Class Assignment – Yahtzee
Part C

Due Date: 7th June, 2016 Weighting: 40%

The Objective of Part C

This objective of part C of the assignment is to add some additional classes that will provide sufficient functionality to begin to play Yahtzee albeit without scoring.

UML Partial System Diagram

The UML System diagram on the next page shows the classes which provide the high level logic of the game. There are four (4) classes shown in the UML diagram. The details shown in the class diagram give the public methods and the essential private instance variables for each class.

The diagram shows the relationship between the classes using lines ending in diamond shaped arrowheads.

The **diamond shaped arrowhead** indicates that a class contains at least one object of another class, this is called a "**has-a**" relationship. For example, the **Game class** contains at least one **Die object** (we know that it is actually five.) as well the **Game class** contains at least one **Player object** and the **Player class** contains at least one **Score object** (actually 19 scores).

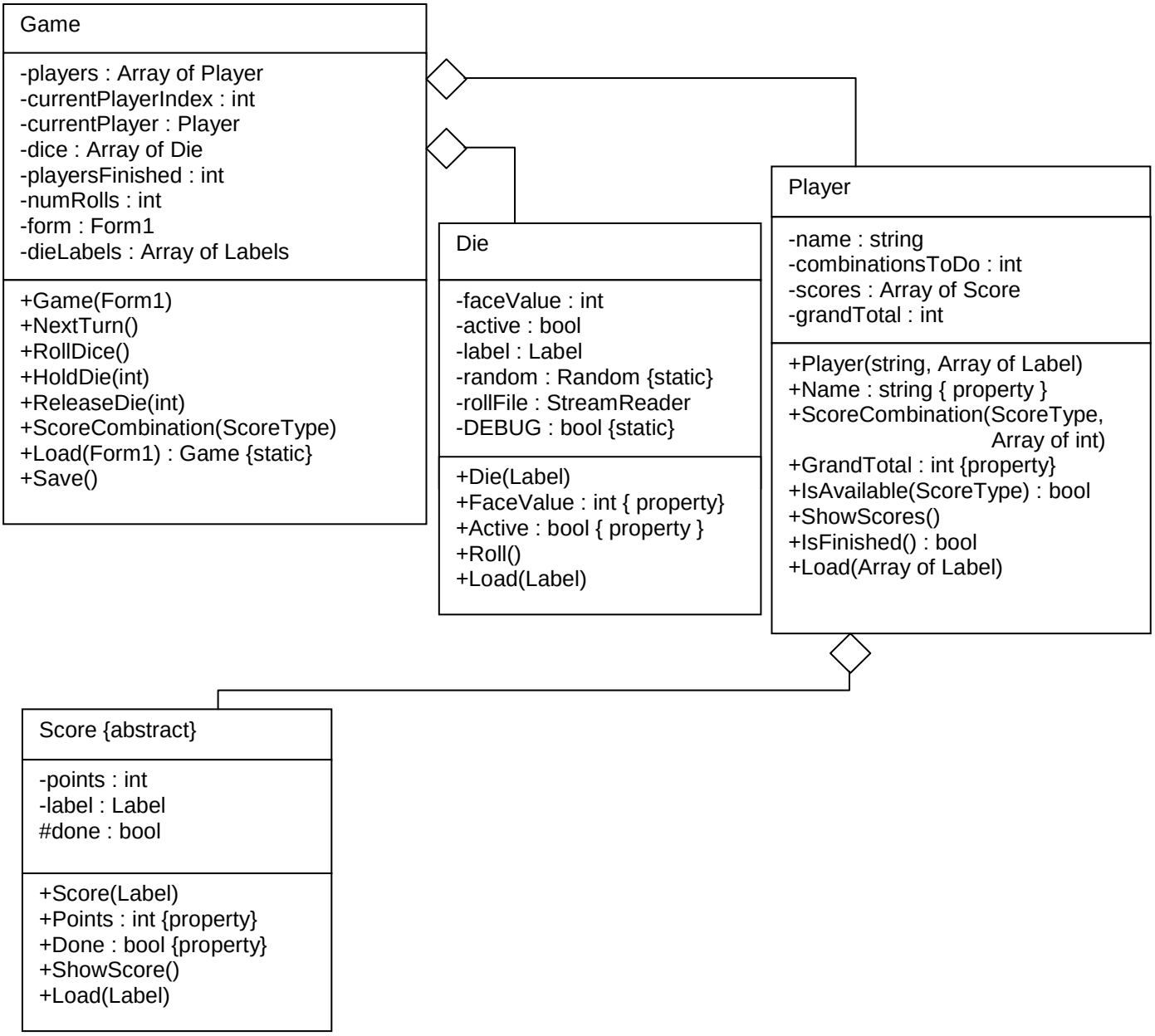
You will need to declare constants in some of these classes as well as private helper methods. But you MUST use this design including all specified instance variables and methods when implementing your project.

In these class diagrams, only the **type** of a parameter is named. You will need to supply the parameter's name which should be meaningful.

In any class, except for **Player class**, any method labelled with **{property}** can be implemented as **a C# Property** or as separate **mutator** and **accessor** methods**.**

However for **Player class, Name** and **GrandTotal** must be implemented as **C# Properties.** If you are unsure how to implement and/or use **a C# Property** ask a tutor or a PLF at STIMulate. **This is the only time where you can have someone else write code for you.**

A brief explanation of each of the four classes, their instance variables and methods follows after the diagram.



As advised in Part B, start by adding instance variables and method headings only for each and every class starting with **Score**, then **Player**, then **Die** and finally **Game**. Note an almost empty **Game class** already exists in your project, **Games.cs**.

Any class which has a **Label** object as an instance variable or as a parameter will need to add the following **using** directive: `using System.Windows.Forms;`

Do not attempt to write the bodies of any methods until a skeleton of each class exists and the project compiles without error.

Score class

An **abstract** class which represents a single scoring combination in a Yahtzee game. This class will have a number of subclasses defined in the next specification release.

Instance Variables

Each **Score** object has a number of **points** that have been scored for that scoring combination.

label is the **Label** object that represents this **Score** on the GUI.

done denotes whether or not this **Score** has been selected. This variable is **protected**, it is not **private** nor **public**. This allows access to the variable from any subclasses of **Score** but no other classes may access it.

Methods

Score(Label) is the class constructor.

Points is an accessor and mutator property for **points**. This method may be implemented either as standard mutator method, like **SetPoints(int)** and a standard accessor method, like **int GetPoints()** or as a **C# Property.**

Done is an accessor only property for **done**. It may be implemented either a standard accessor method or as a read only **C# Property**.

ShowScore will display the number of points on the associated label on the GUI, but only if this **Score** has been attempted, otherwise no points are displayed (***note:*** display nothing, do not display zero).

Load(Labels) will be explained later. For the moment leave the body of this method empty.

Player class

This class represents a player in the game.

Instance variables

name a player's name . The **name** will be shown on the GUI when it is this player's turn as the **Player label**. In the initial version of the game, the name of the players will be **"Player 1", "Player 2" etc.**

combinationsToDo keeps a count of how many scoring combinations a player has left to attempt before they are finished.

scores is an array of **Score** objects, where each element represents a scoring combination for this player.

grandTotal will be the Player's total score at the end of the game. During the game this will be the total score of the scoring combinations selected up to that point of the game.

Methods

The **Player** constructor has two parameters. The first is the **name** of this player; the second is the array of **Label** representing the score totals on the GUI. For the moment the constructor will just assign the name to the player. How to initialise the array **scores** with the appropriate **scoreTotal** of **Form1** will be discussed in the next specification release.

Name must be implemented as a **C# Property** for **name**.

ScoreCombination(…) will calculate the score for a specified scoring combination (the **ScoreType** parameter), given the five face values of the dice (the array of int parameter). This method will be empty until the subclasses of **Score** are implemented in the next specification.

GrandTotal is **C# Property** for **grandTota**l.

IsAvailable(ScoreType) checks whether or not this player has attempted the specified **ScoreType**. For them moment the body of this method can be `return true;`

ShowScores() will display all the scores for this player on their associated **ScoreTotals** labels. Leave the body of this method empty for the moment.

IsFinished() checks if this player has attempted all of the combinations.

Load(Array of Labels) will be explained later. For the moment leave the body of this method empty.

Die class

This class represents a die. You will need to add the following **using** directives:

```
using System.Windows.Forms;
using System.IO;
```

Note this class differs slightly from the Die Class used in lectures and workshop examples.

Instance variables

faceValue is the current number showing on the die (an integer between 1 and 6 inclusively).

active denotes whether the die is available to be rolled.

label is the Label object that represents the die in **`Form1.`**

random is a **Random** object used to generate the random numbers to simulate the rolling of a die. This object is instantiated in its declaration, **NOT** in the constructor as it is **static**.

rollFile is included for testing and marking purposes. Later specifications will explain the use of this variable as well as code snippets involving the variable which you will then insert into your code.

DEBUG is included for testing and will be used by the tutors to mark your project. You will be supplied with some code which uses this variable.

Methods

Die(Label) is a constructor. The parameter is the label that represents this die in **`Form1`**.

FaceValue is an accessor for **faceValue**. It may be implemented either a standard accessor method **GetFaceValue()** or as a read only **C# Property**.

Active is the accessor and mutator for the instance variable, **`active`**. A die can only be rolled if it is active. This method may be implemented either as standard mutator method, like **SetActive(bool)** and a standard accessor method, like **`bool` GetActive()** or as a **C# Property.**

Roll() simulates the rolling of this die.

Load(Label) use will be explained later. For the moment leave the body of this method empty.

Game class

This class represents a Yahtzee game (the model in terms of MVC terminology).

Instance variables

players is an array of **Players** of this Yahtzee game. Initially make the size of the array 2 in order to check that the game can be played correctly with this number of players.

currentPlayerIndex the index of the player whose turn it currently is.

currentPlayer is the **Player** object of the player whose turn it currently is.

dice is an array of **Die** representing the five dice in the game.

playersFinished keeps a track of how many of the players have finished attempting all of the combinations. It is used to determine that the current game is finished or not.

numRolls is the number of rolls that the current player has had in their current turn.

form is a link to **Form1** object. It is initialised by the constructor of **Game**. f**orm** is used to enable/disable various GUI controls as well as updating various labels of the GUI (updating the view in MVC terminology.)

dieLabels is an array of **Label** objects that represent the dice on the GUI. It is used in the initialisation of **dice** at the start of a game.

Methods

Game(Form1) is the constructor which will initialize all instance variables of **Game** in preparation for a game to begin in the correct state. [Recall from Part B that this constructor is called from **StartNewGame()** of **Form1.]**

NextTurn() updates **currentPlayer** and **currentPlayerIndex** to be the next player to play their turn, and then updates the GUI so that this player can start their turn.

The method involves setting the GUI to display the player's name and their scores (from previous turns) as well as enabling or disabling relevant buttons and labels.

RollDice() will roll all of the active dice and **numRolls** will need to be updated and if this is the first roll, then the score combinations that the player has not attempted yet should become available on the GUI. The message shown to the player via the **message label** will tell them what they need to do next. (***see page 4 of Part A for five possible messages***).

HoldDie(int) makes the die with the specified index (the parameter) inactive.

ReleaseDie(int) makes the die with the specified index active.

ScoreCombination(ScoreType) calculates the score for the specified scoring combination (the parameter) for the current player by calling **ScoreCombination(…)** of **Player**. So for the moment the body of this method will be left empty except for one statement until the subclasses of **Score** are implemented. The last action of this method will make an **OK button** appear on the GUI just below the **message label** and above the **Section label.** Add the single statement `form.ShowOkButton();` to this method.

Load(Form1) is a **static** method. For the moment leave the body empty except you will need to place the statement `return null;` in the method so that you code compiles. **You will be given the code for this method in a future specification.**

Save() will save the current game to a file. For the moment leave the body of this method empty. **You will be given the code for this method in a future specification.**

What to do now?

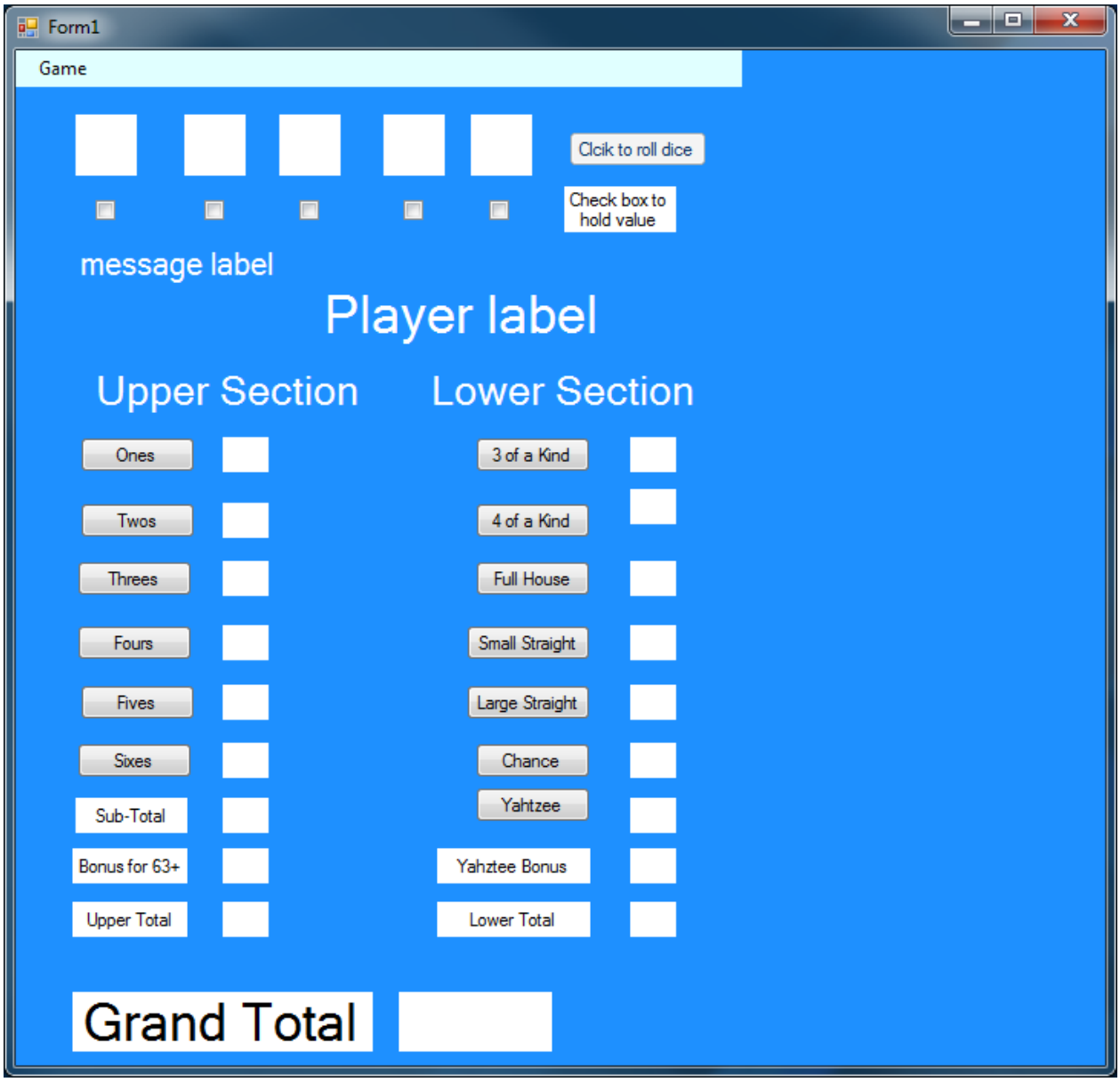
Having partially implemented the classes, **Form1**, **Game**, **Die**, **Player** and **Score** it is time to add the mechanism to begin playing a game, albeit without actual scoring. We will add a menu to our form with 3 options, **New**, **Load** and **Save**. Then we will progressively add event handlers in order for the game to proceed.

Open **Form1.cs** in Design view. Open the **Toolbox** and open the **Menus & Toolbars** section and place a **MenuStrip** onto the top left hand corner of **Form1**. Enter "**Game**" where the words "**Type Here**" are displayed. A submenu option will then appear, enter "**New** ", followed by "**Load**" and finally "**Save**", so that there are three menu options under "**Game**".

You may notice that the actual **MenuStrip** object appears in the **Component Tray** at the bottom of the design window. Click on this object to access the Properties Window for the **MenuStrip**. You may wish to change the **BackColor** though it is not a requirement to do so.

Open the Menu and double click on **New** which will create an event handler for that menu item. Add a call to **StartNewGame** method as the body of the event handler. Compile your code and then run your code to see something similar to this screenshot.

You may need to edit your code at various times to ensure that your program matches the description which follows.

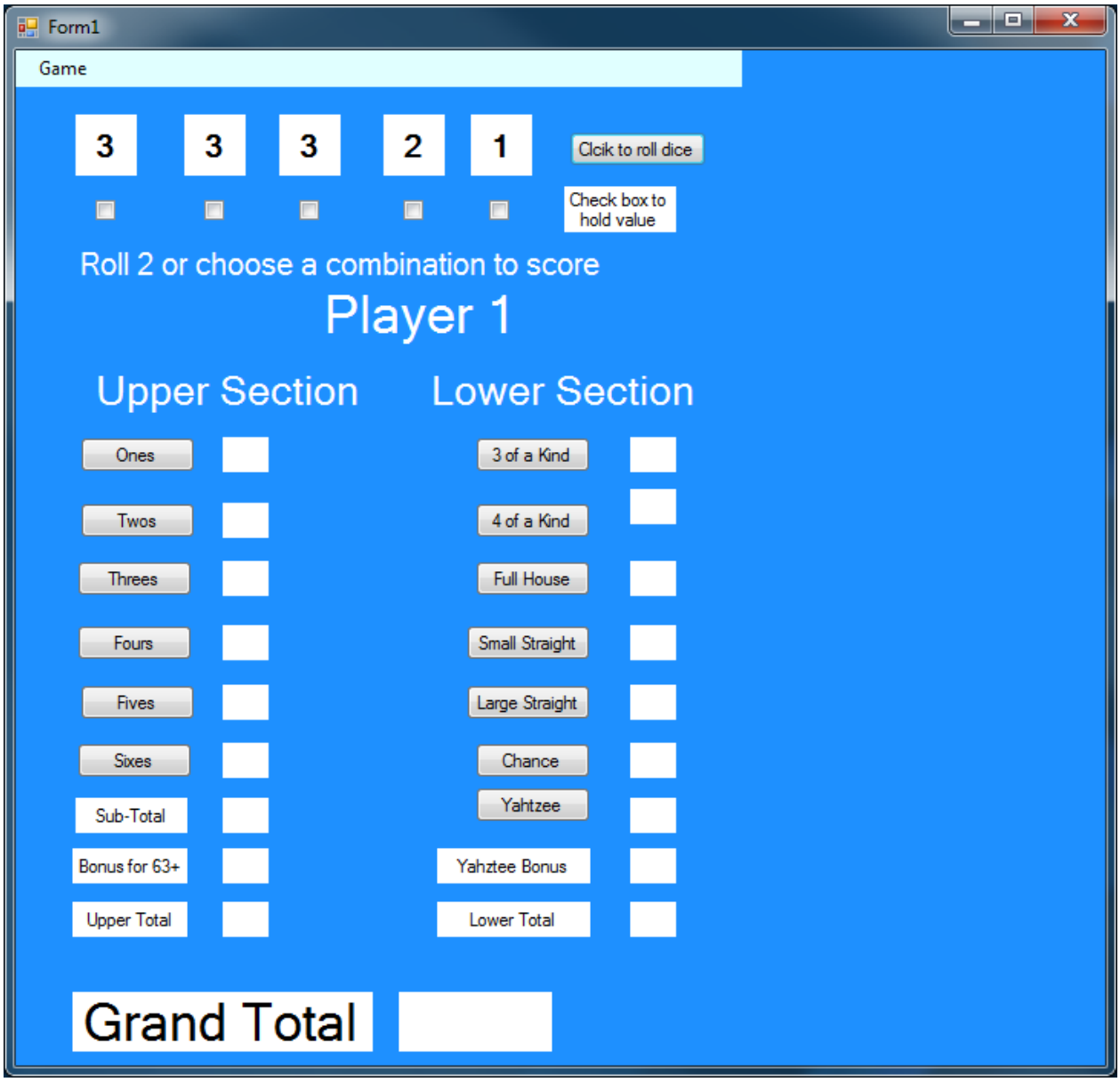


At this point, only the MenuStrip should be active, all buttons and checkboxes must be disabled.

After selecting **New**, the "**Click to roll dice**" button should be enabled and the **message label** should show **Roll 1** and the **Player label** should show **Player 1.**

Now to give the "roll dice" button an event handler. This event handler will simply call **RollDice** method of the **Game** class. Add the event handler, run you code, select **New** and click the roll dice button. If you have initialised everything correctly you should see something similar to the following screenshot albeit with different dice values.

You might need to change the **Properties** of the **dice labels** as well as other **controls** as you proceed to add additional functionality.



At this point, the checkboxes should be enabled. Now add event handler(s) for the checkboxes. This event handler will need to either call **HoldDie(int)** or **ReleaseDie(int)** of **Game** depending upon if the checkbox has just been **checked** or **unchecked**.

Also the thirteen **scoreButtons** should be enabled at this time. Each button requires an event handler which will call the **ScoreCombination(…)** of **Game.** Note that in a player's future turn not all of these buttons will be enabled as the player will have already played various scoring combination.

After a player has selected s scoring combination, the **OK button** appears along with an appropriate message telling the current player that their turn has finished and to click the **OK button.**

Your program should be at the stage where you can roll the dice, the message label is updated, check and uncheck various dice on any roll, select a scoring combination button and then repeat the process for a second player and then again for first player endlessly (or 13 times if you are counting how many scoring buttons a player has selected.)