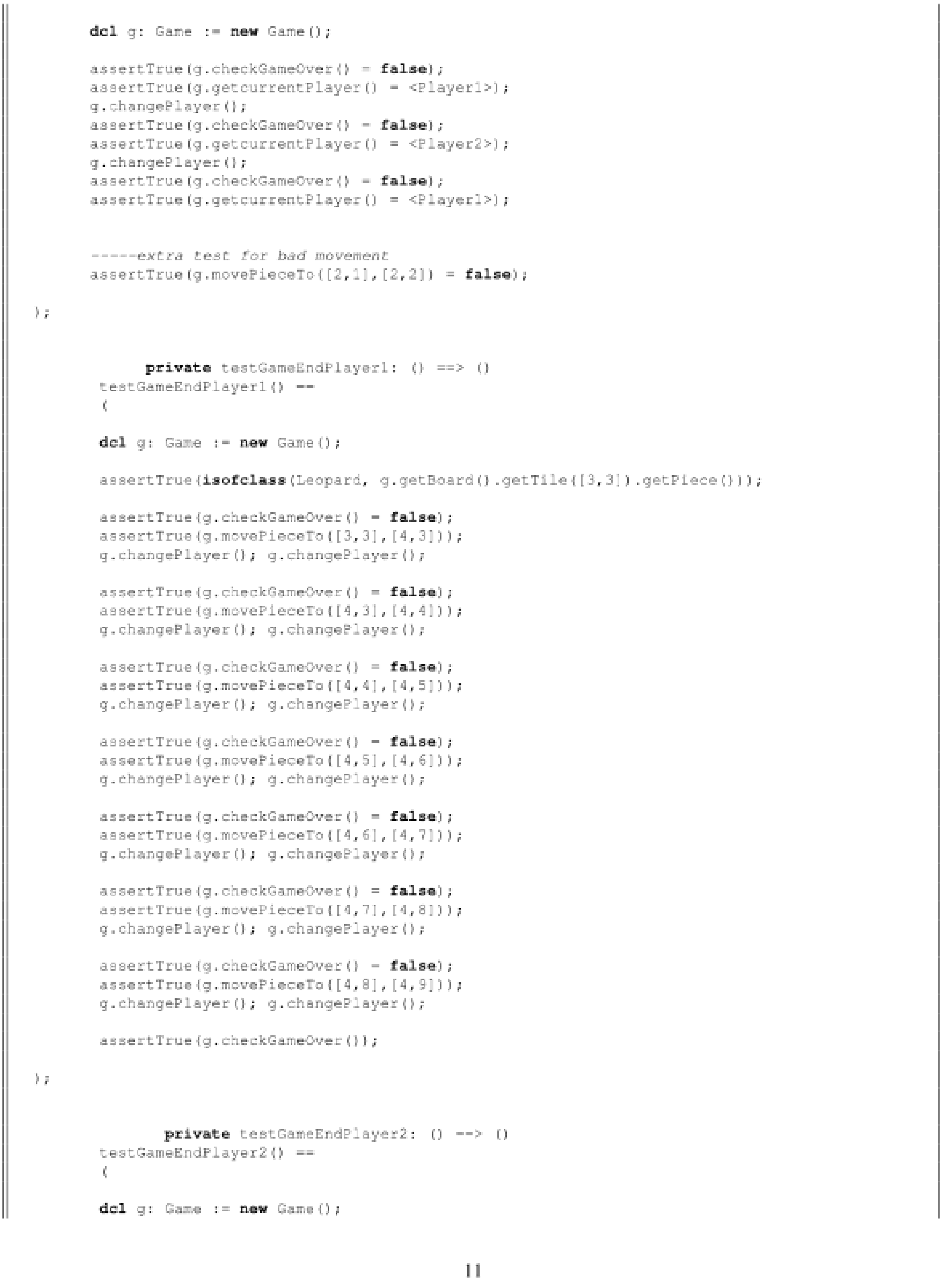
```
           dcl t2:Tile := new Tile([2,2]);
           dcl t3:Tile := new Tile([3,3]);
           dcl t4:Tile := new Tile([1,4]);

           dcl t5:Tile := new Tile([2,7]);

           dcl t6:Tile := new Tile([3,3],r);
           dcl t7:Tile := new Tile([3,3],e);

           dcl t9:Tile := new Tile([4,4]);

           dcl t11:Tile := new Tile([4,5], r);

           assertTrue(l.canMoveTo(t1, t2)); -- vertical movement
           assertTrue(l.canMoveTo(t1, t3)); -- horizontal movement
           assertTrue(l.canMoveTo(t1, t4) = false); -- diagonal movement

           assertTrue(l.canMoveTo(t1, t5)); --- jump the river vertically
           assertTrue(l.canMoveTo(t8, t9)); --- jump the river horizontally

           assertTrue(l.canMoveTo(t10, t11)); --- jump the river horizontally eating a rat

           assertTrue(l.canMoveTo(t1, t6)); -- horizontal movement eating a weaker creature
           assertTrue(l.canMoveTo(t1, t7) = false); -- horizontal movement but with a stronger
               creature
);


     private testRatMovement: () ==> ()
       testRatMovement() ==
       (
               dcl r1: Piece := new Rat(<Red>, 1);
               dcl r2: Piece := new Rat(<Red>, 1);

               dcl e: Piece := new Elephant(<Blue>, 8);
               dcl l: Piece := new Lion(<Blue>, 7);

           dcl t1:Tile := new Tile([2,3],r1); ---rat near the river
           dcl t2:Tile := new Tile([2,2]);
           dcl t3:Tile := new Tile([3,3]);
           dcl t4:Tile := new Tile([1,4]);

           dcl t5:Tile := new Tile([2,4], r2); ---rat in the river
           dcl t6:Tile := new Tile([2,3], e);

           dcl t7:Tile := new Tile([3,3],e);
           dcl t8:Tile := new Tile([3,3],l);

           dcl t9:Tile := new Tile([2,5]); --- Tile in the river

           assertTrue(r1.canMoveTo(t1, t2)); -- vertical movement
           assertTrue(r1.canMoveTo(t1, t3)); -- horizontal movement
           assertTrue(r1.canMoveTo(t1, t4) = false); -- diagonal movement

           assertTrue(r2.canMoveTo(t5, t6) = false); --- can't eat the elephant from the river
           assertTrue(r2.canMoveTo(t5, t9)); --- rat can move through the river

           assertTrue(r1.canMoveTo(t1, t7)); -- horizontal movement eating an elephant
           assertTrue(r1.canMoveTo(t1, t8) = false); -- horizontal movement but with a stronger
                creature
);


      private testTigerMovement: () ==> ()
```



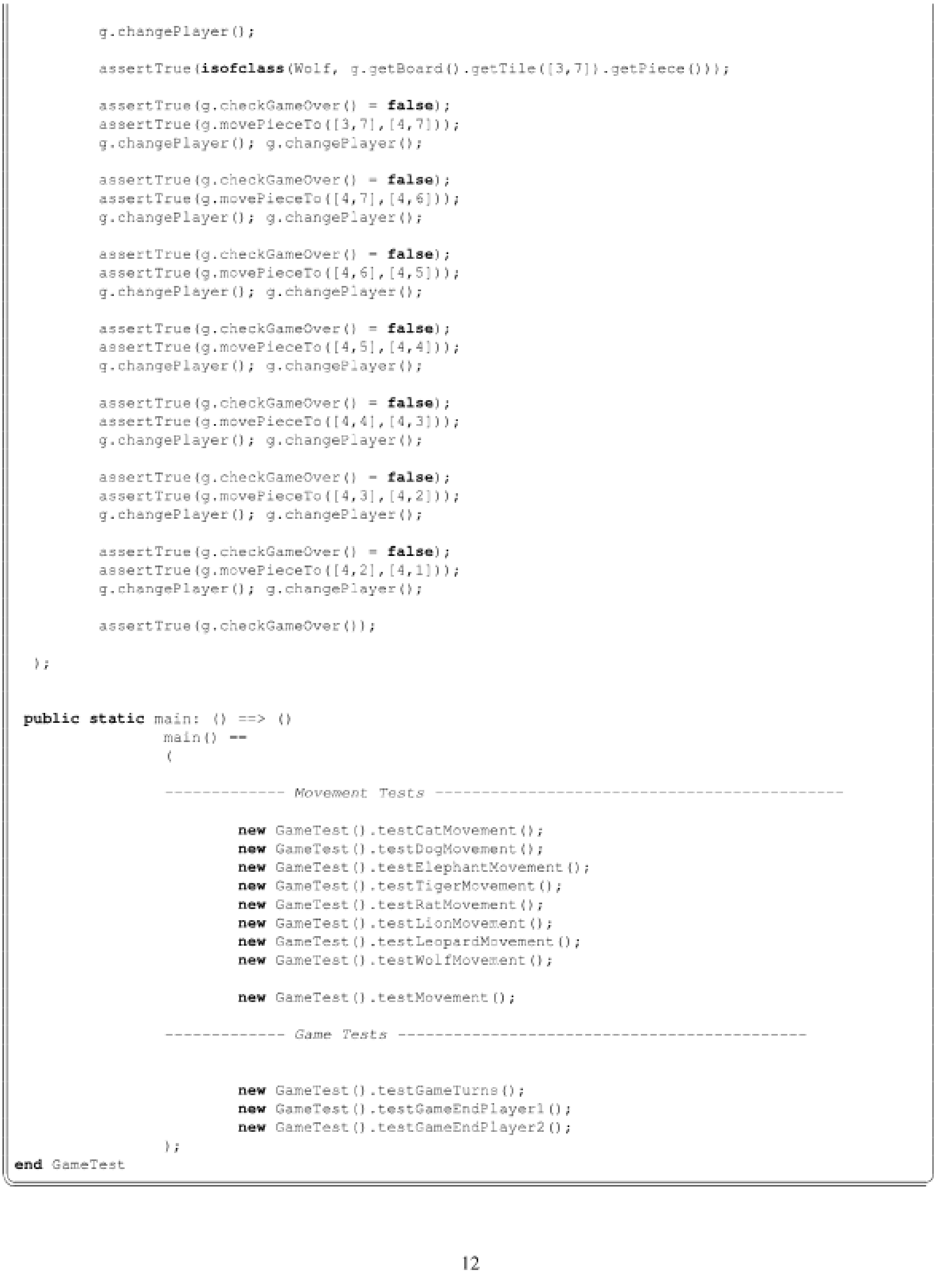
```
        testTigerMovement() ==
        (
                dcl t: Piece := new Tiger(<Red>, 6);
                dcl r: Piece := new Rat(<Blue>, 1);
                dcl e: Piece := new Elephant(<Blue>, 8);

            dcl t1:Tile := new Tile([2,3],t); ---lion near the river vertically
            dcl t8:Tile := new Tile([1,4],t); ---lion near the river horizontally
            dcl t10:Tile := new Tile([1,5],t); ---lion near the river horizontally but unable to
                 jump

            dcl t2:Tile := new Tile([2,2]);
            dcl t3:Tile := new Tile([3,3]);
            dcl t4:Tile := new Tile([1,4]);

            dcl t5:Tile := new Tile([2,7]);

            dcl t6:Tile := new Tile([3,3],r);
            dcl t7:Tile := new Tile([3,3],e);

            dcl t9:Tile := new Tile([4,4]);

            dcl t11:Tile := new Tile([4,5], r);

            dcl t12:Tile := new Tile([3,4]); ----river horizontal movement
            dcl t13:Tile := new Tile([2,5]); ----river vertical movement


            assertTrue(t.canMoveTo(t1, t2)); -- vertical movement
            assertTrue(t.canMoveTo(t1, t3)); -- horizontal movement
            assertTrue(t.canMoveTo(t1, t4) = false); -- diagonal movement

            assertTrue(t.canMoveTo(t1, t5)); --- jump the river vertically
            assertTrue(t.canMoveTo(t1, t13)); --- jump the river vertically but fall in water
            assertTrue(t.canMoveTo(t8, t9)); --- jump the river horizontally
            assertTrue(t.canMoveTo(t8, t12)); --- jump the river horizontally but fall in water

            assertTrue(t.canMoveTo(t10, t11)); --- jump the river horizontally eating a rat

            assertTrue(t.canMoveTo(t1, t6)); -- horizontal movement eating a weaker creature
            assertTrue(t.canMoveTo(t1, t7) = false); -- horizontal movement but with a stronger
                creature
);


         private testWolfMovement: () ==> ()
        testWolfMovement() --
        (
                dcl w: Piece := new Wolf(<Red>, 4);
                dcl r: Piece := new Rat(<Blue>, 1);
                dcl l: Piece := new Lion(<Blue>, 7);

            dcl t1:Tile := new Tile([1,1],w);
            dcl t2:Tile := new Tile([1,2]);
            dcl t3:Tile := new Tile([2,1]);
            dcl t4:Tile := new Tile([2,2]);

            dcl t5:Tile := new Tile([1,2],r);
            dcl t6:Tile := new Tile([1,2],l);

            assertTrue(w.canMoveTo(t1, t2)); -- vertical movement
            assertTrue(w.canMoveTo(t1, t3)); -- horizontal movement
            assertTrue(w.canMoveTo(t1, t4) = false); -- diagonal movement

            assertTrue(w.canMoveTo(t1, t5)); -- horizontal movement eating a weaker creature
```



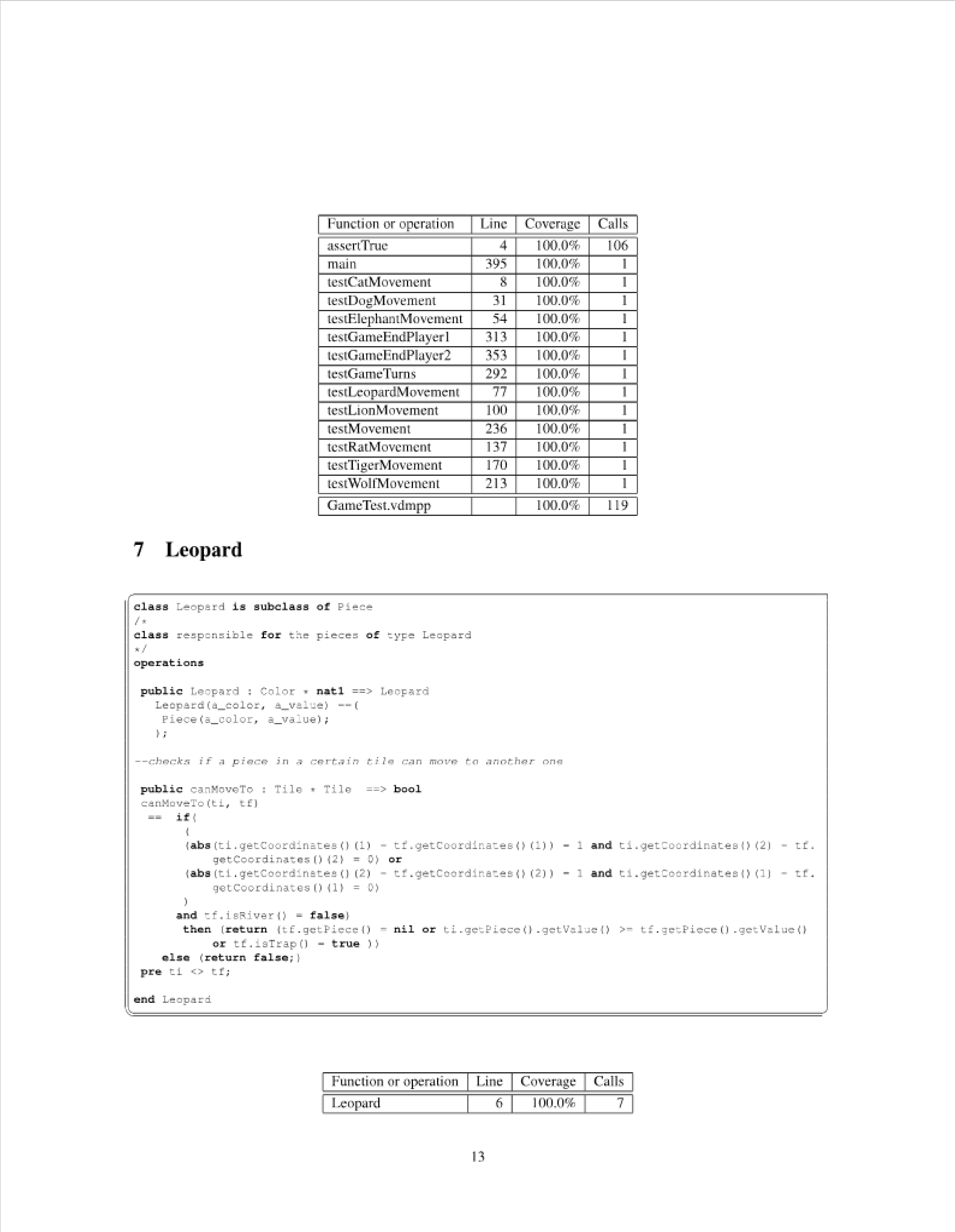
```
            assertTrue(w.canMoveTo(t1, t6) = false); -- horizontal movement but with a stronger
                creature
);


        private testMovement: () ==> ()
        testMovement() ==
        (

        dcl p: Piece := new Lion(<Blue>, 7);
        dcl p2: Piece := new Rat(<Red>, 1);
        dcl p3: Piece := new Wolf(<Red>, 4);

       dcl p4: Piece := new Lion(<Blue>, 7);


        dcl tl1: Tile := new Tile([1,1], p);
        dcl tl2: Tile := new Tile([2,1]);

        dcl tl3: Tile := new Tile([1,1], p2);
        dcl tl4: Tile := new Tile([2,1]);

        dcl tl5: Tile := new Tile([1,1], p3);
        dcl tl6: Tile := new Tile([2,2]);
        dcl tl7: Tile := new Tile([1,2]);

        dcl tl8: Tile := new Tile([2,3], p4);
        dcl tl9: Tile := new Tile([2,7]);
        dcl tl12: Tile := new Tile([2,8]);

        dcl tl10: Tile := new Tile([1,4], p4);
        dcl tl11: Tile := new Tile([4,4]);
        dcl tl13: Tile := new Tile([5,4]);

        assertTrue(p.movePieceTo(tl1, tl2));
        assertTrue((tl2.getPiece() = nil) = false);
        assertTrue(tl1.getPiece() = nil);

        assertTrue(p2.movePieceTo(tl3, tl4));
        assertTrue((tl4.getPiece() = nil) = false);
        assertTrue(tl3.getPiece() = nil);

        assertTrue(p3.movePieceTo(tl5, tl6) = false); ----extra testing for wrong movement

        assertTrue(p3.movePieceTo(tl5, tl7));
        assertTrue((tl7.getPiece() = nil) = false);
        assertTrue(tl5.getPiece() = nil);
        assertTrue(p3.movePieceTo(tl7, tl6));
        assertTrue((tl6.getPiece() = nil) = false);
        assertTrue(tl7.getPiece() = nil);


        ----extra testing for river movements
        assertTrue(p4.isAllRiverV(tl8, tl9));
        assertTrue(p4.isAllRiverV(tl8, tl12) = false);

        assertTrue(p4.isAllRiverH(tl10, tl11));
        assertTrue(p4.isAllRiverH(tl10, tl13) = false);

);


            private testGameTurns: () ==> ()
        testGameTurns() ==
        (
```



```
        dcl g: Game := new Game();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.getcurrentPlayer() = <Player1>);
        g.changePlayer();
        assertTrue(g.checkGameOver() = false);
        assertTrue(g.getcurrentPlayer() = <Player2>);
        g.changePlayer();
        assertTrue(g.checkGameOver() = false);
        assertTrue(g.getcurrentPlayer() = <Player1>);


        -----extra test for bad movement
        assertTrue(g.movePieceTo([2,1],[2,2]) = false);

);


             private testGameEndPlayer1: () ==> ()
         testGameEndPlayer1() ==
         (

         dcl g: Game := new Game();

         assertTrue(isofclass(Leopard, g.getBoard().getTile([3,3]).getPiece()));

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([3,3],[4,3]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([4,3],[4,4]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([4,4],[4,5]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([4,5],[4,6]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([4,6],[4,7]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([4,7],[4,8]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver() = false);
         assertTrue(g.movePieceTo([4,8],[4,9]));
         g.changePlayer(); g.changePlayer();

         assertTrue(g.checkGameOver());

);


               private testGameEndPlayer2: () ==> ()
         testGameEndPlayer2() ==
         (

         dcl g: Game := new Game();
```



```
        g.changePlayer();

        assertTrue(isofclass(Wolf, g.getBoard().getTile([3,7]).getPiece()));

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([3,7],[4,7]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([4,7],[4,6]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([4,6],[4,5]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([4,5],[4,4]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([4,4],[4,3]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([4,3],[4,2]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver() = false);
        assertTrue(g.movePieceTo([4,2],[4,1]));
        g.changePlayer(); g.changePlayer();

        assertTrue(g.checkGameOver());

 );

public static main: () ==> ()
           main() ==
           (

           ------------- Movement Tests -------------------------------------------

                  new GameTest().testCatMovement();
                  new GameTest().testDogMovement();
                  new GameTest().testElephantMovement();
                  new GameTest().testTigerMovement();
                  new GameTest().testRatMovement();
                  new GameTest().testLionMovement();
                  new GameTest().testLeopardMovement();
                  new GameTest().testWolfMovement();

                  new GameTest().testMovement();

           ------------- Game Tests -------------------------------------------


                  new GameTest().testGameTurns();
                  new GameTest().testGameEndPlayer1();
                  new GameTest().testGameEndPlayer2();
           );
end GameTest
```

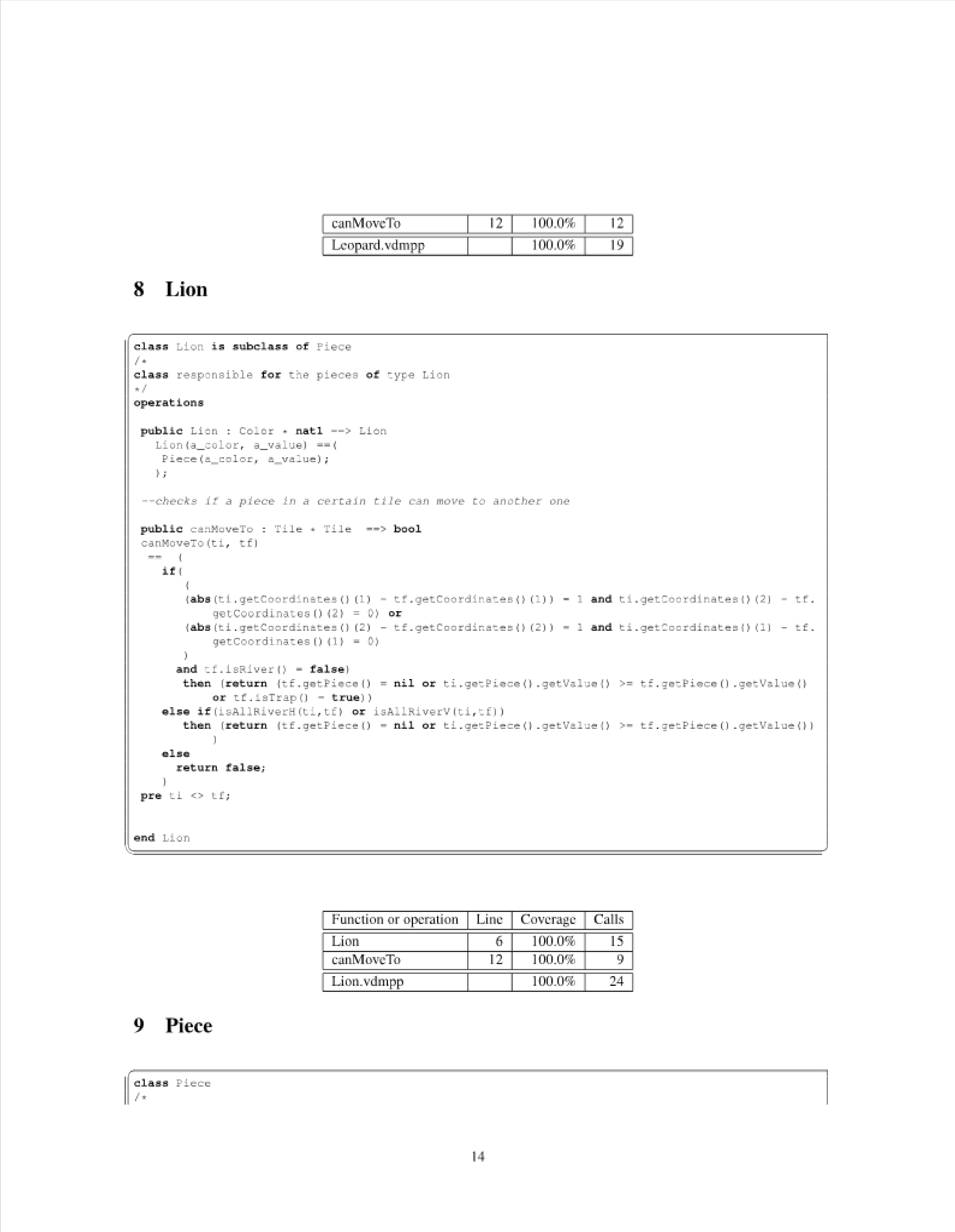


Function or operation	Line	Coverage	Calls
assertTrue	4	100.0%	106
main	395	100.0%	1
testCatMovement	8	100.0%	1
testDogMovement	31	100.0%	1
testElephantMovement	54	100.0%	1
testGameEndPlayer1	313	100.0%	1
testGameEndPlayer2	353	100.0%	1
testGameTurns	292	100.0%	1
testLeopardMovement	77	100.0%	1
testLionMovement	100	100.0%	1
testMovement	236	100.0%	1
testRatMovement	137	100.0%	1
testTigerMovement	170	100.0%	1
testWolfMovement	213	100.0%	1
GameTest.vdmpp		100.0%	119

7 Leopard

```
class Leopard is subclass of Piece
/*
class responsible for the pieces of type Leopard
*/
operations

 public Leopard : Color * nat1 ==> Leopard
   Leopard(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

--checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  ==  if(
        (
        (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
            getCoordinates()(2) = 0) or
        (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
            getCoordinates()(1) = 0)
        )
      and tf.isRiver() = false)
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue()
            or tf.isTrap() = true ))
    else (return false;)
 pre ti <> tf;

end Leopard
```

Function or operation	Line	Coverage	Calls
Leopard	6	100.0%	7



canMoveTo	12	100.0%	12
Leopard.vdmpp		100.0%	19

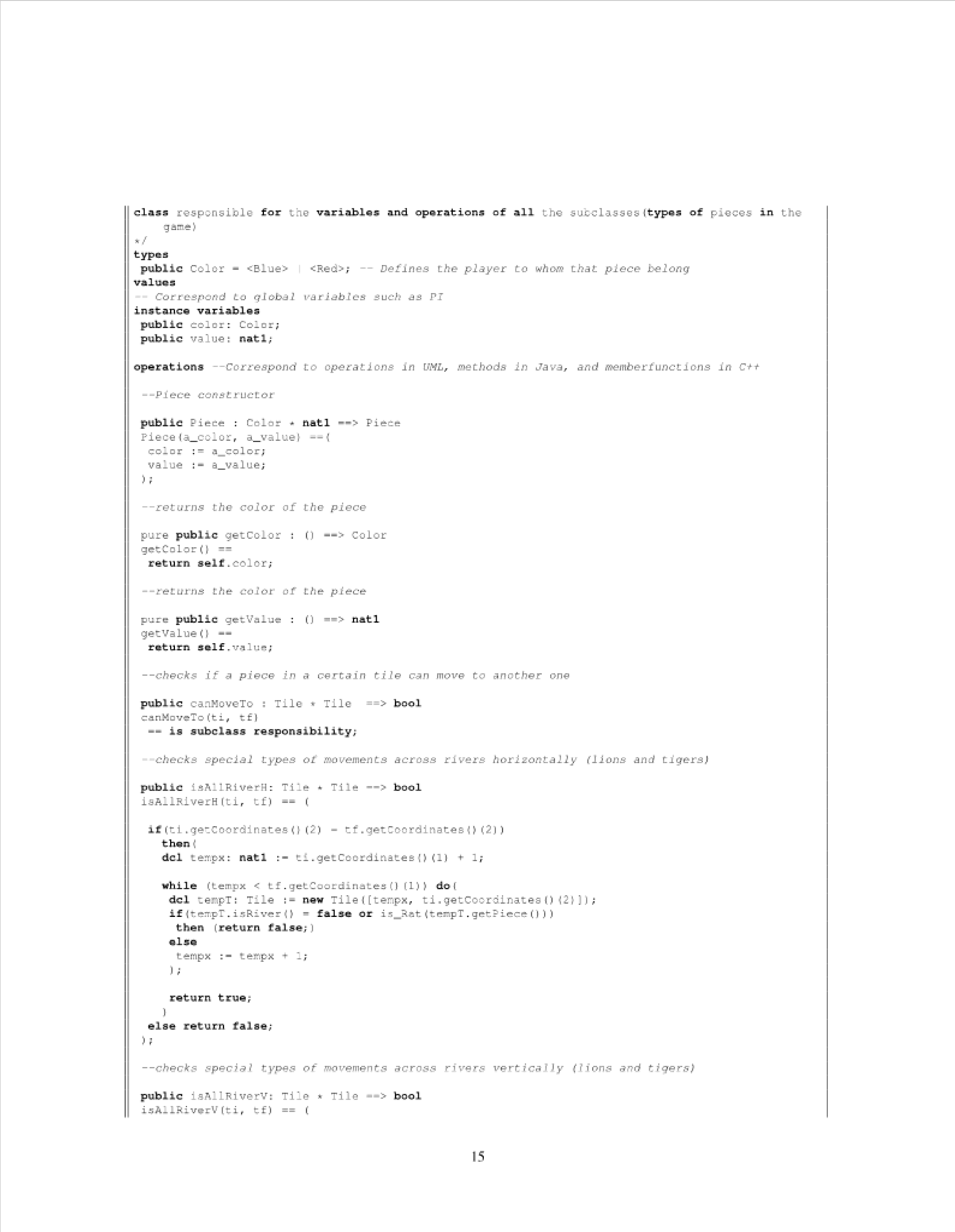
8 Lion

```
class Lion is subclass of Piece
/*
class responsible for the pieces of type Lion
*/
operations

 public Lion : Color * nat1 ==> Lion
   Lion(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

 --checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  ==  (
    if(
       (
       (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
           getCoordinates()(2) = 0) or
       (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
           getCoordinates()(1) = 0)
       )
      and tf.isRiver() = false)
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue()
           or tf.isTrap() = true))
    else if(isAllRiverH(ti,tf) or isAllRiverV(ti,tf))
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue())
           )
    else
      return false;
   )
 pre ti <> tf;


end Lion
```

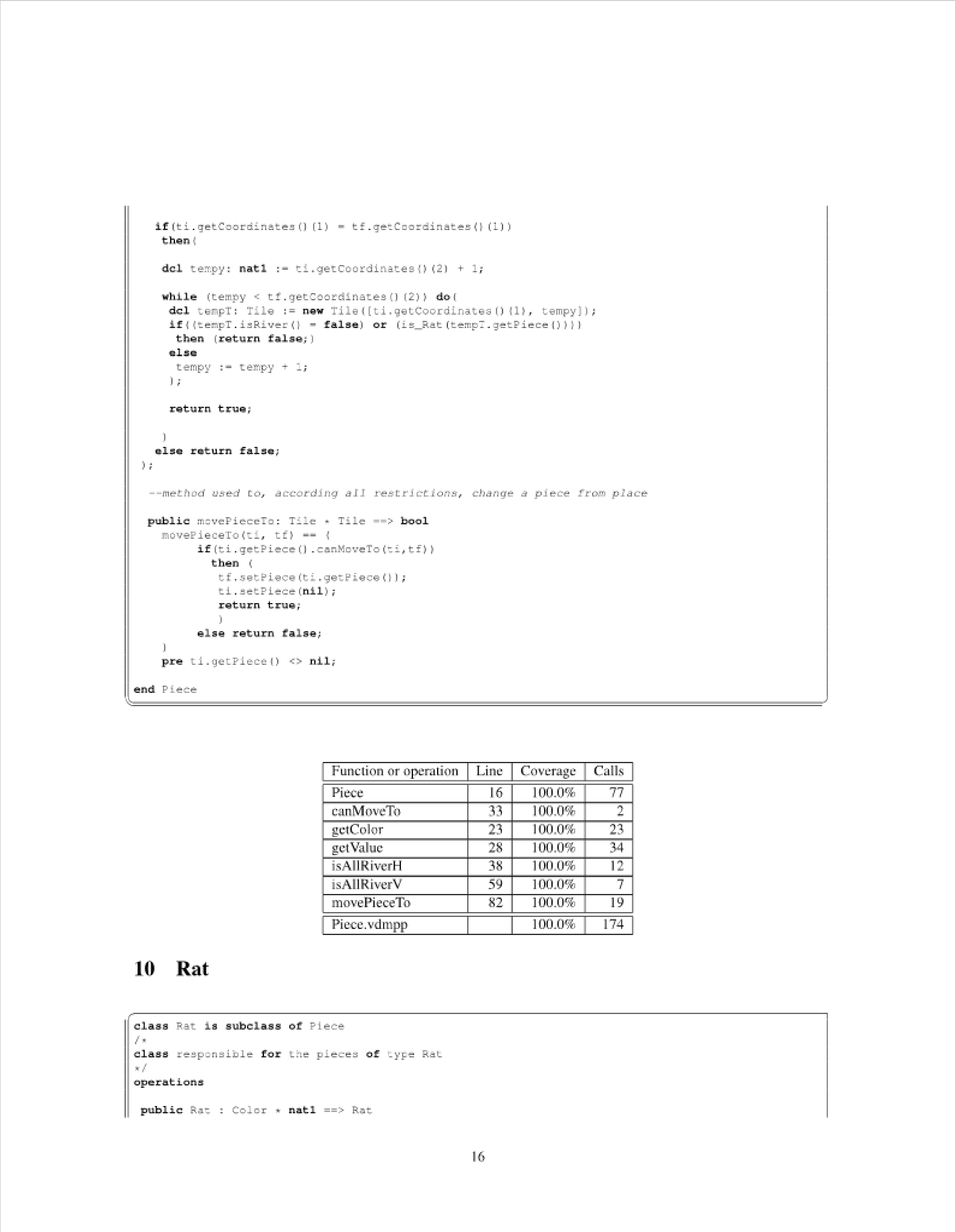
Function or operation	Line	Coverage	Calls
Lion	6	100.0%	15
canMoveTo	12	100.0%	9
Lion.vdmpp		100.0%	24

9 Piece

```
class Piece
/*
```



```
class responsible for the variables and operations of all the subclasses(types of pieces in the
    game)
*/
types
 public Color = <Blue> | <Red>; -- Defines the player to whom that piece belong
values
-- Correspond to global variables such as PI
instance variables
 public color: Color;
 public value: nat1;

operations --Correspond to operations in UML, methods in Java, and memberfunctions in C++

 --Piece constructor

 public Piece : Color * nat1 ==> Piece
 Piece(a_color, a_value) ==(
  color := a_color;
  value := a_value;
 );

 --returns the color of the piece

 pure public getColor : () ==> Color
 getColor() ==
  return self.color;

 --returns the color of the piece

 pure public getValue : () ==> nat1
 getValue() ==
  return self.value;

 --checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  == is subclass responsibility;

 --checks special types of movements across rivers horizontally (lions and tigers)

 public isAllRiverH: Tile * Tile ==> bool
 isAllRiverH(ti, tf) == (

  if(ti.getCoordinates()(2) = tf.getCoordinates()(2))
    then(
    dcl tempx: nat1 := ti.getCoordinates()(1) + 1;

    while (tempx < tf.getCoordinates()(1)) do(
     dcl tempT: Tile := new Tile([tempx, ti.getCoordinates()(2)]);
     if(tempT.isRiver() = false or is_Rat(tempT.getPiece()))
      then (return false;)
     else
      tempx := tempx + 1;
     );

     return true;
    )
  else return false;
 );

 --checks special types of movements across rivers vertically (lions and tigers)

 public isAllRiverV: Tile * Tile ==> bool
 isAllRiverV(ti, tf) == (
```

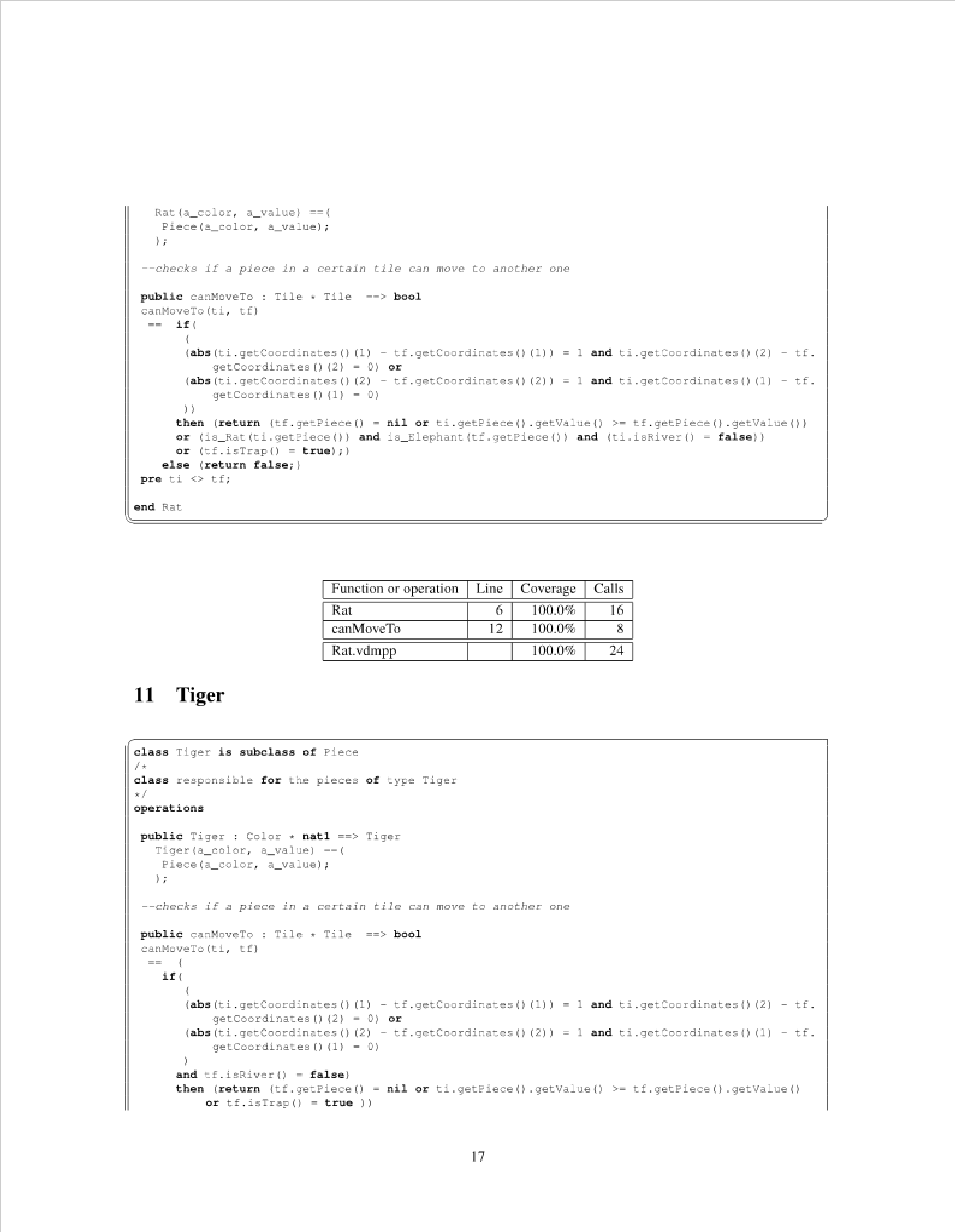


```
    if(ti.getCoordinates()(1) = tf.getCoordinates()(1))
     then(

     dcl tempy: nat1 := ti.getCoordinates()(2) + 1;

     while (tempy < tf.getCoordinates()(2)) do(
      dcl tempT: Tile := new Tile([ti.getCoordinates()(1), tempy]);
      if((tempT.isRiver() = false) or (is_Rat(tempT.getPiece())))
       then (return false;)
      else
       tempy := tempy + 1;
      );

      return true;

     )
    else return false;
  );

  --method used to, according all restrictions, change a piece from place

  public movePieceTo: Tile * Tile ==> bool
    movePieceTo(ti, tf) == (
         if(ti.getPiece().canMoveTo(ti,tf))
           then (
            tf.setPiece(ti.getPiece());
            ti.setPiece(nil);
            return true;
            )
         else return false;
    )
    pre ti.getPiece() <> nil;

end Piece
```

Function or operation	Line	Coverage	Calls
Piece	16	100.0%	77
canMoveTo	33	100.0%	2
getColor	23	100.0%	23
getValue	28	100.0%	34
isAllRiverH	38	100.0%	12
isAllRiverV	59	100.0%	7
movePieceTo	82	100.0%	19
Piece.vdmpp		100.0%	174

10 Rat

```
class Rat is subclass of Piece
/*
class responsible for the pieces of type Rat
*/
operations

 public Rat : Color * nat1 ==> Rat
```

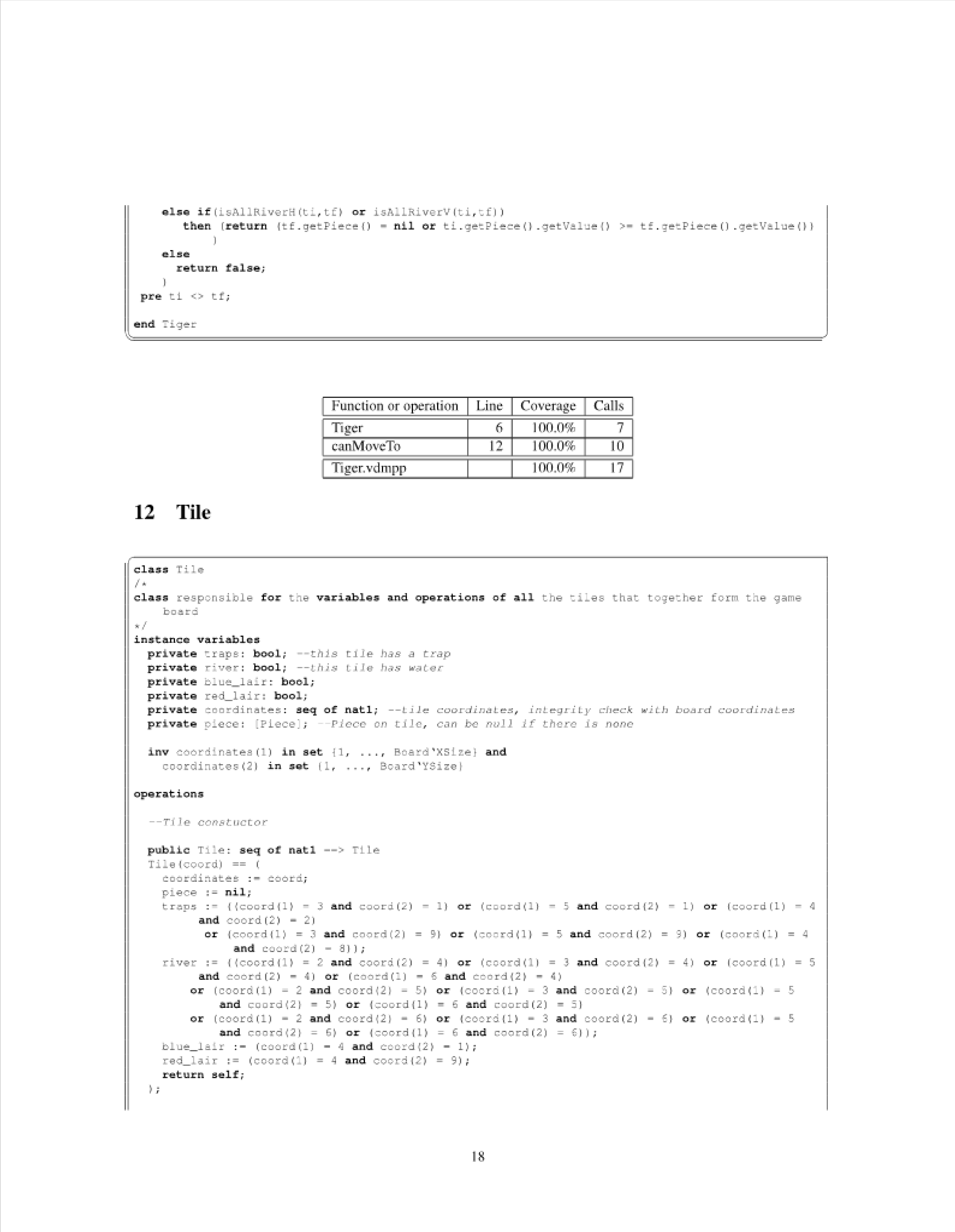


```
   Rat(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

 --checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  ==  if(
        (
        (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
            getCoordinates()(2) = 0) or
        (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
            getCoordinates()(1) = 0)
        ))
      then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue())
      or (is_Rat(ti.getPiece()) and is_Elephant(tf.getPiece()) and (ti.isRiver() = false))
      or (tf.isTrap() = true);)
    else (return false;)
 pre ti <> tf;

end Rat
```

Function or operation	Line	Coverage	Calls
Rat	6	100.0%	16
canMoveTo	12	100.0%	8
Rat.vdmpp		100.0%	24

11 Tiger

```
class Tiger is subclass of Piece
/*
class responsible for the pieces of type Tiger
*/
operations

 public Tiger : Color * nat1 ==> Tiger
   Tiger(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

 --checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  ==  (
    if(
        (
        (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
            getCoordinates()(2) = 0) or
        (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
            getCoordinates()(1) = 0)
        )
      and tf.isRiver() = false)
      then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue()
          or tf.isTrap() = true ))
```

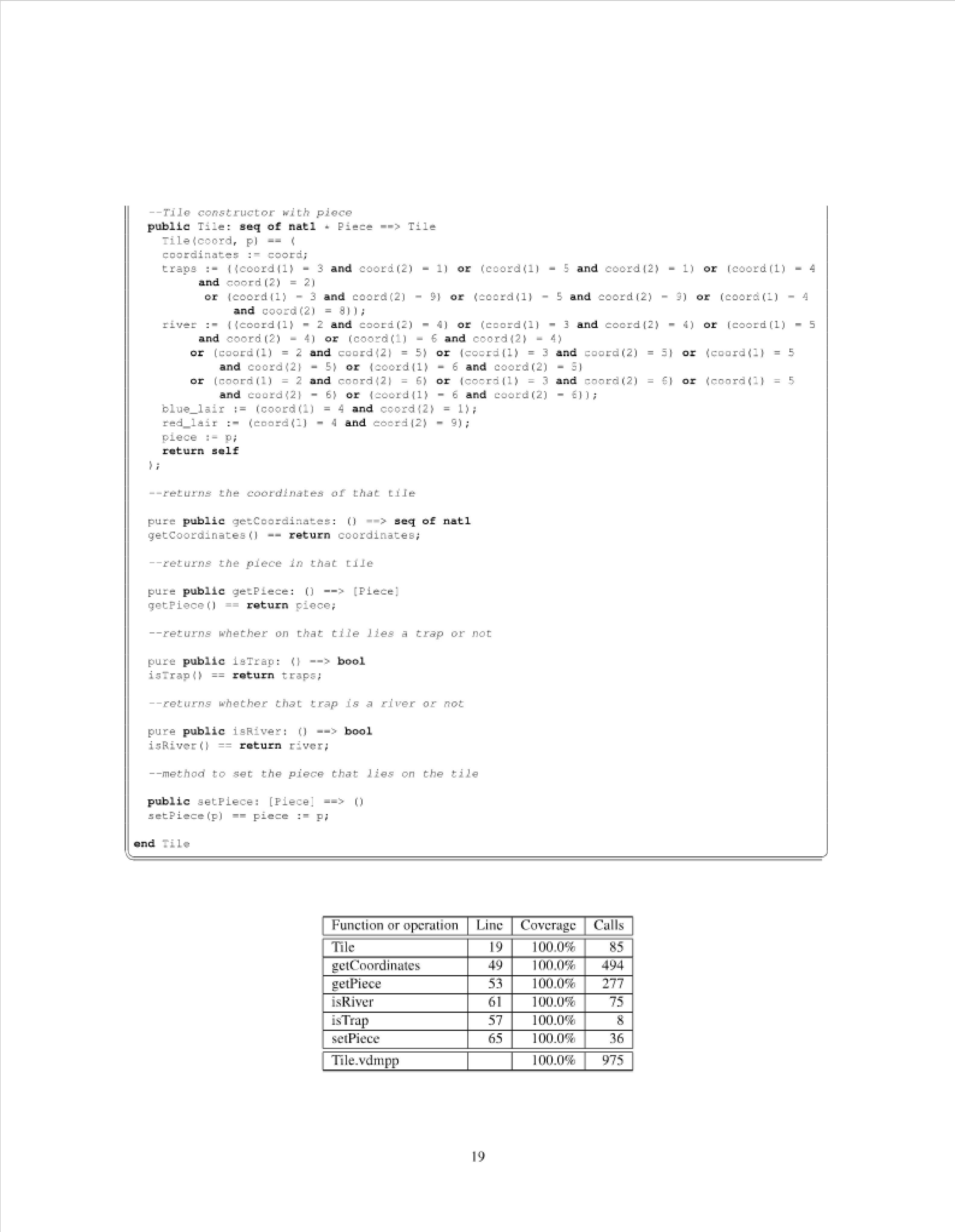


```
    else if(isAllRiverH(ti,tf) or isAllRiverV(ti,tf))
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue())
           )
    else
      return false;
    )
 pre ti <> tf;

end Tiger
```

Function or operation	Line	Coverage	Calls
Tiger	6	100.0%	7
canMoveTo	12	100.0%	10
Tiger.vdmpp		100.0%	17

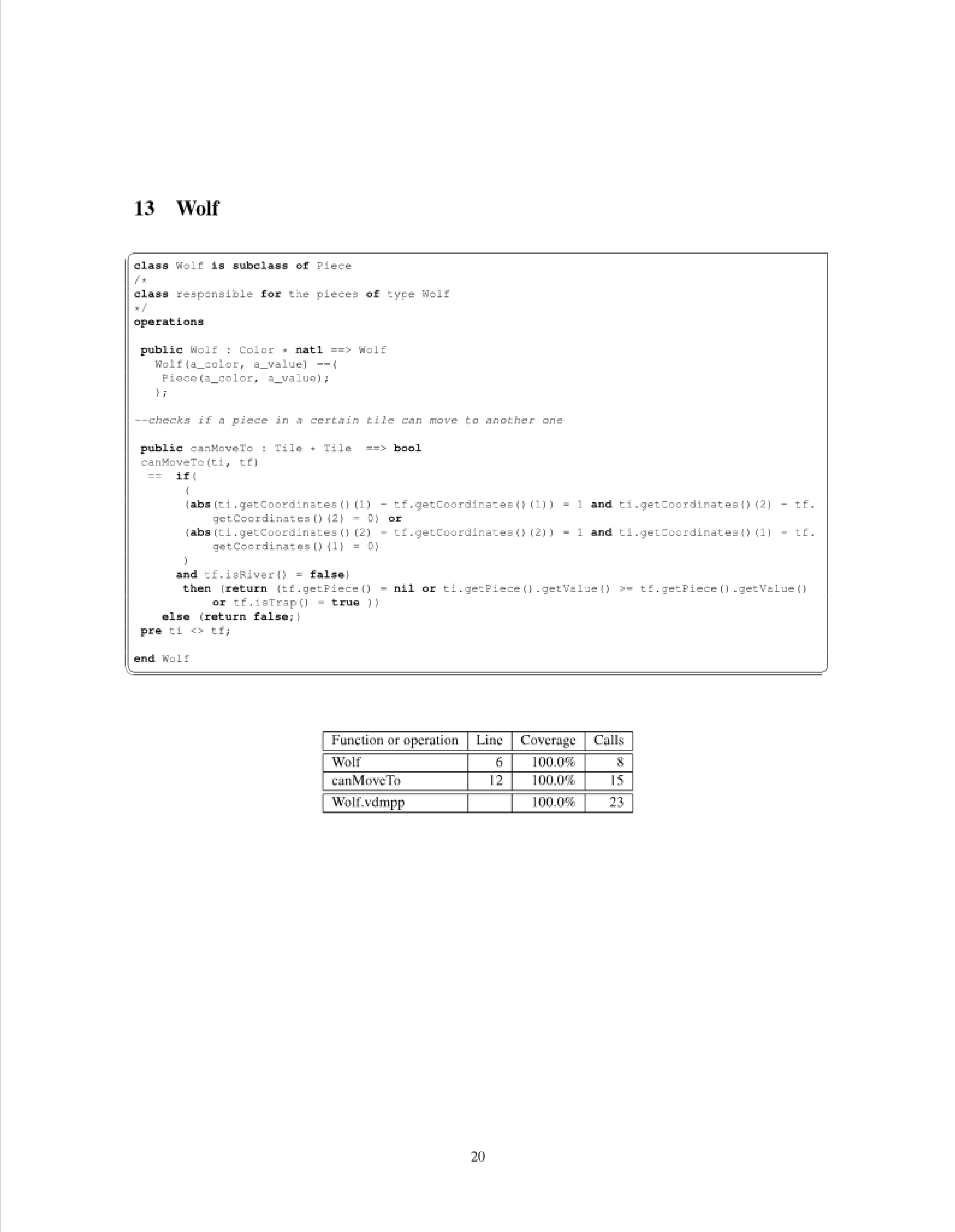
12 Tile

```
class Tile
/*
class responsible for the variables and operations of all the tiles that together form the game
    board
*/
instance variables
  private traps: bool; --this tile has a trap
  private river: bool; --this tile has water
  private blue_lair: bool;
  private red_lair: bool;
  private coordinates: seq of nat1; --tile coordinates, integrity check with board coordinates
  private piece: [Piece]; --Piece on tile, can be null if there is none

  inv coordinates(1) in set {1, ..., Board`XSize} and
    coordinates(2) in set {1, ..., Board`YSize}

operations

  --Tile constuctor

  public Tile: seq of nat1 ==> Tile
  Tile(coord) == (
    coordinates := coord;
    piece := nil;
    traps := ((coord(1) = 3 and coord(2) = 1) or (coord(1) = 5 and coord(2) = 1) or (coord(1) = 4
         and coord(2) = 2)
          or (coord(1) = 3 and coord(2) = 9) or (coord(1) = 5 and coord(2) = 9) or (coord(1) = 4
              and coord(2) = 8));
    river := ((coord(1) = 2 and coord(2) = 4) or (coord(1) = 3 and coord(2) = 4) or (coord(1) = 5
         and coord(2) = 4) or (coord(1) = 6 and coord(2) = 4)
        or (coord(1) = 2 and coord(2) = 5) or (coord(1) = 3 and coord(2) = 5) or (coord(1) = 5
            and coord(2) = 5) or (coord(1) = 6 and coord(2) = 5)
        or (coord(1) = 2 and coord(2) = 6) or (coord(1) = 3 and coord(2) = 6) or (coord(1) = 5
            and coord(2) = 6) or (coord(1) = 6 and coord(2) = 6));
    blue_lair := (coord(1) = 4 and coord(2) = 1);
    red_lair := (coord(1) = 4 and coord(2) = 9);
    return self;
  );
```



```
  --Tile constructor with piece
  public Tile: seq of nat1 * Piece ==> Tile
    Tile(coord, p) == (
    coordinates := coord;
    traps := ((coord(1) = 3 and coord(2) = 1) or (coord(1) = 5 and coord(2) = 1) or (coord(1) = 4
         and coord(2) = 2)
          or (coord(1) = 3 and coord(2) = 9) or (coord(1) = 5 and coord(2) = 9) or (coord(1) = 4
              and coord(2) = 8));
    river := ((coord(1) = 2 and coord(2) = 4) or (coord(1) = 3 and coord(2) = 4) or (coord(1) = 5
         and coord(2) = 4) or (coord(1) = 6 and coord(2) = 4)
        or (coord(1) = 2 and coord(2) = 5) or (coord(1) = 3 and coord(2) = 5) or (coord(1) = 5
            and coord(2) = 5) or (coord(1) = 6 and coord(2) = 5)
        or (coord(1) = 2 and coord(2) = 6) or (coord(1) = 3 and coord(2) = 6) or (coord(1) = 5
            and coord(2) = 6) or (coord(1) = 6 and coord(2) = 6));
    blue_lair := (coord(1) = 4 and coord(2) = 1);
    red_lair := (coord(1) = 4 and coord(2) = 9);
    piece := p;
    return self
  );

  --returns the coordinates of that tile

  pure public getCoordinates: () ==> seq of nat1
  getCoordinates() == return coordinates;

  --returns the piece in that tile

  pure public getPiece: () ==> [Piece]
  getPiece() == return piece;

  --returns whether on that tile lies a trap or not

  pure public isTrap: () ==> bool
  isTrap() == return traps;

  --returns whether that trap is a river or not

  pure public isRiver: () ==> bool
  isRiver() == return river;

  --method to set the piece that lies on the tile

  public setPiece: [Piece] ==> ()
  setPiece(p) == piece := p;

end Tile
```

Function or operation	Line	Coverage	Calls
Tile	19	100.0%	85
getCoordinates	49	100.0%	494
getPiece	53	100.0%	277
isRiver	61	100.0%	75
isTrap	57	100.0%	8
setPiece	65	100.0%	36
Tile.vdmpp		100.0%	975



13 Wolf

```
class Wolf is subclass of Piece
/*
class responsible for the pieces of type Wolf
*/
operations

 public Wolf : Color * nat1 ==> Wolf
   Wolf(a_color, a_value) ==(
    Piece(a_color, a_value);
   );

--checks if a piece in a certain tile can move to another one

 public canMoveTo : Tile * Tile  ==> bool
 canMoveTo(ti, tf)
  ==  if(
       (
       (abs(ti.getCoordinates()(1) - tf.getCoordinates()(1)) = 1 and ti.getCoordinates()(2) - tf.
           getCoordinates()(2) = 0) or
       (abs(ti.getCoordinates()(2) - tf.getCoordinates()(2)) = 1 and ti.getCoordinates()(1) - tf.
           getCoordinates()(1) = 0)
       )
      and tf.isRiver() = false)
       then (return (tf.getPiece() = nil or ti.getPiece().getValue() >= tf.getPiece().getValue()
           or tf.isTrap() = true ))
    else (return false;)
 pre ti <> tf;

end Wolf
```

Function or operation	Line	Coverage	Calls
Wolf	6	100.0%	8
canMoveTo	12	100.0%	15
Wolf.vdmpp		100.0%	23