Refactoring Documentation for Project “King-Survival-3”

# Team King-Survival-3

* Дженко Пенев – **dzhenko**
* Никола Тодоров – **nmtodorov**
* Ясен Михайлов – **Yasen**
* Павел Христов – **lithical**
* Светлин Няголов – **tinskiwe**
* Лъчезар Томов – **lbtomov**
* Мартин Маринов – **Martin\_N\_Marino**
* Христо Илиев - **Hristo.Iliev**

# Redesigned the project structure

* Renamed the solution to KingSurvival.
* Included 3 projects – KingSurvival, UnitTestsKingSurvival, OriginalCode
* Renamed the main class Program to MainGame.
* Extracted each class in a separate file with a good name: PieceKing.cs, GameBoard.cs, etc.
* Extracted interfaces and moved them into a separate project folder
* Extracted enumerations and moved them into a separate project folder
* Introduced several design patterns described in point 6

# Reformatted the source code

Removed all unneeded empty lines, e.g. in the method proverka2 ().

Inserted empty lines between the methods.

Split the lines containing several statements into several simple lines, e.g.:

if (input[i] != ' ') break; -> if (input[i] != ' ')

{

break;

}

Formatted the curly braces { and } according to the best practices for the C# language.

Put { and } after all conditionals and loops (when missing).

Character casing: variables and fields made camelCase; types and methods made PascalCase.

Formatted all other elements of the source code according to the best practices introduced in the “[High-Quality Programming Code](http://telerikacademy.com/Courses/Courses/Details/174)” course.

# Refactoring

* private static bool proverka2(int turn, RC A, RC B, RC C, RC D, RC K)

***became:***

private void CheckGameState()

Where state of the game is maintained in GameEngine properties.private

* static bool proverka1(RC pawn, RC obstacle1, RC obstacle2, RC obstacle3, RC obstacle4)

***became:***

private bool ArePiecesStuck(IList<IPiece> piecesToCheck)

Which is possible because all pieces are within collection and thanks to the strategy each player can figure if he/she has valid moves left.

* private static bool isMoveLeft(int turn, ref RC A, ref RC B, ref RC C, ref RC D, ref RC K)

**and**

private static bool proverka(int notOverlapedRow, int notOverlapedColumn, RC overlap1, RC overlap2, RC overlap3, RC overlap4)

***became:***

private bool IsPossibleMove(IPiece pieceToMove, ICoordinates newPieceCoordinates)

Move does not happen if it is not valid and both checks come into one as all pieces are within collection.

* do ... while (!kraj)

***became:***

while (this.gameIsRunning)

# Design

* Game Initialization is put in sepparate class where all the pieces come together:

public Game()

{

var decoratedRenderer = new TextRendererDecorator(new RendererConsole());

var renderer = new GameRendererAdaptor(decoratedRenderer);

var controller = new GameController();

this.engine = new GameEngine(renderer, controller);

}

* Piece logic is abstracted through strategy pattern.

LogicPieceMover holds different strategies per player and finds appropriate piece within the collection, that can carry the command:

public IPiece FindPieceToMove(ICommand command, IList<IList<IPiece>> allPieces, out bool addScore)

{

return this.pieceMoverStrategy.FindPieceToMove(command, allPieces, out addScore);

}

Thus defeating the dreaded switch-case & endless if-else blocks.

Every PieceStrategy extend base class so new strategies can be implemented.

* GameEngine class to put all pieces together and model behaviour and logic of the game implementation.

# Introduced constants

* PawnAStartRow = 0;
* PawnAStartCol = 0;
* PawnAName = 'A';
* PawnBStartRow = 0;
* PawnBStartCol = 2;
* PawnBName = 'B';
* PawnCStartRow = 0;
* PawnCStartCol = 4;
* PawnCName = 'C';
* PawnDStartRow = 0;
* PawnDStartCol = 6;
* PawnDName = 'D';
* KingStartRow = 7;
* KingStartCol = 3;
* KingName = 'K';
* MessageShowTimeMilliseconds = 2000;
* MessageToPlayerOffset = 2;
* IllegalMove = "Illegal move!";
* Player1Turn = "King's turn: ";
* Player2Turn = "Pawns' turn: ";
* KingWinsInXTurns = "King wins in {0} turns.";
* KingLooses = "King loses.";
* CommandMoveUpRight = "UR";
* CommandMoveUpLeft = "UL";
* CommandMoveDownRight = "DR";
* CommandMoveDownLeft = "DL";

# Introduced design patterns

### Creational design patterns

* Abstract Factory – PawnsAndKingsFactory.cs (implementing IGamePieceFactory) taking care of creating instances of player 1 and player 2 objects (currently PieceKing.cs and PiecePawn.cs)
* Builder – PlayersGamePieceBuilder.cs as the base builder, Player1GamePieceBuilder.cs and Player2GamePieceBuilder.cs acting as the concrete builders and PlayerGamePieceDirector.cs acting as the director, having the sole responsibility of creating all of the game pieces properly. Uses the abstract factory as objects instance creator and GameConstants.cs to assign the correct properties of each game piece.
* Singleton – Since only one game board is needed for every game the GameBoard.cs is a singleton implementation, allowing both extending the class and only using its sole instance.

### Structural design patterns

* Decorator – RendererBase.cs as the component base, RendererConsole.cs as the concrete component, TextRendererDecoratorBase.cs as the decorator base and TextRendererDecorator.cs as the component decorator aiming to add method for rendering text as well.
* Adaptor – GameRenderer.cs is implementing the IRenderer interface by using inner TextRendererDecoratorBase and translating to it IRenderable objects as 2D char arrays and coordinates as x and y offset.
* Façade – PlayersAllGamePiecesCreator.cs – having the only responsibility to create all the game pieces needed for the game. Innerly using large amount of classes (factory, builders etc.) returning a List of player pieces.

### Behavioral design patterns

* Observer – IGamePieceObserver.cs interface acting as the observer base and GameBoard.cs as the concrete observer are observing the IPiece subject base (the moved event) and its implementations – the abstract Piece.cs and its implementations KingPiece and PawnPiece. Every time the move method of any of the pieces is invoked, the event moved is fired allowing the observer to react properly, which in this case is to show the new position of the game piece and delete its old one.
* Strategy – LogicPlayerPieceMoveBase.cs as the strategy base and the concrete strategies LogicPlayer1PieceMover.cs and LogicPlayer2PieceMover.cs as its implementations. Both aim at supplying the LogicPieceMover.cs with concrete implementations of the abstract LogicPlayerPieceMoveBase.cs so that the inner method FindPieceToMove can work differently depending on the player that is in turn.

# Created code documentation

* A HTML documentation was created using Sand Castle - [click](help/index.html)