**Development document**

# **Requirements of the game:**

* Presence of a start screen for the "Key Barricade" game, with a start button.
* A square playing surface with fixed walls and barricades.
* The player is operated with the use of arrow keys.
* The player is able to collect keys in the "pocket". This key disappears from the maze can therefore not be put back on the playing field.
* The key in the "pocket" disappears when another key is picked up and the current key held by the player appears on the screen.
* The player may only hold one key, but use it multiple times.
* Barricades can only be removed from the playing field with a suitable key. As an example: a key with pin 100 only fits on a barricade with pin 100 and a key with pin 300 fits only on a barricade with pin 300 etc.
* When a barricade with an incorrect key tries to be opened, a message appears with an incorrect key.
* When the player reaches the end field, a message appears to return to the home screen or exit the game.
* During the game it must be possible to restart the same playing field.
* The game must be capable of adding extensions at a later time. (To show the expandability an extra collectable item has been implemented in the system.)

# **Analyse of the class diagram:**

See the document “*BarricadeGame\_ClassDiagram.vpp*” for the class diagram.

The whole program has been designed having in mind the Model-View-Controller pattern.

Where the *Map/NumericBlock/StandardBlock* are acting as the model, *GameConsole/GameUI* as view and the *GameController* is as the controller.

*GameConsole* is just a container class extension of JFrame which creates the *GameUI* (this class is painted on this class). It is used only to set the program icon, the windows size and some other attributes.

*GameUI*, this class is an extension of JPanel and it contains all the major graphical element and the keyboard listener. This class is used to visualize the start menu and the game map. This class send the key pressed on the keyboard to the *GameController* and reads from it a new map, then it updates the visualized map on the screen with the new one.

During the game, it is visualized a text detailing the code of the key the user picked up from the ground. Also during the game it is possible to reload the game or change map by pressing two buttons on the screen.

*GameController* is the most important class, here based on the command received from the *GameUI* the next player movement and consequence of that is calculated. The result of this calculation will than read by the UI using the Map class. The controller extends the Observable class and it is able to send actively information to the Observer (*GameUI*), this is used to notify the user about the end of the game or if the key is wrong to open a door. This class beside generating the game map is also able to mutate that using special algorithm.

Map, this class contains two bidimensional arrays that are used to located items in the game. The two 2D arrays are an integer and the number are referenced to the *BlockType* enum. The *blockPosition* array as the name suggests define the position of all the elements and the *blockValue* define the value of the keys and the doors. These two will always have a reference one both arrays in the same coordinates.

*StandardBlock* is an extension of JButton and it is used to paint the block on the screen, the constructor receives the *BlockType* enum relative to the type of block and the button is created having as cover the right image. *NumericBlock* is an extension of *StandardBlock* and adds to the previous one a text field, this block is used to paint on screen the key and the door elements.

# **Analyse of the JUnit tests:**

* **testAssets**: This test is used to verify the presence of all the assets, to check if the images used in the game are present in the right folder and can be opened.

Input: Creation of all the block present in the game

Expected output: The program will not throw an exception

Actual output: Successful. (Because is equal to the expected one )

* **testINITConsole**: This test is used to check if the constructor of this class works without throwing errors

Input: Creation of the GameConsole class

Expected output: The class will not give any error

Actual output: Successful.

* **testINITController**: This test is used to check if the constructor of this class works without throwing errors

Input: Creation of the GameController class

Expected output: The class will not give any error

Actual output: Successful.

* **movementLogicTestRightAndLeft**: This test is used to check if the player can be moved on the map (Left and Right)

Input: Left and Right movement comand

Expected output: After sending the command to move the player Right/Left the position of the player is of one block to the Right/Left

Actual output: Successful.

* **movementLogicTestUpAndDown**: This test is used to check if the player can be moved on the map (Up and Down)

Input: Up and Down movement comand

Expected output: After sending the command to move the player Up/Down the position of the player is of one block to the Up/Down

Actual output: Successful.

* **antiWallCrossTest**: This test is used to check if it is possible or not cross the wall of the game

Input: Movement to the player towards the upper and left boundaries

Expected output: The player does not change is position, because hit a wall

Actual output: Successful.

* **gameSaveTest**: This test is used to test the game saving feature

Input: The game is saved and than the player is moved in the map, than the game save is reloaded

Expected output: The player returns to the previous position and with the previous key (if any)

Actual output: Successful.

* **wallCollisionTest**: This test is used to check the movement of the player towards wall

Input: The player is moved torwards a wall

Expected output: When the player hit the wall is position remain un-changed (do not cross it)

Actual output: Successful.

* **keyPickUPTest**: Test used to check the key picking up feature

Input: The player is moved on a key

Expected output: After the player across the key, in its inventory there should be the key and it’s positioned the same of the key he picked up

Actual output: Successful.