tetris.vdm

January 8, 2017

Contents

1 Board

```
class Board

 -- Represents the game board, with its matrix.


 types

  -- Position in the board.
  public Position = seq of int
  -- A position is a sequence with exactly to integers.
  inv position == len position = 2;
  -- The board matrix is represented by a map.
  public Matrix = map Position to nat;
```

```
instance variables

 -- Total number of rows in the board, including two lines that
 -- are not displayed for the user to see.
 public static maxRow  : nat1   := 22;
 -- Total number of rows that are visible during gameplay.
 public static maxVisibleRow : nat1   := 20;
 -- Total number of columns in the board.
 public static maxColumn  : nat1   := 10;

 -- Aids for printing the board matrix in the console.
 private print_startTag  : Game`String := "   ";
 private print_endTag  : Game`String := "   \n";
 private print_bottomLine : Game`String := "                                   ";

 -- The matrix attributes to each position in the board
 -- a natural number: 0 if the position is empty; the id of the
 -- tetramino if it is occupied.
 private matrix     : Matrix  := {|->};


operations


 public Board : () ==> Board
 Board() == (
  initBoard();
 );

 -- Initiates the board matrix with all positions empty,
 -- that is, with zeros.

 private initBoard : () ==> ()
 initBoard() == (
  for i = 1 to maxRow do
   for j = 1 to maxColumn do
    matrix := matrix ++ {[i,j] |-> 0}
 )
 post matrix([1,1]) = 0 and matrix([maxRow, maxColumn]) = 0;

 -- Returns a string containing the board matrix in its
 -- printing form. If the boolean printNow is true, the matrix is
 -- immediately printed. If the boolean blackConsole is true, the
 -- matrix is printed without using colors. If the boolean testPrint
 -- is true, the invisible lines are printed for test purposes.

 public getBoardPrint : bool * bool * bool ==> Game`String
 getBoardPrint(printNow, blackConsole, testPrint) == (
  dcl print_board : Game`String := "\n";
  dcl start_row : nat1 := 3;
  if testPrint then start_row := 1;
  for i = start_row to maxRow do(
   print_board := print_board ^ print_startTag;
     for j=1 to maxColumn do
      print_board := print_board
       ^ getCellPrint(matrix([i,j]), i, blackConsole);
     print_board := print_board ^  print_endTag;
    );
    if printNow then
     IO`println(print_board ^ print_bottomLine);
   return print_board ^ print_bottomLine ;
 );

 -- Returns the string corresponding to a board cell print.
 -- If blackConsole is true, there are only three strings
```

```
  -- possible, depending on whether the position is empty or filled
  -- and diferentiating the invisible lines.
  -- If blackConsole is false, each tetramino will have a different
  -- print, according to its color.

  private getCellPrint: nat * nat * bool ==> Game`String
  getCellPrint(id, row, blackConsole) == (
   if blackConsole then
    cases id:
     0 -> if row < 3 then return " " else return "   ",
     others -> return "   "
    end
   else
    cases id:
     0 -> if row < 3 then return " " else return "",
     1 -> return "\u001B[38;5;51m" ^ "" ^ "\u001B[0m",
     2 -> return "\u001B[38;5;21m" ^ "" ^ "\u001B[0m",
     3 -> return "\u001B[38;5;208m" ^ "" ^ "\u001B[0m",
     4 -> return "\u001B[38;5;226m" ^ "" ^ "\u001B[0m",
     5 -> return "\u001B[38;5;34m" ^ "" ^ "\u001B[0m",
     6 -> return "\u001B[38;5;165m" ^ "" ^ "\u001B[0m",
     others -> return "\u001B[38;5;196m" ^ "" ^ "\u001B[0m"
    end
  );

  -- Checks a board row to determine if it is completely filled
  -- with tetramino cells. In affirmative case, the row is removed
  -- from the board and the rows above it are shifted down.

  private checkRow : int ==> bool
  checkRow(row) == (
   for column = 1 to maxColumn do
    if (matrix([row, column]) = 0) then return false;
   for i = row - 1 to 1 by -1 do
    for j = 1 to maxColumn do
     matrix([i + 1, j]) := matrix([i, j]);
   return true
  )
  pre row >= 1 and row <= maxRow;

  -- Checks all the rows of the board and returns the number of rows
  -- that where removed.

  public checkRows : () ==> nat
  checkRows() == (
   dcl result : nat := 0;
   for row = 1 to maxRow do
    if checkRow(row) then result := result + 1;
   return result
  )
  post RESULT <= maxRow;


  public setMatrixPosition : Position * nat ==> ()
  setMatrixPosition(position, value) ==
   matrix(position) := value
  pre Tetramino`checkPosition(position, 1, maxRow, 1, maxColumn);


  public getMatrixPosition : Position ==> nat
  getMatrixPosition(position) ==
   return matrix(position)
  pre Tetramino`checkPosition(position, 1, maxRow, 1, maxColumn);

end Board
```

Function or operation	Line	Coverage	Calls
Board	39	100.0%	1
checkRow	105	100.0%	1364
checkRows	118	100.0%	62
getBoardPrint	59	100.0%	654
getCellPrint	82	21.3%	143880
getMatrixPosition	132	100.0%	6349
initBoard	46	100.0%	1
setMatrixPosition	127	100.0%	12924
Board.vdmpp		74.8%	165235

2 Game

```
class Game

 -- Represents the whole game, and provides all the necessary
 -- operations for playing it.


 types

  public String = seq of char;


 instance variables

  -- The game board.
  private board  : Board;
  -- The current game tetramino.
  private tetramino : Tetramino;
  -- Indicates if the game is finished.
  private gameOver : bool   := false;
  -- The game score.
  private score  : nat   := 0;
  -- The number of completed lines in the game.
  private lines  : nat   := 0;
  -- The level of the game.
  private level  : nat1   := 1;

  -- The scores that lines make, according to the number of lines
  -- made at once (1, 2, 3 or 4).
  private static lineScores : seq of nat
   := [100, 300, 400, 800];


 operations


  public Game : () ==> Game
  Game() ==
   board := new Board();
```

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


public setGameOver : () ==> ()
setGameOver() ==
 gameOver := true;


public getGameOver : () ==> bool
getGameOver() ==
 return gameOver;

  -- Generates a new tetramino of the specified type.

public newTetramino : nat1 ==> ()
newTetramino(id) == (
 cases id:
  1 -> tetramino := new TetraminoI(self),
  2 -> tetramino := new TetraminoJ(self),
  3 -> tetramino := new TetraminoL(self),
  4 -> tetramino := new TetraminoO(self),
  5 -> tetramino := new TetraminoS(self),
  6 -> tetramino := new TetraminoT(self),
  7 -> tetramino := new TetraminoZ(self)
 end;
)
pre id >= 1 and id <= 7;

  -- Generates a new random tetramino.

public newRandomTetramino: () ==> ()
newRandomTetramino() ==
 newTetramino(MATH`rand(7) + 1);

  -- Tries to move the current tetramino down.

public down : () ==> bool
down() ==
 return tetramino.moveDown(board);

  -- Tries to move the current tetramino left.

public left : () ==> bool
left() ==
 return tetramino.moveLeft(board);

  -- Tries to move the current tetramino right.

public right : () ==> bool
right() ==
 return tetramino.moveRight(board);

  -- Tries to rotate the current tetramino.

public rotate : () ==> bool
rotate() ==
 return tetramino.rotate(board);

  -- Drops the current tetramino to the bottom of the board.
  -- Also updates the score, using the current level and the
  -- total distance of the drop.

public drop : () ==> nat
```

```
  drop() == (
   dcl dropDistance: nat := tetramino.drop(board);
   score := score + dropDistance * level;
   return dropDistance;
  );

  -- Checks all game lines for completion and updates the game
  -- score according to the number of completed lines and the
  -- current game level.

  public checkLines : () ==> nat
  checkLines() == (
   dcl newLines: nat := board.checkRows();
   lines := lines + newLines;
   if newLines > 0 then score := score + lineScores(newLines) * level;
   level := 1 + (lines div 10);
   return newLines
  );


  public getScore : () ==> nat
  getScore() == return score;


  public getLines : () ==> nat
  getLines() == return lines;


  public getLevel : () ==> nat1
  getLevel() == return level;

  -- Returns a printing string of the board.
  -- If blackConsole is true, there are only three strings
  -- possible, depending on whether the position is empty or filled
  -- and diferentiating the invisible lines.
  -- If blackConsole is false, each tetramino will have a different
  -- print, according to its color.

  public printBoard : bool * bool * bool ==> String
  printBoard(printNow, blackConsole, testPrint) ==
   return board.getBoardPrint(printNow, blackConsole, testPrint);

end Game
```

Function or operation	Line	Coverage	Calls
Game	35	100.0%	1
checkLines	104	100.0%	62
down	72	100.0%	96
drop	94	100.0%	62
getBoard	39	100.0%	309
getGameOver	47	100.0%	78
getLevel	119	100.0%	2
getLines	116	100.0%	2
getScore	113	100.0%	12
left	77	100.0%	248
newRandomTetramino	67	100.0%	14
newTetramino	52	100.0%	62

printBoard	128	100.0%	654
right	82	100.0%	190
rotate	87	100.0%	70
setGameOver	43	100.0%	4
Game.vdmpp		100.0%	1866

3 TestCaseExtra

```
class TestCaseExtra

 -- Class obtained from the Vending Machine example of MFES


 operations
 -- Simulates assertion checking by reducing it to pre-condition checking.
 -- If 'arg' does not hold, a pre-condition violation will be signaled.

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

 -- Simulates assertion checking by reducing it to post-condition checking.
 -- If values are not equal, prints a message in the console and generates
 -- a post-conditions violation.

 protected assertEqual: ? * ? ==> ()
 assertEqual(expected, actual) ==
  if expected <> actual then (
     IO`print("Actual value (");
     IO`print(actual);
     IO`print(") different from expected (");
     IO`print(expected);
     IO`println(")\n")
  )
 post expected = actual

end TestCaseExtra
```

Function or operation	Line	Coverage	Calls
assertEqual	17	38.8%	121
assertTrue	9	100.0%	185
TestCaseExtra.vdmpp		45.0%	306

4 TestTetris

```
class TestTetris is subclass of TestCaseExtra

 instance variables

  private printBoard : Game`String := "";
```

```
operations

 -- Prints the initial board matrix and checks it is as should be for tests.

 private initalMatrix_test: Game * bool * bool * bool ==> ()
 initalMatrix_test(game, printNow, blackConsole, testPrint) == (
  printBoard := game.printBoard(printNow, blackConsole, testPrint);
  assertEqual(printBoard,
   "\ n                " ^
   "\ n                " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\n                               ")
 );

 -- Adds a tetramino of type I to the board and checks it is in the right position.

 private addTetramino1_test: Game * bool * bool * bool ==> ()
 addTetramino1_test(game, printNow, blackConsole, testPrint) == (
  game.newTetramino(1);
  printBoard := game.printBoard(printNow, blackConsole, testPrint);
  assertEqual(printBoard,
   "\ n                " ^
   "\ n                        " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\ n                                  " ^
   "\n                               ")
```

```
);

-- Adds a tetramino of type J to the board and checks it is in the right position.

private addTetramino2_test: Game * bool * bool * bool ==> ()
addTetramino2_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(2);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                  " ^
  "\ n                      " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\n                               ")
);

-- Adds a tetramino of type L to the board and checks it is in the right position.

private addTetramino3_test: Game * bool * bool * bool ==> ()
addTetramino3_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(3);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                  " ^
  "\ n                      " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\n                               ")
);
```

```
-- Adds a tetramino of type O to the board and checks it is in the right position.

private addTetramino4_test: Game * bool * bool * bool ==> ()
addTetramino4_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(4);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                    " ^
  "\ n                    " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\n                               ")
);

-- Adds a tetramino of type S to the board and checks it is in the right position.

private addTetramino5_test: Game * bool * bool * bool ==> ()
addTetramino5_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(5);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                    " ^
  "\ n                    " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\n                               ")
);
```

```
-- Adds a tetramino of type T to the board and checks it is in the right position.

private addTetramino6_test: Game * bool * bool * bool ==> ()
addTetramino6_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(6);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                    " ^
  "\ n                        " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\n                                 ")
);

-- Adds a tetramino of type Z to the board and checks it is in the right position.

private addTetramino7_test: Game * bool * bool * bool ==> ()
addTetramino7_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(7);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                     " ^
  "\ n                     " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\ n                                    " ^
  "\n                                 ")
);

-- Adds another tetramino of type S to the board and checks it is in the right position.
```

```
private addTetramino8_test: Game * bool * bool * bool ==> ()
addTetramino8_test(game, printNow, blackConsole, testPrint) == (
 game.newTetramino(5);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(printBoard,
  "\ n                    " ^
  "\ n                    " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\ n                                  " ^
  "\n                               ")
);

-- Moves the first tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino1_test: Game * bool * bool * bool ==> ()
moveTetramino1_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 18);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 18);
 assertEqual(game.checkLines(), 0);
);

-- Moves the second tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino2_test: Game * bool * bool * bool ==> ()
moveTetramino2_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
```

```
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 18);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 36);
 assertEqual(game.checkLines(), 0);
);

-- Moves the third tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino3_test: Game * bool * bool * bool ==> ()
moveTetramino3_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 18);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 54);
 assertEqual(game.checkLines(), 1);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 154);
);

-- Moves the fourth tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino4_test: Game * bool * bool * bool ==> ()
moveTetramino4_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
```

```
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 18);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 172);
 assertEqual(game.checkLines(), 0);
);

-- Moves the fifth tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino5_test: Game * bool * bool * bool ==> ()
moveTetramino5_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 18);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 190);
 assertEqual(game.checkLines(), 0);
);

-- Moves the sixed tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino6_test: Game * bool * bool * bool ==> ()
moveTetramino6_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.right());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 17);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
```

```
 assertEqual(game.getScore(), 207);
 assertEqual(game.checkLines(), 0);
);

-- Moves the seventh tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino7_test: Game * bool * bool * bool ==> ()
moveTetramino7_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 18);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 225);
 assertEqual(game.checkLines(), 0);
);

-- Moves the eighted tetramino several times and in several ways and checks the drop
-- amount as well as the score and the lines.

private moveTetramino8_test: Game * bool * bool * bool ==> ()
moveTetramino8_test(game, printNow, blackConsole, testPrint) == (
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.down());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.rotate());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertTrue(game.left());
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.drop(), 17);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 242);
 assertEqual(game.checkLines(), 2);
 printBoard := game.printBoard(printNow, blackConsole, testPrint);
 assertEqual(game.getScore(), 542);
);

-- Plays the game for a while, using only I tetraminoes, to test the number of lines
-- completed, as well as the score and level increments.

private play_test: Game * bool * bool * bool ==> ()
play_test(game, printNow, blackConsole, testPrint) == (
 dcl column: nat := 0;
 for i = 1 to 40 do (
  game.newTetramino(1);
  assertTrue(game.down());
```

```
   printBoard := game.printBoard(printNow, blackConsole, testPrint);
   assertTrue(game.down());
   printBoard := game.printBoard(printNow, blackConsole, testPrint);
   assertTrue(game.rotate());
   printBoard := game.printBoard(printNow, blackConsole, testPrint);
   while game.left() do
    printBoard := game.printBoard(printNow, blackConsole, testPrint);
   for j = 1 to column do
    if game.right() then
     printBoard := game.printBoard(printNow, blackConsole, testPrint);
   if column = 2 or column = 6 then assertEqual(game.drop(), 15)
   else assertEqual(game.drop(), 16);
   printBoard := game.printBoard(printNow, blackConsole, testPrint);
   if column = 9 then (
    assertEqual(game.checkLines(), 4);
    printBoard := game.printBoard(true, true, true)
   )
   else assertEqual(game.checkLines(), 0);
   column := (column + 1) mod 10;
   if i = 20 then (
    assertEqual(game.getLines(), 11);
    assertEqual(game.getLevel(), 2);
    assertEqual(game.getScore(), 542 + 16 * 16 + 15 * 4 + 800 * 2);
   )
  );
  assertEqual(game.getLines(), 19);
  assertEqual(game.getLevel(), 2);
  assertEqual(game.getScore(), 2458 + 16 * 16 * 2 + 15 * 4 * 2 + 800 * 2 * 2);
 );

 -- Randomly generates tetraminoes and drops them, in order to test that the game
 -- ends correctly.

 private gameOver_test: Game * bool * bool * bool ==> ()
 gameOver_test(game, printNow, blackConsole, testPrint) == (

  while not game.getGameOver() do (
   game.newRandomTetramino();
   printBoard := game.printBoard(printNow, blackConsole, testPrint);
   if game.drop() > 0
   then printBoard := game.printBoard(printNow, blackConsole, testPrint);
   if game.checkLines() > 0
   then printBoard := game.printBoard(printNow, blackConsole, testPrint);
  );
  assertTrue(game.getGameOver());
 );

 -- Runs all the tests, printing the board for further visual
 -- confirmation of the results.

 public static main: bool * bool * bool ==> ()
 main(printNow, blackConsole, testPrint) == (
  dcl game: Game := new Game();
  IO`print("\n##### TESTS #####\n");
  new TestTetris().initalMatrix_test(game, printNow, blackConsole, testPrint);
  new TestTetris().addTetramino1_test(game, printNow, blackConsole, testPrint);
  new TestTetris().moveTetramino1_test(game, printNow, blackConsole, testPrint);
  new TestTetris().addTetramino2_test(game, printNow, blackConsole, testPrint);
  new TestTetris().moveTetramino2_test(game, printNow, blackConsole, testPrint);
  new TestTetris().addTetramino3_test(game, printNow, blackConsole, testPrint);
  new TestTetris().moveTetramino3_test(game, printNow, blackConsole, testPrint);
  new TestTetris().addTetramino4_test(game, printNow, blackConsole, testPrint);
  new TestTetris().moveTetramino4_test(game, printNow, blackConsole, testPrint);
  new TestTetris().addTetramino5_test(game, printNow, blackConsole, testPrint);
  new TestTetris().moveTetramino5_test(game, printNow, blackConsole, testPrint);
```

```
    new TestTetris().addTetramino6_test(game, printNow, blackConsole, testPrint);
    new TestTetris().moveTetramino6_test(game, printNow, blackConsole, testPrint);
    new TestTetris().addTetramino7_test(game, printNow, blackConsole, testPrint);
    new TestTetris().moveTetramino7_test(game, printNow, blackConsole, testPrint);
    new TestTetris().addTetramino8_test(game, printNow, blackConsole, testPrint);
    new TestTetris().moveTetramino8_test(game, printNow, blackConsole, testPrint);
    new TestTetris().play_test(game, printNow, blackConsole, testPrint);
    new TestTetris().gameOver_test(game, printNow, blackConsole, testPrint);
   );

end TestTetris
```

Function or operation	Line	Coverage	Calls
addTetramino1_test	41	100.0%	1
addTetramino2_test	72	100.0%	1
addTetramino3_test	103	100.0%	1
addTetramino4_test	134	100.0%	1
addTetramino5_test	165	100.0%	1
addTetramino6_test	196	100.0%	1
addTetramino7_test	227	100.0%	1
addTetramino8_test	258	100.0%	1
gameOver_test	540	84.2%	1
initalMatrix_test	11	100.0%	1
main	556	100.0%	1
moveTetramino1_test	290	100.0%	1
moveTetramino2_test	312	100.0%	1
moveTetramino3_test	340	100.0%	1
moveTetramino4_test	372	100.0%	1
moveTetramino5_test	394	100.0%	1
moveTetramino6_test	424	100.0%	1
moveTetramino7_test	452	100.0%	1
moveTetramino8_test	480	100.0%	1
play_test	502	100.0%	1
TestTetris.vdmpp		99.6%	20

5 Tetramino

```
class Tetramino

 -- Defines a tetris piece, with all its relevant attributes.
 -- The position of the tetramino is represented by the position of
 -- its minoes, which are the four cells composing the tetramino.


 types

  public Color = <Cyan> | <Blue> | <Orange>
    | <Yellow> | <Green> | <Purple> | <Red>;
  public Minoes = seq of Board`Position
  -- There are always 4 minoes in each tetramino.
  inv minoes == len minoes = 4
```

```
instance variables

 -- Color of the piece, to aid in visualization
 private color  : Color  := <Cyan>;
 -- Id of the piece, to simplify its representation in the board
 private id   : nat1   := 1;
 -- The current orientation of the tetramino, since it can rotate
 private orientation : nat   := 0;
 -- The positions of each of the individual cells of the tetramino
 private minoes  : Minoes := [[1, 1], [1, 2], [1, 3], [1, 4]];

 -- The id is a natural number between 1 and 7,
 -- because there are 7 different types of tetraminoes
 -- The orientation is a number between 0 and 3,
 -- because there are at most 4 different orientations,
 -- depending on the tetramino type.
 inv id <= 7 and orientation < 4


functions

 -- Checks if a given position is contained within the expected parameters.

 public checkPosition : Board`Position * int * int * int * int -> bool
 checkPosition(position, min1, max1, min2, max2) ==
  position(1) >= min1
  and position(1) <= max1
  and position(2) >= min2
  and position(2) <= max2;

 -- Checks that all the tetramino cells have different positions,
 -- and that those positions are inside the expected parameters,
 -- which will be the board limits

 public checkMinoes: Minoes * int * int * int * int -> bool
 checkMinoes(minoes, min1, max1, min2, max2) ==
  card elems minoes = 4
  and forall mino in set elems minoes &
   checkPosition(mino, min1, max1, min2, max2)


operations


 protected setColor : Color ==> ()
 setColor(c) == color := c;


 protected setId : nat ==> ()
 setId(i) == id := i
 pre i >= 1 and i <= 7;


 public getOrientation : () ==> nat
 getOrientation() == return orientation
 post RESULT < 4;

 -- Given the position of the first of the four minoes, this operation
 -- sets the position of remaining three minoes. It also places the tetramino
 -- on the board if all positions are valid, otherwise, the tetramino remains
 -- in its original position.

 private setMinoes : Board * Board`Position ==> bool
```

```
setMinoes(board, position) == (
 dcl tempMinoes : Minoes := minoes;
 dcl tempPosition : Board`Position := position;
 removeTetramino(board);
 for i = 1 to 4 do (
  if (validPosition(board, tempPosition))
  then tempMinoes(i) := tempPosition
  else (
   addTetramino(board);
   return false
  );
  tempPosition := getNextMino(tempPosition, i);
 );
 minoes := tempMinoes;
 addTetramino(board);
 return true
)
pre checkPosition(position, -1, Board`maxRow + 2, -1, Board`maxColumn + 2)
post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Very similar to the previous operation, but this one is used only when a
-- tetramino is generated. It does not remove the tetramino from its previous
-- position on the board (since it was never placed) and if at least one of the
-- positions of the minoes is invalid, it declares the game over.

protected initialSetMinoes : Game * Board`Position ==> ()
initialSetMinoes(game, position) == (
 dcl tempPosition : Board`Position := position;
 dcl tempMinoes : Minoes := minoes;
 for i = 1 to 4 do (
  if (validPosition(game.getBoard(), tempPosition))
  then (
   tempMinoes(i) := tempPosition;
   tempPosition := getNextMino(tempPosition, i)
  )
  else game.setGameOver()
 );
 if not game.getGameOver() then (
  minoes := tempMinoes;
  addTetramino(game.getBoard())
 )
)
pre checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Given the position of one mino it computes the position of the next mino
-- in the tetramino. The current mino is identified by its index within the
-- tetramino. This is highly dependent on the type of tetramino, so can
-- only be defined in each subclass.

protected getNextMino: Board`Position * nat ==> Board`Position
getNextMino(position, index) ==
 is subclass responsibility;

-- Given the current position of the first mino of a tetramino, determines
-- the position of that mino after the tetramino is rotated.

protected getRotatedMino: Board`Position ==> Board`Position
getRotatedMino(position) ==
 is subclass responsibility;

-- Checks if a given position on the board is valid for occupation, that is,
-- if it is inside the board limits and not currently occupied by some mino

protected validPosition : Board * Board`Position ==> bool
```

```
validPosition(board, position) ==
 return checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  and board.getMatrixPosition(position) = 0;

-- Removes the tetramino from the board.

protected removeTetramino : Board ==> ()
removeTetramino(board) ==
 for mino in minoes do
  board.setMatrixPosition(mino, 0)
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Places the tetramino in the board.

protected addTetramino : Board ==> ()
addTetramino(board) ==
 for mino in minoes do
  board.setMatrixPosition(mino, id)
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Moves the tetramino down in the board.

public moveDown : Board ==> bool
moveDown(board) ==
 return setMinoes(board, [minoes(1)(1) + 1, minoes(1)(2)])
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn)
post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Moves the tetramino left in the board.

public moveLeft : Board ==> bool
moveLeft(board) ==
 return setMinoes(board, [minoes(1)(1), minoes(1)(2) - 1])
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn)
post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Moves the tetramino right in the board.

public moveRight : Board ==> bool
moveRight(board) ==
 return setMinoes(board, [minoes(1)(1), minoes(1)(2) + 1])
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn)
post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Rotates the tetramino in the board.

public rotate : Board ==> bool
rotate(board) == (
 dcl position : Board`Position := getRotatedMino(minoes(1));
 orientation := (orientation + 1) mod 4;
 return setMinoes(board, position)
)
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn)
post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn);

-- Drops the tetramino to the lowest available position in the board.

public drop : Board ==> nat
drop(board) == (
 dcl result : nat := 0;
 while moveDown(board) do
  result := result + 1;
 return result
)
pre checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn)
```

```
  post checkMinoes(minoes, 1, Board`maxRow, 1, Board`maxColumn)
   and RESULT < Board`maxRow;

end Tetramino
```

Function or operation	Line	Coverage	Calls
addTetramino	147	100.0%	1646
checkMinoes	49	100.0%	8172
checkPosition	39	100.0%	145640
drop	185	100.0%	62
getNextMino	122	100.0%	2
getOrientation	65	100.0%	6392
getRotatedMino	128	100.0%	2
initialSetMinoes	98	100.0%	62
moveDown	154	100.0%	1077
moveLeft	161	100.0%	248
moveRight	168	100.0%	190
removeTetramino	140	100.0%	1585
rotate	175	100.0%	70
setColor	58	100.0%	62
setId	61	100.0%	62
setMinoes	73	100.0%	1585
validPosition	134	100.0%	6428
Tetramino.vdmpp		100.0%	173285

6 TetraminoI

```
class TetraminoI is subclass of Tetramino

 -- Tetramino of the type I: xxxx
 --
 --
 -- Minoes:         1234
 --


 operations


  public TetraminoI : Game ==> TetraminoI
  TetraminoI(game) == (
   Tetramino`setColor(<Cyan>);
   Tetramino`setId(1);
   Tetramino`initialSetMinoes(game, [2, 4]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
```

```
    0 -> result(2) := position(2) + 1,
    1 -> result(1) := position(1) + 1,
    2 -> result(2) := position(2) - 1,
    3 -> result(1) := position(1) - 1
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     result(1) := position(1) - 1;
     result(2) := position(2) + 2;
     ),
    1 -> (
     result(1) := position(1) + 2;
     result(2) := position(2) + 1;
     ),
    2 -> (
     result(1) := position(1) + 1;
     result(2) := position(2) - 2;
     ),
    3 -> (
     result(1) := position(1) - 2;
     result(2) := position(2) - 1;
     )
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);


end TetraminoI
```

Function or operation	Line	Coverage	Calls
TetraminoI	12	100.0%	43
getNextMino	20	100.0%	4971
getRotatedMino	34	100.0%	44
TetraminoI.vdmpp		100.0%	5058

7 TetraminoJ

```
class TetraminoJ is subclass of Tetramino

 -- Tetramino of the type J: x
 --       xxx
 --
 -- Minoes:       1
 --        234
```

```
 operations


  public TetraminoJ : (Game) ==> TetraminoJ
  TetraminoJ(game) == (
   Tetramino`setColor(<Blue>);
   Tetramino`setId(2);
   Tetramino`initialSetMinoes(game, [1, 4]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     cases index:
      1 -> result(1) := position(1) + 1,
      others -> result(2) := position(2) + 1
     end
    ),
    1 -> (
     cases index:
      1 -> result(2) := position(2) - 1,
      others -> result(1) := position(1) + 1
     end
    ),
    2 -> (
     cases index:
      1 -> result(1) := position(1) - 1,
      others -> result(2) := position(2) - 1
     end
    ),
    3 -> (
     cases index:
      1 -> result(2) := position(2) + 1,
      others -> result(1) := position(1) - 1
     end
    )
   end;
   return result
  )
  pre index in set {1, ..., 4}
   and Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> result(2) := position(2) + 2,
    1 -> result(1) := position(1) + 2,
    2 -> result(2) := position(2) - 2,
    3 -> result(1) := position(1) - 2
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);

end TetraminoJ
```

Function or operation	Line	Coverage	Calls
TetraminoJ	12	100.0%	4
getNextMino	20	100.0%	258
getRotatedMino	55	100.0%	4
TetraminoJ.vdmpp		100.0%	266

8 TetraminoL

```
class TetraminoL is subclass of Tetramino

 -- Tetramino of the type J: x
 --       xxx
 --
 -- Minoes:         1
 --         432


 operations


  public TetraminoL : (Game) ==> TetraminoL
  TetraminoL(game) == (
   Tetramino`setColor(<Orange>);
   Tetramino`setId(3);
   Tetramino`initialSetMinoes(game, [1, 6]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     cases index:
      1 -> result(1) := position(1) + 1,
      others -> result(2) := position(2) - 1
     end
    ),
    1 -> (
     cases index:
      1 -> result(2) := position(2) - 1,
      others -> result(1) := position(1) - 1
     end
    ),
    2 -> (
     cases index:
      1 -> result(1) := position(1) - 1,
      others -> result(2) := position(2) + 1
     end
    ),
    3 -> (
     cases index:
      1 -> result(2) := position(2) + 1,
      others -> result(1) := position(1) + 1
     end
    )
   end;
   return result
```

```
  )
  pre index in set {1, ..., 4}
   and Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> result(1) := position(1) + 2,
    1 -> result(2) := position(2) - 2,
    2 -> result(1) := position(1) - 2,
    3 -> result(2) := position(2) + 2
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);

end TetraminoL
```

Function or operation	Line	Coverage	Calls
TetraminoL	12	100.0%	2
getNextMino	20	100.0%	146
getRotatedMino	55	100.0%	4
TetraminoL.vdmpp		100.0%	152

9 TetraminoO

```
class TetraminoO is subclass of Tetramino

 -- Tetramino of the type J: xx
 --          xx
 --
 -- Minoes:       12
 --          43


 operations

  public TetraminoO : (Game) ==> TetraminoO
  TetraminoO(game) == (
   Tetramino`setColor(<Yellow>);
   Tetramino`setId(4);
   Tetramino`initialSetMinoes(game, [1, 5]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     cases index:
```

```
      1 -> result(2) := position(2) + 1,
      2 -> result(1) := position(1) + 1,
      others -> result(2) := position(2) - 1
     end
    ),
    1 -> (
     cases index:
      1 -> result(1) := position(1) + 1,
      2 -> result(2) := position(2) - 1,
      others -> result(1) := position(1) - 1
     end
    ),
    2 -> (
     cases index:
      1 -> result(2) := position(2) - 1,
      2 -> result(1) := position(1) - 1,
      others -> result(2) := position(2) + 1
     end
    ),
    3 -> (
     cases index:
      1 -> result(1) := position(1) - 1,
      2 -> result(2) := position(2) + 1,
      others -> result(1) := position(1) + 1
     end
    )
   end;
   return result
  )
  pre index in set {1, ..., 4}
   and Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> result(2) := position(2) + 1,
    1 -> result(1) := position(1) + 1,
    2 -> result(2) := position(2) - 1,
    3 -> result(1) := position(1) - 1
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);

end TetraminoO
```

Function or operation	Line	Coverage	Calls
TetraminoO	11	100.0%	3
getNextMino	19	100.0%	250
getRotatedMino	58	100.0%	4
TetraminoO.vdmpp		100.0%	257

10 TetraminoS

```
class TetraminoS is subclass of Tetramino

 -- Tetramino of the type J:  xx
 --                          xx
 --
 -- Minoes:       21
 --               43


 operations


  public TetraminoS : (Game) ==> TetraminoS
  TetraminoS(game) == (
   Tetramino`setColor(<Green>);
   Tetramino`setId(5);
   Tetramino`initialSetMinoes(game, [1, 6]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     cases index:
      2 -> result(1) := position(1) + 1,
      others -> result(2) := position(2) - 1
     end
    ),
    1 -> (
     cases index:
      2 -> result(2) := position(2) - 1,
      others -> result(1) := position(1) - 1
     end
    ),
    2 -> (
     cases index:
      2 -> result(1) := position(1) - 1,
      others -> result(2) := position(2) + 1
     end
    ),
    3 -> (
     cases index:
      2 -> result(2) := position(2) + 1,
      others -> result(1) := position(1) + 1
     end
    )
   end;
   return result
  )
  pre index in set {1, ..., 4}
   and Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> result(1) := position(1) + 2,
    1 -> result(2) := position(2) - 2,
    2 -> result(1) := position(1) - 2,
```

```
    3 -> result(2) := position(2) + 2
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);

end TetraminoS
```

Function or operation	Line	Coverage	Calls
TetraminoS	12	100.0%	6
getNextMino	20	100.0%	416
getRotatedMino	55	100.0%	5
TetraminoS.vdmpp		100.0%	427

11 TetraminoT

```
class TetraminoT is subclass of Tetramino

 -- Tetramino of the type J:  x
 --       xxx
 --
 -- Minoes:        1
 --          234


 operations


  public TetraminoT : (Game) ==> TetraminoT
  TetraminoT(game) == (
   Tetramino`setColor(<Purple>);
   Tetramino`setId(6);
   Tetramino`initialSetMinoes(game, [1, 5]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     cases index:
      1 -> (
       result(1) := position(1) + 1;
       result(2) := position(2) - 1;
      ),
      others -> result(2) := position(2) + 1
     end
    ),
    1 -> (
     cases index:
      1 -> (
       result(1) := position(1) - 1;
       result(2) := position(2) - 1;
```

```
      ),
      others -> result(1) := position(1) + 1
     end
    ),
    2 -> (
     cases index:
      1 -> (
       result(1) := position(1) - 1;
       result(2) := position(2) + 1;
      ),
      others -> result(2) := position(2) - 1
     end
    ),
    3 -> (
     cases index:
      1 -> (
       result(1) := position(1) + 1;
       result(2) := position(2) + 1;
      ),
      others -> result(1) := position(1) - 1
     end
    )
   end;
   return result
  )
  pre index in set {1, ..., 4}
   and Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     result(1) := position(1) + 1;
     result(2) := position(2) + 1;
     ),
    1 -> (
     result(1) := position(1) + 1;
     result(2) := position(2) - 1;
     ),
    2 -> (
     result(1) := position(1) - 1;
     result(2) := position(2) - 1;
     ),
    3 -> (
     result(1) := position(1) - 1;
     result(2) := position(2) + 1;
     )
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);

end TetraminoT
```

Function or operation	Line	Coverage	Calls
TetraminoT	12	100.0%	2

getNextMino	20	100.0%	157
getRotatedMino	67	100.0%	5
TetraminoT.vdmpp		100.0%	164

12 TetraminoZ

```
class TetraminoZ is subclass of Tetramino

 -- Tetramino of the type Z: xx
 --         xx
 --
 -- Minoes:      12
 --             34


 operations


  public TetraminoZ : (Game) ==> TetraminoZ
  TetraminoZ(game) == (
   Tetramino`setColor(<Red>);
   Tetramino`setId(7);
   Tetramino`initialSetMinoes(game, [1, 4]);
   return self
  );


  protected getNextMino: Board`Position * nat ==> Board`Position
  getNextMino(position, index) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> (
     cases index:
      2 -> result(1) := position(1) + 1,
      others -> result(2) := position(2) + 1
     end
    ),
    1 -> (
     cases index:
      2 -> result(2) := position(2) - 1,
      others -> result(1) := position(1) + 1
     end
    ),
    2 -> (
     cases index:
      2 -> result(1) := position(1) - 1,
      others -> result(2) := position(2) - 1
     end
    ),
    3 -> (
     cases index:
      2 -> result(2) := position(2) + 1,
      others -> result(1) := position(1) - 1
     end
    )
   end;
   return result
  )
  pre index in set {1, ..., 4}
   and Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
```

```
  post Tetramino`checkPosition(RESULT, 0, Board`maxRow + 1, 0, Board`maxColumn + 1);


  protected getRotatedMino: Board`Position ==> Board`Position
  getRotatedMino(position) == (
   dcl result : Board`Position := position;
   cases Tetramino`getOrientation():
    0 -> result(2) := position(2) + 2,
    1 -> result(1) := position(1) + 2,
    2 -> result(2) := position(2) - 2,
    3 -> result(1) := position(1) - 2
   end;
   return result
  )
  pre Tetramino`checkPosition(position, 1, Board`maxRow, 1, Board`maxColumn)
  post Tetramino`checkPosition(RESULT, -1, Board`maxRow + 2, -1, Board`maxColumn + 2);

end TetraminoZ
```

Function or operation	Line	Coverage	Calls
TetraminoZ	12	100.0%	2
getNextMino	20	100.0%	124
getRotatedMino	55	100.0%	4
TetraminoZ.vdmpp		100.0%	130