Refactoring Documentation for Project “King Survival”

Team “Erbium”

Provided project separated to three projects – class library **KingSuvival.Common**, console application **KingSuvival.UI** and unit test project **KingSurvival.Tests**. The methods are separated in classes to follow the OOP design.

Structure *RC* from class *Program* refactored as follows:

* Code reformatted
* 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**. Operators **==, !=, +, -** and methods **Equals** and **GetHashCode** implemented for full functional usage of the structure.
* Created **Pawn** class which represents the pawns **A, B, C, D** and **King**. The class has **public properties** for instance’s **symbol** and **coordinates** (Matrix coordinates).
* Created **King** class which represents the king pawn. The class **inherits Pawn** class and uses **constant symbol** for representation.

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

* Code reformatted
* Introduced **constants** **HorizontalBorderSymbol**, **VerticalBorderSymbol**,

**BlackCellSymbol**, **WhiteCellSymbol**, **WhiteSpaceSymbol** holding the needed symbols for the image of the board

* Introduced **private field image** whichhold the image
* Introduced **private fields** **rows and columns** and encapsulated with **public properties Rows and Columns** 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: **AppendRowIndicatorsLine**, **AppendBody**, **AppendBorder**. They use the constants described above to implement the logic for constructing the image of the board.
* Method *find* renamed to private method **GetSymbol** which gets 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 **foreach**

|  |
| --- |
| *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;* |
| to |
| foreach (MatrixCoordinates coordinates in pawnsCoordinates)  {  if (newCoordinates == coordinates)  {  return false;  }  } |

* Renamed method
  + In class GameManager
    - method *proverka*() to method **IsAvailableNextPosition**()
    - method *isMoveLeft*() to method **IsValidMove**()
* Change method parameter
  + In Class GameManager: method IsAvailableNextPosition()
    - Change *notOverlapedXCoordinate, notOverlapedYCoordinate*  parameters with **MatrixCoordinates newCoordinates**
    - Change *overlap1, overlap2, overlap3, overlap4* parameters with **params MatrixCoordinates[] pawnsCoordinates**
  + In Class GameManager: method isValidMove()
    - Parameter *A* to **pawnA**
    - Parameter *B* to **pawnB**
    - Parameter *C* to **pawnC**
    - Parameter *D* to **pawnD**
    - Parameter *K* to **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 returns true or false according to the checks in method.
    - Method **IsValidMove**()
      * Add **bool isValid**. This variable returns true or false according to 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

Method *proverka1* refactored to new methods **CanAtLeastOnePawnMove**. This new method divides the old one to several new: **CanAtLeastOnePawnMove**, **CanCurrentPawnMove**. Explanation of methods:

* **CanAtLeastOnePawnMove –** In every turn, checks movement “ability” of every pawn using boolean variables returned from ***CanCurrentPawnMove()***. If at least one pawn can move (used boolean OR logic) returns *true*, if every pawn is “trapped” returns *false*
* **CanCurrentPawnMove –**  Gets every pawn coordinates, and check if *currentPawn*, next coordinates are available, it’s determined by this way :
  + If pawn is border (due to hostile pawns can move only downside) checks only down diagonal position if is available , using ***IsAvailableNextPosition();***
  + If pawn is “in” field , checks down-right and down-left diagonal positions if are avaible, again with method ***IsAvailableNextPosition();***
  + If king is on top of board , all pawns are forced to be “trapped”;

Method *proverka2* refactored to new method **HasGameEnded**. This method divides the old one to several new: **HasGameEnded**, **DisplayCurrentEndOnConsole, HasKingWon, CanKingMove, IsKingUpRightMovementAvailable, IsKingDownRightMovementAvailable, IsKingUpLeftMovementAvailable and IsKingDownLeftMovementAvailable.** Explanation of this methods:

* **HasGameEnded** – this method is really a programming-logic representation of game logic, so this game returns boolean variables in several conditions :
  + ***True -*** 
    - If king is on top of board using king pawn current coordinates;
    - if all hostile pawns are “trapped” using ***CanAtLeastOnePawnMove();***
    - ***i***f king is trapped by hostile pawns, checks are made using ***CanKingMove();***
  + ***False –*** In every other condition;
* **HasKingWon** – first checks if game is ended by boolean variable from ***HasGameEnded,*** if game is ended, returns boolean variable in several conditions :
  + ***True -*** in conditions :
    - If king is on top of board using king pawn current coordinates;
    - if all hostile pawns are “trapped” using ***CanAtLeastOnePawnMove();***
  + ***False –***If king is trapped by hostile pawns, checks are made using ***CanKingMove();***
* **DisplayCurrentEndOnConsole** – determines if king has won , using ***HasKingWon***, and from this displaying different ends along with playing board :
  + *King wins in \_\_ turns!*
  + *King loses in \_\_ turns …*
* **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, checks are made with current methods for current direction:
  + ***Up-Right*** – determined by ***IsKingUpRightMovementAvailable();***
  + ***Up-Left*** – determined by ***IsKingUpLeftMovementAvailable();***
  + ***Down-Right*** – determined by ***IsKindDownRightMovementAvailable();***
  + ***Down-Left*** – determined by ***IsKingDownLeftMovementAvailable();***
* **IsKingUpLeftMovementAvailable** – checks, if king up left position is available for movement. First determines where king is – on border or not and from this conclusion returns boolean value if king can move up left. Checks are made using actual king coordinates and **IsAvailableNextPosition*()*** for certain position.
* **IsKingUpRightMovementAvailable** – checks, if king up right position is available for movement. First determines where king is – on border or not and from this conclusion returns boolean value if king can move up right. Checks are made using actual king coordinates and **IsAvailableNextPosition*()*** for certain position.
* **IsKingDownLeftMovementAvailable** – checks, if king down left position is available for movement. First determines where king is – on border or not and this conclusion returns boolean value if king can move down left. Checks are made using actual king coordinates and **IsAvailableNextPosition*()*** for certain position.
* **IsKingDownRightMovementAvailable** – checks, if king down right position is available for movement. First determines where king is – on border or not this conclusion returns boolean value if king can move down right. Checks are made using actual king coordinates and **IsAvailableNextPosition*()*** for certain position.