OBJECT-ORIENTED PROGRAMMING AND MODELING

Professor Saverio Delpriori

APPLIED COMPUTER SCIENCE

Goose Game

Project for the Academic Year 2020/2021
(summer session)

Student: Ani Yeganyan - 280570

Contents

1. Problem Specification

Creating a desktop application that simulates the classic game of the goose using the C# OOP language and Windows Forms; in particular the program is based on the following rules:

It is played on a board on which a spiral path made up of 63 squares is drawn (sometimes this number goes up to 90), marked with numbers or other symbols. The players start with a marker in the starting square and, in turn, proceed along the path of a number of squares obtained by rolling a pair of dice. The aim of the game is reach the central square of the spiral.

Some arrival boxes have a special effect; in the traditional version, the boxes that represent geese (hence the name of the game) allow you to immediately move forward one number of squares equal of those covered by the movement just made. These boxes are placed every nine starting from boxes 5 and 9 (a consequence of this arrangement is that a initial roll of 9 immediately brings the player to the final square and therefore to victory).

The other special boxes are the followings:

- square 6 "the bridge" the movement is repeated as in the boxes with geese;
- square 19 "house" or "inn" remaining stationary for three turns;
- squares 31 "water well" and 52 "prison" remaining stationary until another pawn reach the same square, which is ‘’imprisoned’’;
- square 42 "labyrinth" returns to number 39;
- square 58 "skeleton" returns to number 1.

The finish square (63 or 90) must be reached with an exact roll of the dice otherwise, when you reach the bottom, you move back some points in excess.

Study of the problem and architectural choices

- **Language**

  The programming language chosen for this program is C #. Language oriented to Objects (OOP) is very multiform especially to create desktop applications that they must be run on a Windows environment.

- **Visual Studio 2019 Community**

  The development environment used is Visual Studio 2019 Community Edition, realized free from Microsoft, with a .NET Framework at version 4.7.2, all on Windows 10 operating system; it has been chosen to use this development environment since being from Microsoft just like the language guarantees full compatibility of two offering a simple, but at the same time powerful, system for creating functional graphics aesthetically pleasing.

- **GUI**

  To obtain a level of the game very close to the real one, it has been opted for a graphical application, in fact with a console application we would never have obtained the same level of usability. The graphic application makes extensive use of Windows Forms and some contained classes within the development environment, including:

  - Button
  - Panel
  - Label
  - GroupBox
  - MessageBox

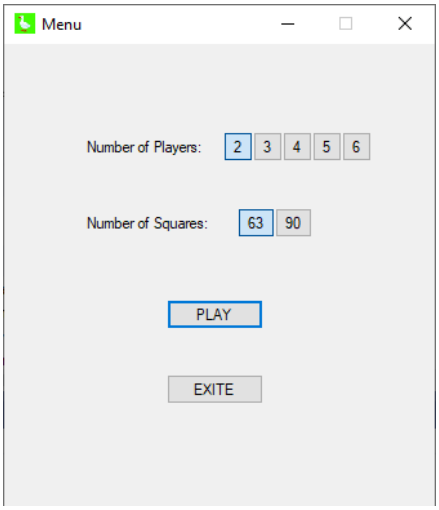
  Two Forms have been created for the graphic:

  - Form Menu
  - Form GameView

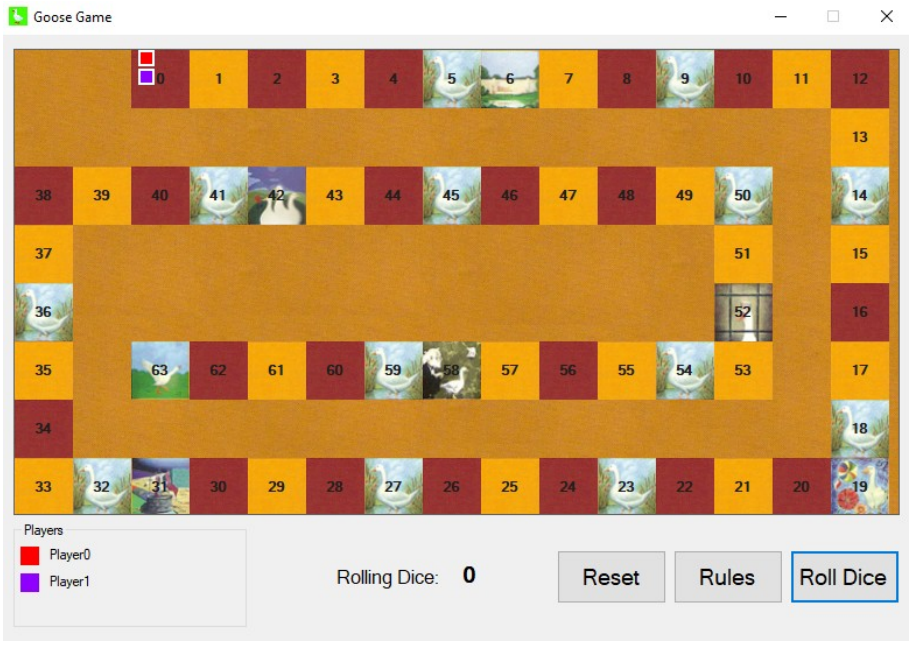
  The first is shown when the program starts, disconnected from the MVC logic is used only to ask the user with how many players and squares he wants to create the game, this form has a fixed size of 350x400.

The second Form is generated when the first is closed and is the real game table has fixed dimensions **800x400** and in central part the playing field is shown with all boxes spiraled clockwise (each box has an image of background which varies according to its type and has dimensions **50x50** if there are **63** squares and **40x40** if there are **90** squares), in the lower part, starting from the left, there is a sort of legend that shows the name of each player associated with the color of his pawn, below we have a *Label* that shows the result of the user's roll of the dice and finally three *Buttons: ''Rest'', ''Rules'', ''Roll Dice'',* which will allow us to reset the game, consult the rules and roll the dice.

An image of the Form *Menu* is shown below



and of the Form *GameView*



- **MVC**

By creating a graphical application it was almost obligatory to use Model-

View-Control(MVC) design pattern, which allows to divide the functional aspects of the program from the visual ones, giving the controller the task of acting as an intermediary between this two modules.

The MVC Design Pattern consists of three modules:

Model: it contains all the instructions that allow the logic of the program to function, for example, the ‘’Square’’, ‘’Pawn’’ and ‘’GameBoard’’ classes.

View: the difference between the Model, here are all the methods that create the visual appearance, therefore all the elements that make up the GUI.

Controller: Is an intermediate module and allows the exchange of information among the two others by means of methods invoked each time a given value is updated in the model or a button is clicked in the view.

Events

The methods invoked in the controller are called events, they are obtained from the Design Pattern Observer and are able to signal a situation of interest to other classes. The development environment provides its own implementation of the Observer called *EventHandler* which, once invoked, also brings with it arguments that they can be useful values for other functions.

Inheritance and Polymorphism

Within the program are the concepts of inheritance (a class derives attributes and methods from the other class) and polymorphism (use the class as if it is its derived class). Classes: *Normal, MoveForward, MoveBack, Inn, StayStill* are all derived from the *Square* class, while *ArgEvent* derives from the class native *EventArgs*. The only case of polymorphism is to use the Square class as if it were one of its sublasses to apply the effect method without worrying what type of square is.

- **The project structure**

  The solution fully respects the MVC pattern, is divided into three packages:

  - *Model:* contains the classes *GameBoard, Square, MoveForward, MoveBack, StayStill, Inn, Normal, Pawn, Dice, ArgEvent.*

- *View:* contains the GameView Form.

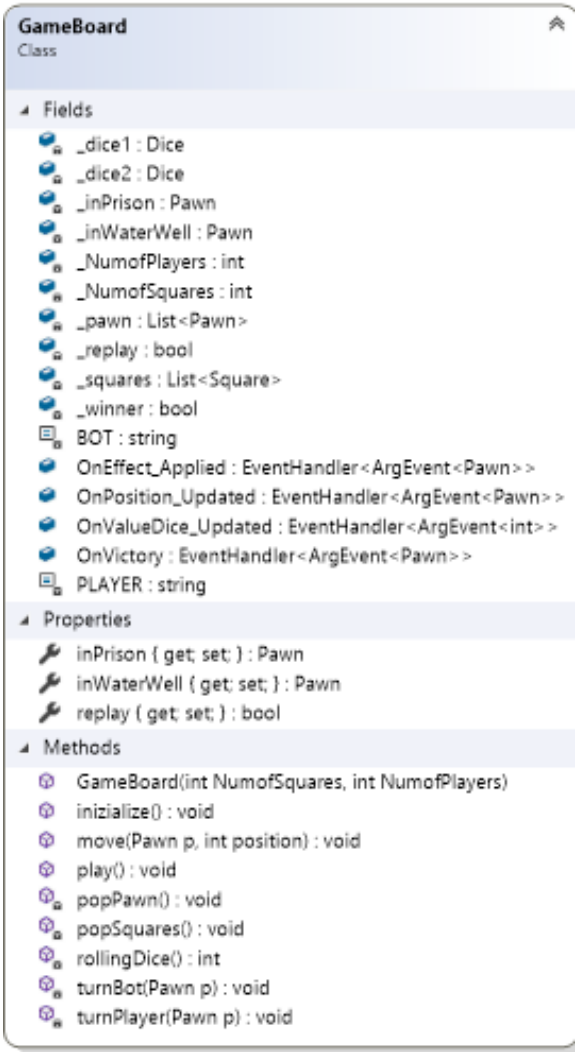
- *Controller:* contains the Controller class.

Also the *Model* package is contained in a different project called *GooseGameLib* and it is compiled into an external library (.dll) to further separate the functional logic from all the rest of the program.

- **Classes**

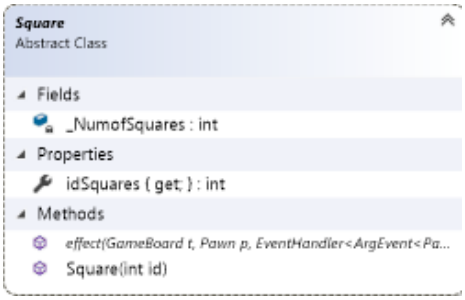
  - Model

  GameBoard



The main class is the GameBoard, which instantiates and uses all the other classes of the model. The class contains within two constants of sting type (*PLAYER, BOT*) which identify the type of the pawn, two attributes (*_NumofSquares,_NumofPlayers*) contain respectively the number of squares and players and their value will be passed to the constructor from the Main; below there are two lists (*_squares, _pawn*) containing the real squares and pawns, two attributes of type Dice represent the two dice, other attributes the pawn type and boolean are used to manage Inn, Water Well and Prison squares. Lastly, the type attributes EventHandler <ArgEvento> are used to communicate to the model the application of the effect (OnEffect_Applied), updating the value of the rolling dice (OnValueDice_Updated), the update of thechecker's position (OnPosition_Updated), a players victory (OnVictory).

Square



The Square class represents each of the squares and an abstract class that contains nor the attribute *_NumofSquares* which is a whole representative, as it suggests the name, number of the square; a property returns this value and a abstract method (effect), implemented by derived classes, applies the effect of the square.

Normal

The Normal class derives from the Square class and implements the effect method without executing no special instructions, since normal square has no effect.

MoveForward

The MoveForward class derives from the Square class and implements the effect method moving the pawn forward the same number of rolls it made to get there on.

MoveBack

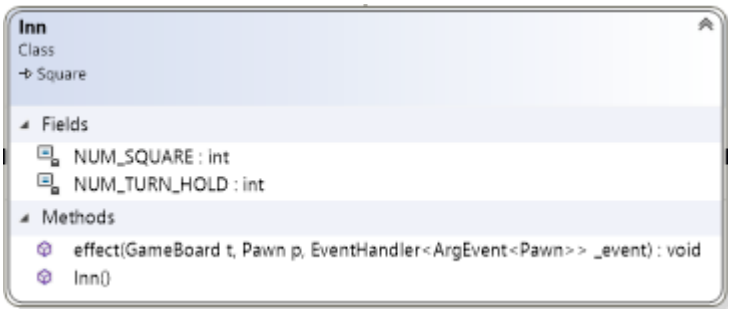
The MoveBack class derives from the Square class, it contains an intenger attribute *_destination* which represents the position in which the pawn must be moved and implement effect method by moving the pawn to the destination square which varies depending on square number.

StayStill

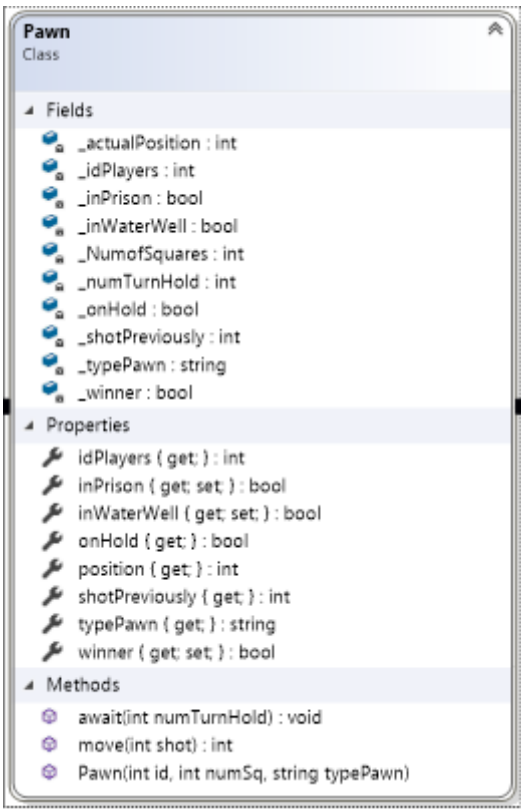
The StayStill class derives from the Square class and contains two constants *WATERWELL and PRISON* (the first one is in the

square number 31 and the second one is in the square number 52); the class implements the effect method keeping the pawn stationary until another pawn lands on the same square. Every time a pawn lands on this square, the variables *_inPrison, _inWaterWell* contained in the GameTable are chacked and if theycontain a pawn different than the current one frees it and takes its place.

Inn



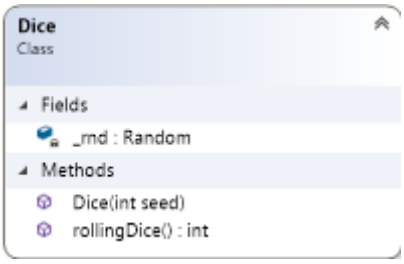
The Inn class derives from the Square class and contains two constants, *NUM_SQUARE, NUM_TURN_HOLD* (the first one is in the square number 19, the second one indicates how many turns pawn should be in hold), the class implements the effect method making the pawn stand still for a number of turns established by the constant *NUM_TURN_HOLD.*



Pawn

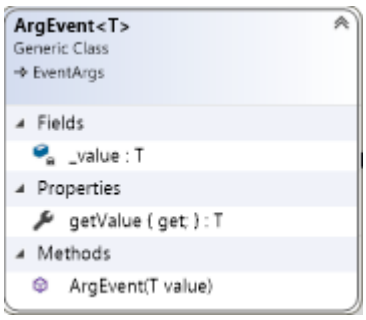
The Pawn class represents each of the pawns playing the game; within it contains some attributes which identify the pawn (*_idPlayers*), its type (*_typePawn*), its position (*_actualPosition*) its previous condition (*_shotPreviously*), if has won the game (*_winner*) and if the pawn is blocked in some way (*_onHold, _inWaterWell, _inPrison*) each of which is obtained from a specific property. The class exposes two main methods: move and await; the first one is used to deriv the position in which to move the pawn given in the input the number extracted from the roll of the dice, while the second one serves to put the pawn on hold for three turns whenever it ends up in an Inn-type square.

Dice



The Dice class represents the two game dice and contains as its only attribute a random number generator (*Random*) and exposes only one method (*rollingDice*) that returns an integer between 1 and 6.

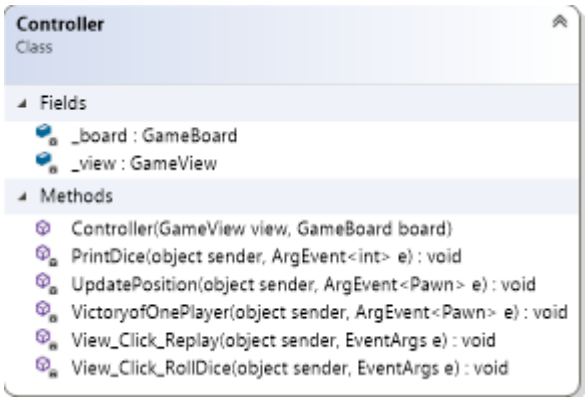
ArgEvent



The ArgEvent class derives from the *EventArgs* class, it is a generic class and has a purpose of passing a value of any type (in the project they are pawn and integer) to the events invoked.

- Controller

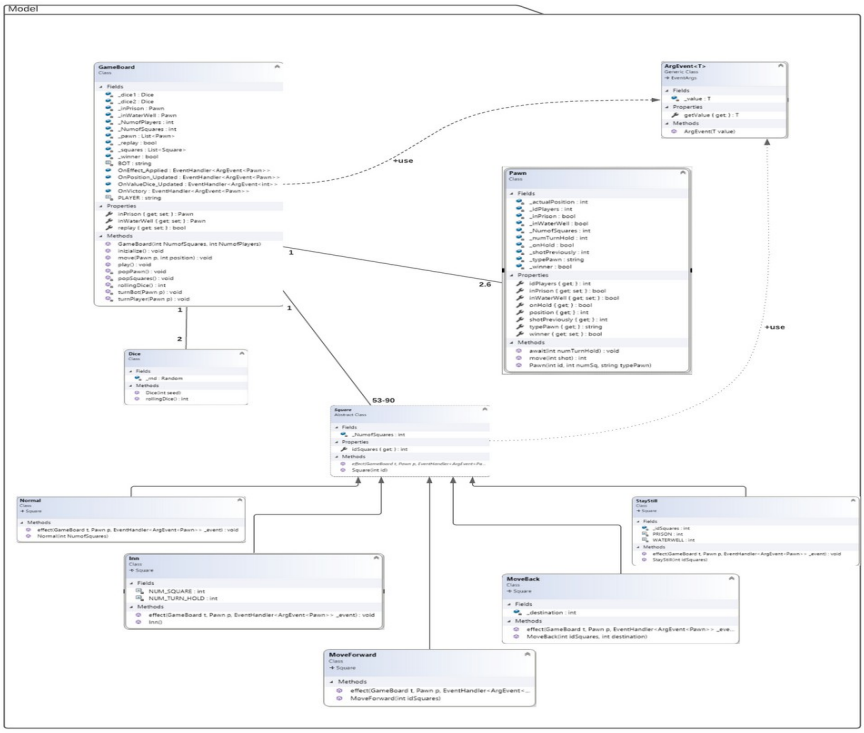
Controller



The Controller class contains a variable of the type (*_board*) and another of the type (*_view*) passed during its creation; the class contains some methods invoked when certain events occur: *UpdatePosition* is called every time in the model when a pawn is moved and communicates to the view its new position, *VictoryofOnePlayer* invoked as soon as a player reaches the last square, and it is communicated that the game is over. *PrintDice* is called each time in the model when a human player rolls the dice, *View_Click_RollDice* invoked each time in the view when the ''Roll Dice'' button is clicked and communicates to the model that must perform a new turn. *View_Click_Replay* invoked each time the ''Reset'' button is clicked in the view or it is chosen to relay the game, allows you to re-instantiate the main elements of the model and view.

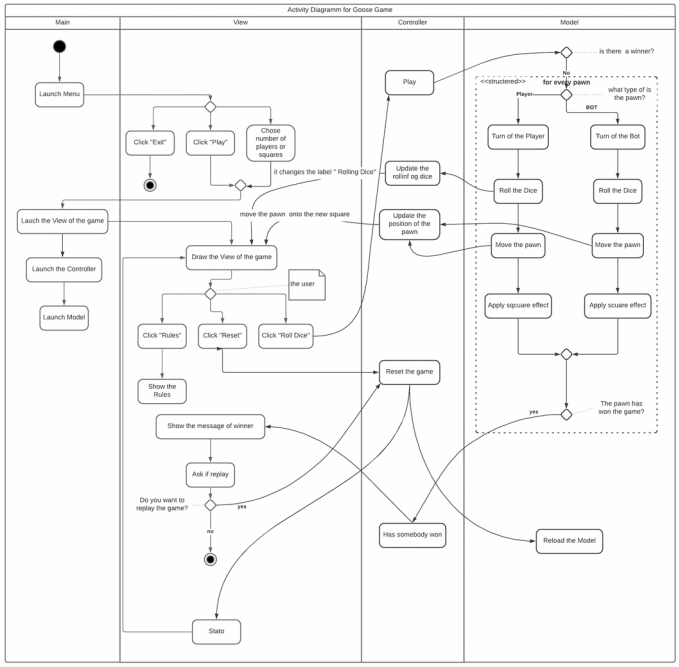
• **Class Diagramm**

The complete diagram classi is shown below in which are presented the different packages and the links between the various classes. In the *Controller* package it is present only the homonymous class, while in the *Model* there are all those that make it up. The *GameBoard* class is linked to the *Dice* class by a **1-2** association (only one table, two dice), to the *Pawn* class by an association **1-2..6** (one table, from two to six pawns) and to the *Square* class by an association **1-63 / 90** (a table, **63** or **90** boxes); plus the table has a "use" dependency on the *ArgEvent* class.The classes *Normal*, *Inn*, *MoveForward*, *MoveBack*, *StayStill* are all linked to the *Square* class it gives a generalization, since they are all its classes. The *Square* class has a dependency of type "use" with the *ArgEvent* class.



• **Activity Diagram**

The activity diagram is shown below, very similar to a flow diagram allows you to view the entire row of the program, highlighting any parallel flows (although there are none in this one).The diagram is divided into four sections: *Main, View, Controller, Model* each of which it represents a part of the project. The starting point is in the *Main* and starts the first menu, from here you go to the *View* where there are the only two exit points; continuing with the row you return to the *Main* to start the *GameView*, the *Controller* and *Model*. There is a section of loop in the model that executes the code inside it for each pawn. As you can see, each operation of the View performed on the Model passes through the controller and vice versa, from here we understand that the MVC design is fully respected.



Documentation on use

Development environment: Visual Studio 2019 Community Edition - .NET framework 4.7.2

To compile the game just follow these instructions:

- Extract the GooseGame folder from PMO.
- Double click on the .sln extension (Microsoft Visual Studio Solution) and wait for the loading the project
- Click on the Compile drop-down menu → Compile solution

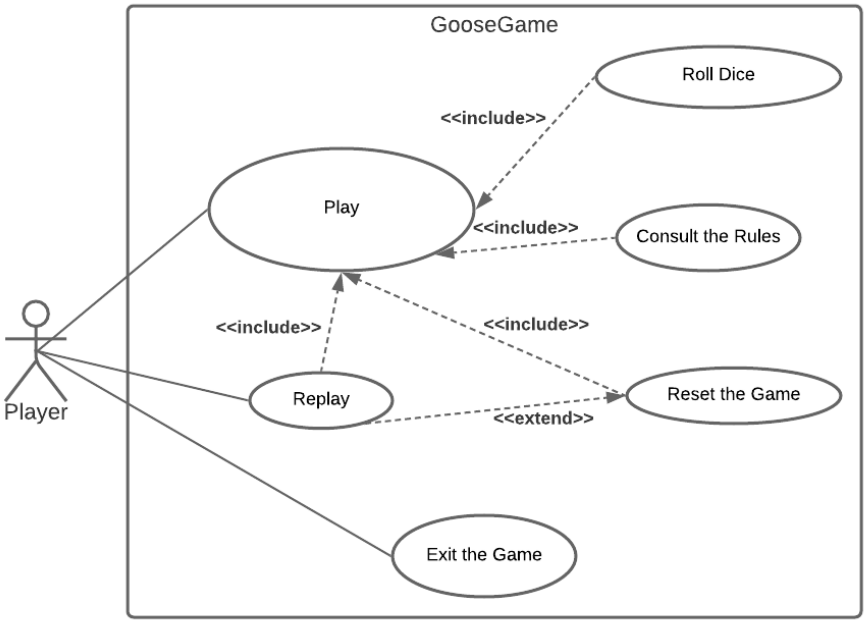
Once the compilation is complete, it will be possible to run the program by navigating to the course:

- GooseGame / bin / Debug
- Double click on the GooseGame.exe executable

Inside the / bin / Debug folder there are more self-generated files from the development environment, among these the only two to consider in order to copy the project elsewhere are the GooseGame.exe and GooseGameLib.dll

Use cases with relative UML schema

Below is the diagram of use cases to help to understand fully how the program works by a basic user.



The diagram has highlighted the participation of a single actor: the player, who must have opened the main Form containing the game menu:

- Play
  - the actor selects the number of players(2-6)
  - the actor select the number of squares(63 or 90)
  - click the ‘’Play’’ button

  If the actor clicks the " play " button without changing the number of players or squares, the game will start with 2 players and 63 squares.

The Form closes correctly and the Form containing the game board with the players and boxes selected previously is loaded, where the actor can play the game, consult the rules or reset the game.

- Roll Dice
  - the actor click the ‘’Roll Dice’’ button
  - the dice are rolled and the number is notified
  - a turn is performed for each pawn and their positions are updated

  The Form remains awaiting the actor.

- Consult the Rules
  - the actor click the ‘’Rules’’ button
  - a box containing all the rules appears on the screen
  - it is closed clicking ‘’OK’’

- Reset the Game
  - the actor click the ‘’Reset’’ button
  - the game board is restored to its initial conditions by returning all the pawn to the initial box (the initial number of players and squares are not going to be changed).

- Replay
  If the player has won the game:
  - a message is shown on the screen asking if you he wants to replay
  - if the actor click the button ‘’Yes’’, the game will be reset restoring the initial condition of the Form without changing the number of the players and squares
  - if the actor click the button ‘’No’’ the whole program will be terminated

- Exit the Game
  - the actor click the ‘’Exit’’ button in the Form menu
  - the Form containing the menu will be closed and the program will be terminate without loading the game board