

Event Organizer

1. Objectives

The main objectives of this assignment are to take a first step into the world of the object-oriented programming (OOP) by learning to work with one of the essential aspects of OOP, namely encapsulation. We will also practice data-hiding to complete the process of encapsulation. This assignment covers the following concepts:

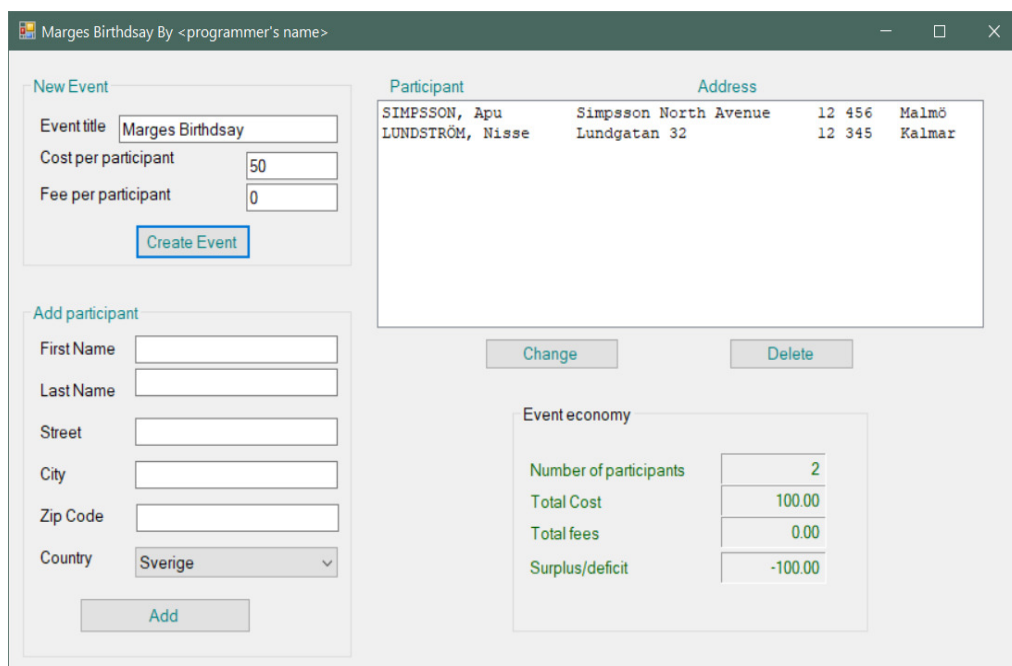
- Encapsulation
- Constructors
- “Has a” relation between objects
- Properties
- Array of objects

In this assignment, it is only possible to get a pass grade C. If you would like to receive a higher grade (A or B), you have to solve the alternative assignment available in the module. In that case, you don't have to continue with this one.

2. Description

In the previous assignment, you programmed an application that managed a single event such as a party. For the event, the application created a list of participants and stored first and last names for each participant.

In this assignment, we are going to develop the application further with some new ideas and manage the list of participants, using a class **Participant** for storing not only names but also the address of a participant. To manage a collection of **Participants** object, we are going to create another class **ParticipantManager**.



The screenshot shows a Windows application titled "Marges Birthdsay By <programmer's name>". The interface is divided into several sections:

- New Event:** Contains input fields for "Event title" (filled with "Marges Birthdsay"), "Cost per participant" (filled with "50"), and "Fee per participant" (filled with "0"). A "Create Event" button is below these fields.
- Add participant:** Contains input fields for "First Name", "Last Name", "Street", "City", "Zip Code", and a "Country" dropdown menu (set to "Sverige"). An "Add" button is at the bottom.
- Participant Table:** A table with columns "Participant" and "Address". It contains two rows:

Participant	Address
SIMPSON, Apu	Simpson North Avenue 12 456 Malmö
LUNDSTRÖM, Nisse	Lundgatan 32 12 345 Kalmar
- Buttons:** "Change" and "Delete" buttons are located below the participant table.
- Event economy:** A summary box showing:

Number of participants	2
Total Cost	100.00
Total fees	0.00
Surplus/deficit	-100.00

Figure 1: A run-time example

- 2.1 We use a, a collection List <T> for storing participant objects, and therefore, the application does not have to know the maximum number of participants to create the list. Collection of this type grow and shrink in size as items are added and removed. Figure 1 shows an example of the application execution.
- 2.2 For each participant, the application should store the address of a participant in addition to first name and last name.
- 2.3 The address consists of information for Street, City, Zip Code and Country. Use string data type for the first three items and an enum for countries. The enum (Countries), is available for downloading on the Assignment page on Canvas.

Functionalities and Features:

- 2.4 The application should provide the following functionalities:
 - Create a new event with a title, cost and fees per person
 - Add a new participant with name and address
 - Change data for an existing participant
 - Remove an existing participant
- 2.5 Every participant has:
 - A first name and a last name
 - An address, street, zip code, city and country.
- 2.6 At program start, the group box for reading participant information should be disabled until the user has given a title and pressed the button Create Event. If the user does not provide values for cost and fee per person, use 0.0 as default value, but the title should be given.
- 2.7 When the Add Button is pressed, prepare an object of the class Participant using the data inputted by the user, and save it in the list of participants.
- 2.8 For the change and delete buttons, the user must highlight an item in the list box.
- 2.9 Display a list of the registered participants in a ListBox. Show names and details of the participant's address as shown in Figure 1. Information about the number of registered participants and cost calculations are also displayed on the GUI. The GUI is kept updated as data is added, changed or deleted.

Note: By saving or storing data it is meant that data is stored into variables in the program and the data exists temporarily in the computer's memory. The data will not be available to the program after the program terminates. On a later occasion when we talk about saving data to files, we will learn how to save the contents of the registry permanently into files.

3. The Project

Figure 2 shows the classes that can be identified for the problem, the class diagram below shows the classes and the association between the classes.

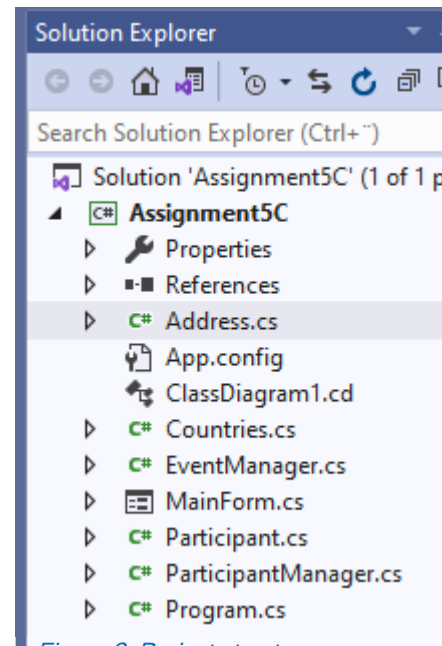


Figure 2: Project structure

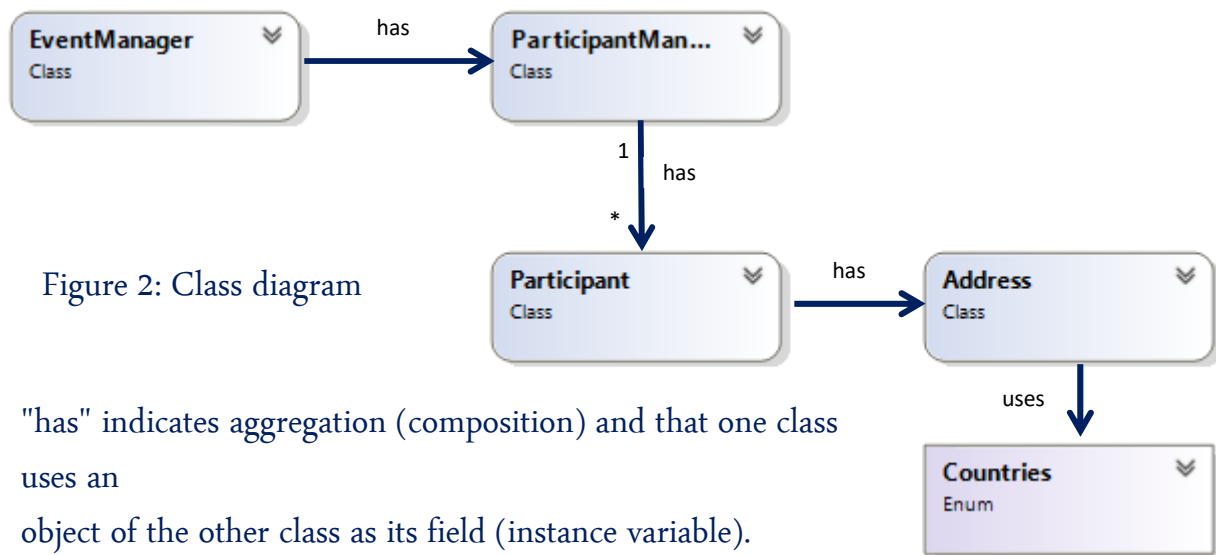


Figure 2: Class diagram

"has" indicates aggregation (composition) and that one class uses an

object of the other class as its field (instance variable).

EventManager uses an object of **ParticipantManger**

ParticipantManager has a collection of **Participant** objects
(*indicates many)

Participant has an object of **Address**

Address uses the enum **Countries**.

4. To Do

You can begin with sketching your GUI form or begin with writing the classes that are to store and process data. Use proper names for the controls (GUI components) that you use to design your interface before starting to program the code-behind the GUI.

Assuming that you are ready for writing code, it is important to structure the solution by first thinking about which classes are necessary to write. Begin with identifying good classes. In the above GUI example, the user is required to input values for the event and values for each participant of the event. For these you, you should get an idea of some of the classes like **Event**, **Participant**, **Address**, etc.. However, for this assignment you can use the above class diagram.

Get started with writing the classes

When you start to create classes, begin with the class that is least dependent on other classes. In our case, the **Address** class is the least dependent class; it is actually an independent class. The next class to write would be **Participant** that uses **Address** and then **ParticipantManager** that uses **Participant**. When you are done with these classes, you can create the class **EventManager** that uses the **ParticipantManager**. Finally, code in the **MainForm** that creates and uses an object of the **EventManager**.

Note: You don't have to write all the methods that are listed in the class-diagrams that are given with the class descriptions. Write those you need or use your own methods. When you can't figure out the purpose of method, neglect it and write your own methods.

General requirements:

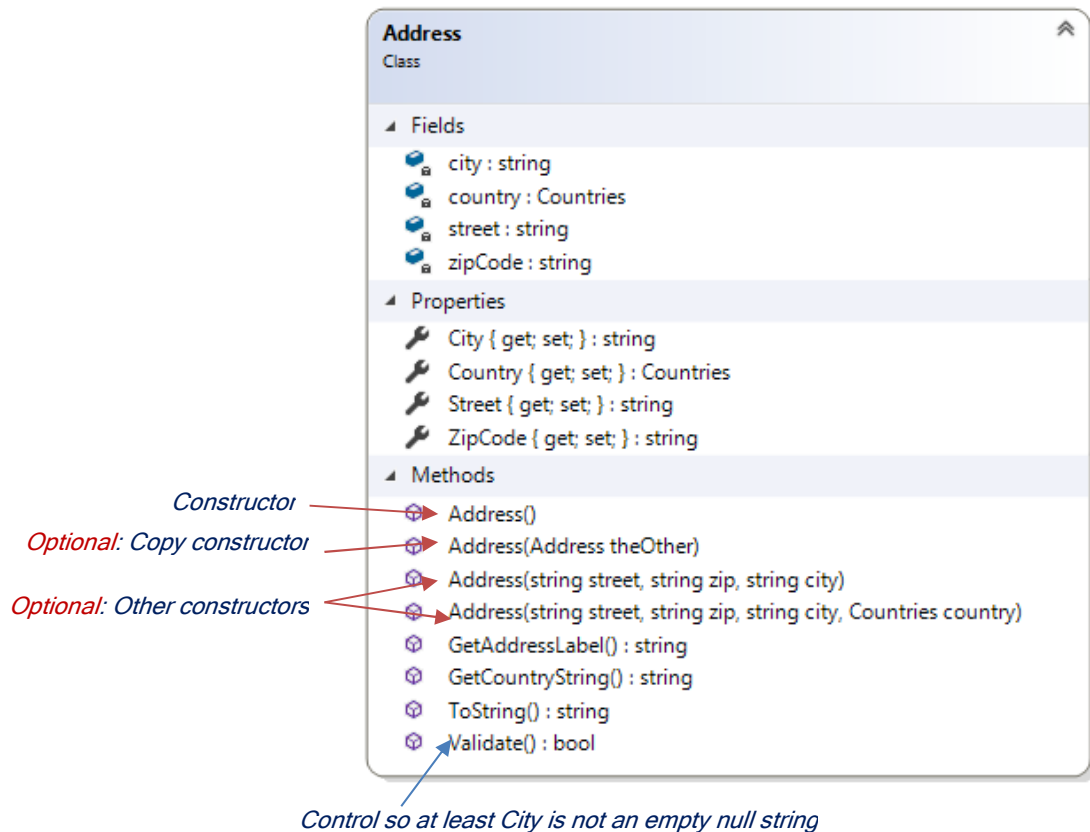
- 4.1 All class fields are to be declared private. Use of public instance variables is strictly forbidden and will cause resubmission. To provide access to the private fields of a class, use properties.
- 4.2 All classes and methods of each class are to be documented using the xml-documentation comments. The comments should have appropriate and valuable information to add clarification to your code.
- 4.3 Write your name as the programmer on the title of the Form.

In the sections that follow, classes listed in the Project figure (Figure 2) are described in more details.

Note: The items marked as **Optional**, are not mandatory but they are recommended for learning more.

5. The Address class

- 5.1 Create a new class and save it as **Address.cs**. Define fields and write properties connected to the fields.



The method `Validate()` should check so the City (which is a string) is given i.e. it is not an empty or null string.

You can test all textual data (strings) using the code below:

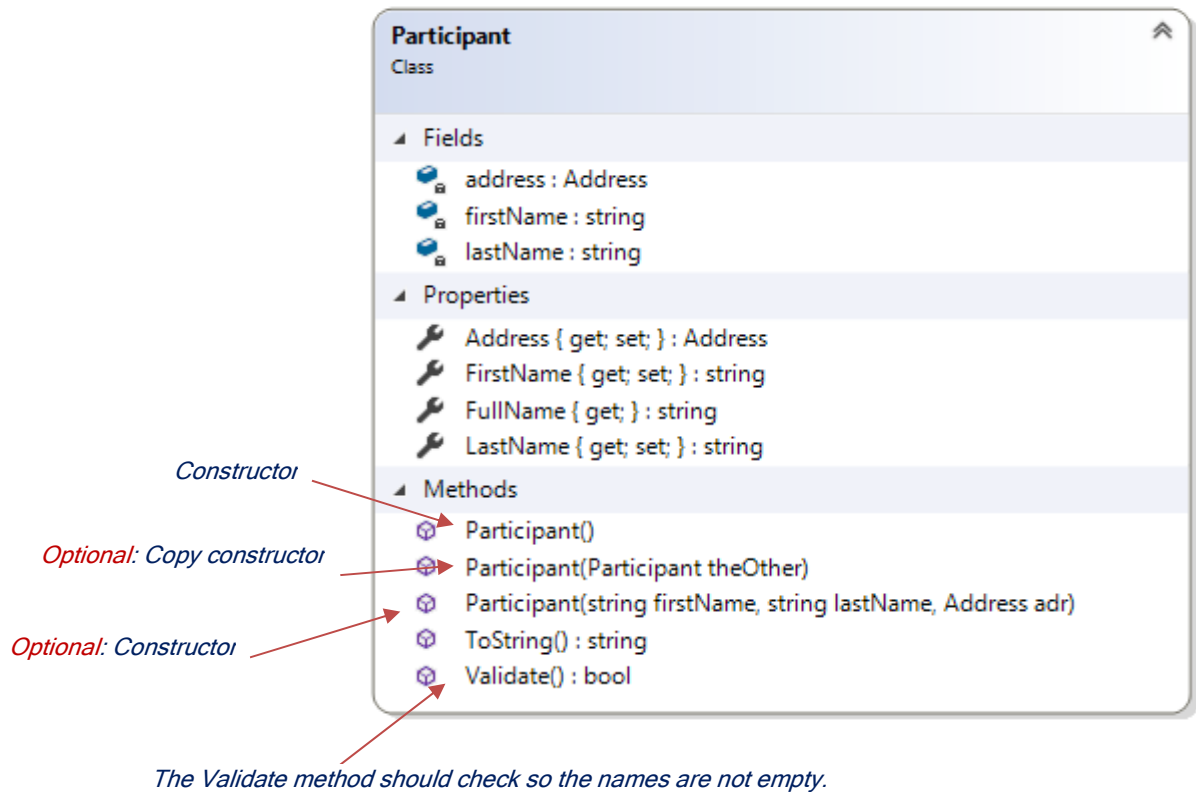
```
bool ok = !string.IsNullOrEmpty(textToValidate);
```

Optional: When displaying the names of countries, replace the '_' char in countries like `United_States_Of_America` to a blank space so it can be read as "United States of America".

Optional: The Address class should have three constructors. These should chain-call each other; do not write same code in the constructors. Chain-calling implies that a constructor with less number of parameters call one that has more parameters. As an example a constructor with 2 parameters should call the constructor with 3 parameters passing the two parameters and a default value for the third one. If there no such constructor, call the next one with 4 parameters and so forth.

6. The Participant class

6.1 Create a class **Participant** with Address as its field.



7. The ParticipantManager class

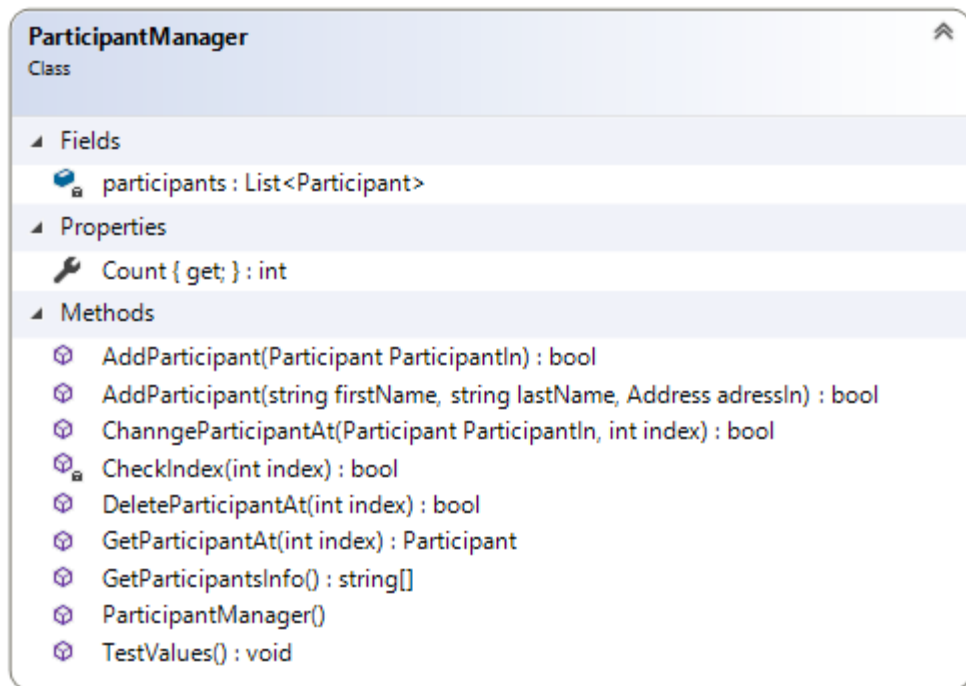
The responsibility of this class is to maintain and handle the collection of participants, add a new participant object, edit or delete an existing object, return a participant at a certain position and return a list of strings representing each participant.

This class serves as a container class for objects of Participants. Use a `List<Participant>` to store the participants added via the user interface.

```
class ParticipantManager
{
    //declar an object of the collection
    private List<Participant> participants;

    public ParticipantManager()
    {
        participants = new List<Participant>();
        //TestValues();
    }
}
```

The class diagram below can be used as a template.



The method **GetParticipantsInfo** returns an array of strings where each string is the return value of a call to the participant object's `ToString()`. The `ToString` method in the `Participant` class can in turn call the `ToString` method in the `Address` class.

For storing a participant's data, we are going to use a dynamic array. Dynamic arrays are known as collections. Collections are a big subject and as it is both too early and too much to talk all about them at this time, we will only concentrate on the very useful and perhaps the most used collection called the **List<T>**.

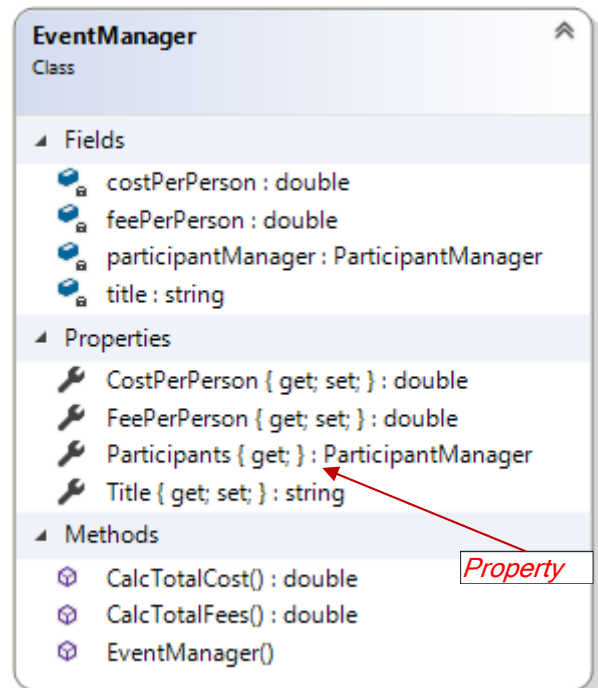
The collection `List<T>` is generic which means it works for all kinds of objects. The collection is recommended to be used instead of its non-generic counterpart **ArrayList** that was used a lot before (and it still can be used).

7.1. Control must be done in your code so that the list (registry) is not indexed out of range (make sure the index, when trying to access an element in the list is ≥ 0 and $<$ the length of the list, the method `CheckIndex` in the above figure).

8. The EventManager class

This class is not a container class although it might sound so. A class name "Event" would sound better but Event is a class in .NET Framework and we should avoid using the same name.

EventManager's area of responsibility is to handle the event itself. As can be seen from the class diagram, it uses an object of the **ParticipantManager** (through aggregation) as its field. MainForm should use an object of the **EventManager** to save not only the event data such as the title, cost and fee but also save participants' data sent by **MainForm** in the **participantManager**.



In the class diagram, a property returning the **ParticipantManager** is included. This is necessary for the **MainForm** to be able to get access to the methods of the **ParticipantManager**.

9. The MainForm class

Most of the functionalities should be familiar to your from the previous assignment.

9.1 **MainForm** should only have one instance variable and that is an object of **EventManager**. Declare no more instance variables.

```
public partial class MainForm : Form
{
    EventManager eventManager;

    public MainForm()
    {
        InitializeComponent();
        InitializeGUI();
    }
}
```

Create an event. (Figure 1)

9.2 Create (or recreate if already created) the instance variable **eventManager**. Read the title, cost and fee given by the user, and store the data in the **eventManager** calling the corresponding Property.

Add a new participant

- 9.3 Create a new Participant object. Read and save input in the object, validate input and store the object in the list of participants. Call the **Participants** property provided in the **EventManager** object:

```
eventManager.Participants.AddParticipant(participant);
```

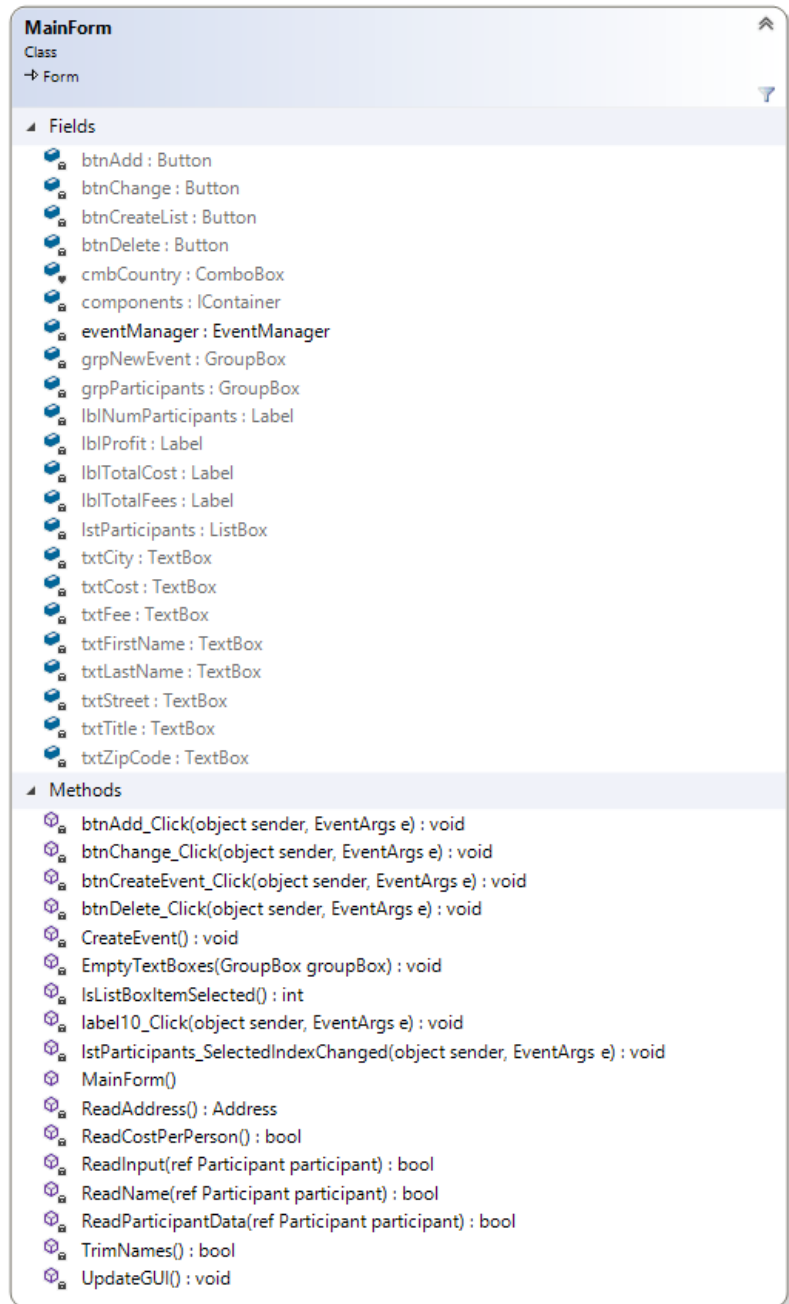
9.4 **Optional: Edit an existing participant:**

- The user selects a participant on the ListBox, containing all participants. If no participant is highlighted, a message is given and no further processing is to be done.
- Using the index, acquire the object from the participant manager (eventManager.Participants.GetParticipantAt(..))
- Display the selected participants's data in the input controls.
- When the user clicks on the Edit button, update the list of participants in participant manager and update also GUI.
-

9.5 **Delete an existing participant:**

- The user selects a participant on the ListBox, containing all participants. If no participant is highlighted, a message is given and no further processing is to be done.
- Using the index, call the participant manager's Remove (or Delete) method to remove the object at position index in the list of the participants. Update GUI accordingly.

The class diagram for the MainForm is also provided below in case it can be of help for you as far as naming of components are concerned and also get an idea of methods handling functionalities.



10. Submission, help and guidance

Compress all files and sub-folders, from both Part I and this part, into a Zip or Rar file and upload it as before. A detailed description of how to program different sections of this part of the assignment is available as a separate document on the Module. The document is only to provide some type of a help. You may of course complete the assignment without the help document.

Good Luck!

Farid Naisan,
Course Responsible and Instructor