Project separated to two projects – class library KingSuvival.Common and console application KingSuvival.UI. The methods are separated in classes to follow the OOP design.

Structure RC from class Program refactored as follows:

* Renamed to MatrixCoordinates structure, which stays internal for the KingSuvival.Common library to protect it. Public fields “r” and “c” renamed to “row” and “column”, made private and encapsulated with public properties “Row” and “Column”.
* Created Pawn class which represents the pawns A, B, C, D and King. The class have public properties for symbol, x-coordinate and y-coordinate (Matrix coordinates) of the pawn it represents.
* Created King class which represents the king which inherits the Pawn class and uses constant symbol for representation

Methods “PE4AT\_DASKA” and “find” moved to class Board and refactored as follows:

* Introduced constants “HorizontalBorderSymbol”, “VerticalBorderSymbol”, “BlackCellSymbol”, “WhiteCellSymbol”, “WhiteSpaceSymbol” holding the needed symbols for the image of the board
* Introduced private fields “image”, “boardRows” and “boardColumns”
* Introduced properties “BoardRows” and “BoardColumns” which hold the size of the board
* Method “PE4AT\_DASKA” renamed to “GetImage” which returns string and is now testable and separated to small private methods which get the image of the board at the moment. The private methods are: “AppendRowsLine”, “AppendRowAndColumnIndicators”, “AppendBorder”. They use the constants described above to implement the logic for constructing the image of the board. Private method “GetSymbol” get the symbol of the pawn to be appended to the image of the board.
* --------------------------------------------------------------------------------------------------------
* Reformatted the source code
  + In class GameManager: method proverka()
  + Split the lines containing several statements into several simple lines
  + Formatted the curly braces **{** and **}** according to the best practices for the C# language.

|  |
| --- |
| if (notOverlapedRow == overlap1.Row && notOverlapedColumn == overlap1.Column) return true;  else if (notOverlapedRow == overlap2.Row && notOverlapedColumn == overlap2.Column) return true;  else if (notOverlapedRow == overlap3.Row && notOverlapedColumn == overlap3.Column) return true;  else if (notOverlapedRow == overlap4.Row && notOverlapedColumn == overlap4.Column) return true;  else  return false; |
| ↓ |
| if (notOverlapedRow == overlap1.Row && notOverlapedColumn == overlap1.Column)  {  return true;  }  else if (notOverlapedRow == overlap2.Row && notOverlapedColumn == overlap2.Column  {  return true;  }  else if (notOverlapedRow == overlap3.Row && notOverlapedColumn == overlap3.Column)  {  return true;  }  else if (notOverlapedRow == overlap4.Row && notOverlapedColumn == overlap4.Column)  {  return true;  }  else  {  return false;  } |

* Renamed method
  + In class GameManager
    - method proverka() → method IsAvailableNextPosition()
    - method isMoveLeft() → method IsValidMove()
* Renamed method parameter
  + In Class GameManager: method IsAvailableNextPosition()
    - Parameter notOverlapedXCoordinate → isAvaliableXCoordinate
    - Parameter notOverlapedYCoordinate → isAvaliableYCoordinate
    - Parameter overlap1 → figureOne
    - Parameter overlap2 → figureTwo
    - Parameter overlap3 → figureThree
    - Parameter overlap4 → figureFour
  + In Class GameManager: method isValidMove()
    - Parameter A → pawnA
    - Parameter B → pawnB
    - Parameter C → pawnC
    - Parameter D → pawnD
    - Parameter K → king
* Renamed variables
  + In class GameManager: method IsValidMove()
    - String move → command
* Change cases in method isValideMove()
  + From uppercase to lowercase
* Add new bool variable
  + In class GameManager
    - Method IsAvailableNextPosition()
      * Add bool isAvalable. This variable return true or false according the checks in method.
    - Method IsValidMove()
      * Add bool isValid. This variable return true or false according the checks in method.
* Change method implementation
  + In class GameManager: method IsValidMove() – separate this method to two. One for King’s (IsValidKingMove()) and another for Pawn’s (IsValidPawnMove()).
* Reverse logic
  + in method IsAvailableNextPosition() – return **true** when next position is available

-\*-\*-\*

Private method “proverka1” refactored to new method “CanKingMove”. This new method divides the old one to several new : “CanKingMove”, “IsKingUpLeftMovementAvailable”, “IsKingUpRightMovementAvailable”,”IsKingDownLeftMovementAvailable”, “IsKingDownRightMovementAvailable”, “IsKingTrapped”. Explanation of methods:

* “CanKingMove” - Checks, if king is restricted to any of four directions – if king can move at least at one direction returns true, else if king is trapped returns false.
* “IsKingUpLeftMovementAvailabe” – checks, if king up left position is available for movement. First determines where king is – on border or not and from that fact returns if king can move up left.
* “IsKingUpRightMovementAvailable” – checks, if king up right position is available for movement. First determines where king is – on border or not and from that fact returns if king can move up right.
* “IsKingDownLeftMovementAvailable” – checks, if king down left position is available for movement. First determines where king is – on border or not and from that fact returns if king can move down left.
* “IsKingDownRightMovementAvailable” – checks, if king down right position is available for movement. First determines where king is – on border or not and from that fact returns if king can move down right.
* “IsKingTrapped” – checks if pawn is near king and if king is on border and from this fact returns boolean value if king can move.

Private method “proverka2” refactored to new method “HasGameEnded”. This method divides the old one to several new: “HasGameEnded”, “HasKingWon”, and “DisplayCurrentEndOnConsole”. Explanation of this methods:

* “HasGameEnded” – first checks which pawn type is on turn. If “king” is on turn checks where is king, if is on top of board (XCoordiante = 0) -> game ends, and king wins, else checks with methods “CanKingMove” and “IsKingTrapped” different situation of game. If king is trapped and can’t move -> game ends, king loses -> method returns true. If all of these situations are not fulfilled, games continues -> method returns false.
* “HasKingWon” - using logic from “HasGameEnded” determines if king has won, returns boolean value -> if king has won – returns true, else returns false.
* “DisplayCurrentEndOnConsole” – if game has ended, checks if king has won and prints on console current ending and turns of king pawn.