The Control Tower – Delegates and Events

1 Objectives

The main objectives of this assignment are:

- To work with events and delegates using a publisher/subscriber pattern,
- To create a WPF application and become acquainted with some basic WPF components.

2 Description

Delegates are mainly used in event handling but also for callbacks, i.e. when a method is to call one or more other methods. A practical use case is a publisher-subscriber model in which an object publishes one or more events at certain times and the objects who have subscribed to an event receives and handles the messages. Senders are called **publishers** and receiver objects are called **subscribers**.

A publisher is an object that offers services to other objects. Those objects that request to use the services of the publisher are the subscribers. Every publisher object can publish one or several events, to which other objects can subscribe. An object can be both a publisher of events and a subscriber to events published by other objects. When an object subscribes to an event, the object will be notified by the publisher, whenever the event is fired.

The Delegate type is a powerful construct in .NET, yet very object-oriented, and will definitely come to use in many situations and together with Events, can be used to produce advanced features. Use the forum to get ideas and more help. Most of the more advanced topics in .NET require that you have a good understanding of delegates and events. They are used among other things in event-driven programming (GUI) in Windows Forms and WPF.

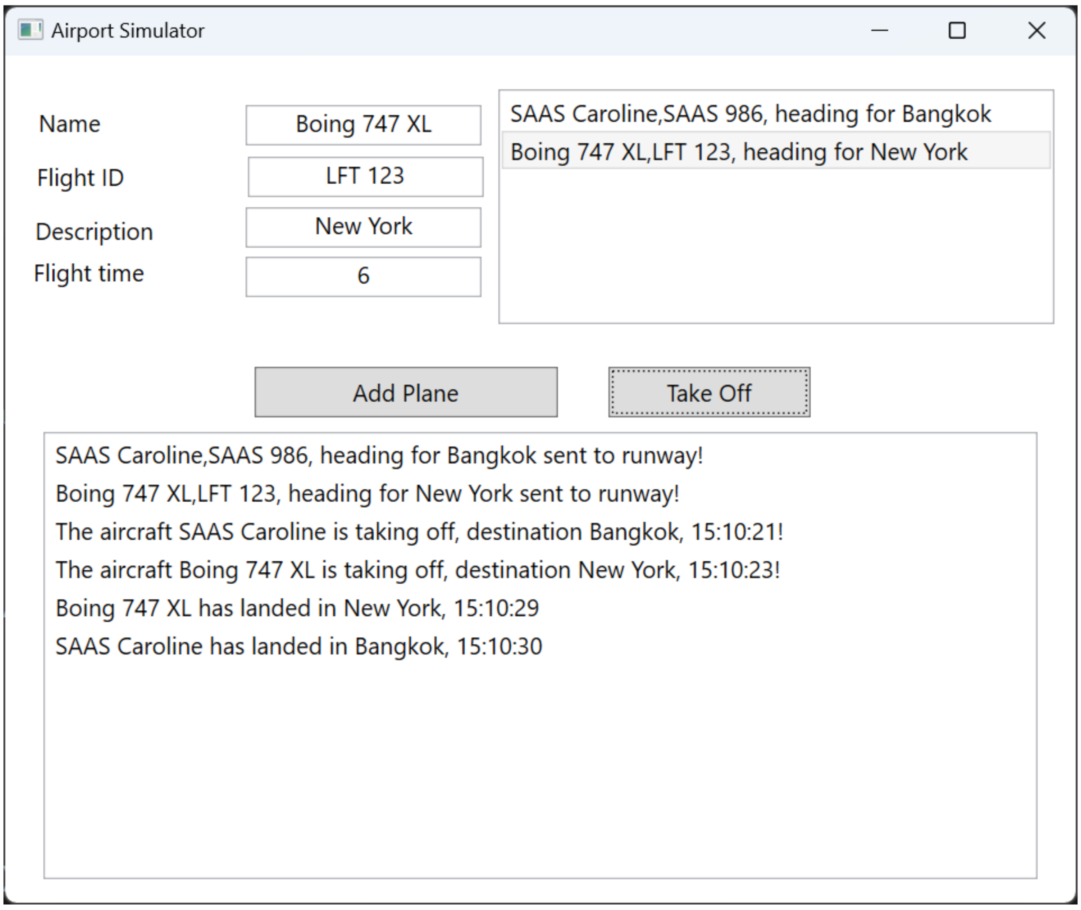
In this assignment, we are going to simulate a simple use case of flight departures in an airport. The airport has a control tower. A controller registers a plane to get ready on the runway and to take off using a software with a UI that may be designed as illustrated in the figure below. The top ListBox lists the planes added in a list of airplanes handled by the Control Tower while the lower ListBox is used to display the flight status and other information. The user selects an airplane in the top ListBox before clicking the Take Off button.

After taking off, the airplane (monitored by its pilot, of course) notifies the same control tower when taking off is complete and when the plane has landed, as shown in the above sample.

Create a Presentation Foundation (WPF) or a MAUI project. Although the description in this document assumes that you are using a WPF project. In case you select to work with MAUI, you can ignore the requirements and features that are not applicable in MAUI, and this will not affect your grade.

Note: An **airplane** represents a **flight** and therefore both terms are equivalent.

You can design the UI and solve the problem in your way, but the solution needs to be based on the requirements outlined below.



Name: Boing 747 XL
Flight ID: LFT 123
Description: New York
Flight time: 6

SAAS Caroline,SAAS 986, heading for Bangkok
Boing 747 XL,LFT 123, heading for New York

```
SAAS Caroline,SAAS 986, heading for Bangkok sent to runway!
Boing 747 XL,LFT 123, heading for New York sent to runway!
The aircraft SAAS Caroline is taking off, destination Bangkok, 15:10:21!
The aircraft Boing 747 XL is taking off, destination New York, 15:10:23!
Boing 747 XL has landed in New York, 15:10:29
SAAS Caroline has landed in Bangkok, 15:10:30
```

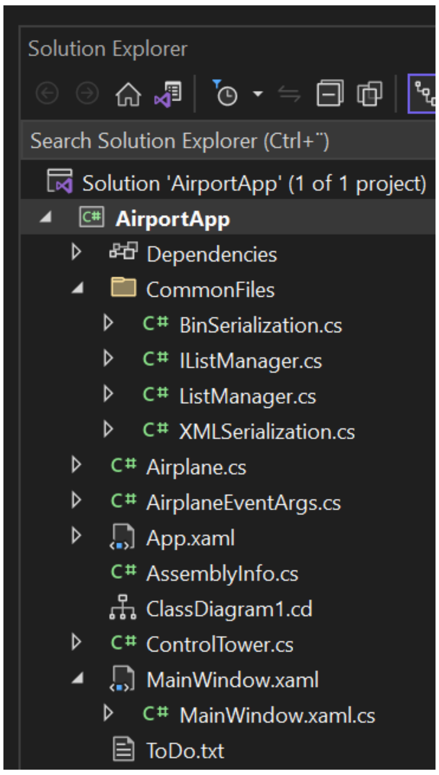
3 Requirements for a pass grades (C)

3.1 Create a **MainWindow** class to handle the user interface. This class is to display and handle only the input and output. It uses an object of the **ControlTower,** which is responsible for all air traffic.

3.2 Define a class for the **ControlTower** and a class for describing an **Airplane** class with properties shown in the class diagram below.

3.3 Use at least two event delegates in the Airplane class to notify and handle taking off and landing events.

3.4 Define a class **AirplaneEventArgs** deriving from **EventArgs**, for the events, **TakeOff**, and **Landed**. This class can handle both of the events, but you may create a class for each event if you find it more suitable.



```
Solution 'AirportApp' (1 of 1 project)
AirportApp
  Dependencies
  CommonFiles
    BinSerialization.cs
    IListManager.cs
    ListManager.cs
    XMLSerialization.cs
  Airplane.cs
  AirplaneEventArgs.cs
  App.xaml
  AssemblyInfo.cs
  ClassDiagram1.cd
  ControlTower.cs
  MainWindow.xaml
    MainWindow.xaml.cs
  ToDo.txt
```

3.5 se a timer to simulate the flight time. One second representing one hour is a suggested scale. The timer in WPF is called **DispatcherTimer**. Start the timer when the Take-Off button is clicked. The airplane must stop its timer when it has landed.

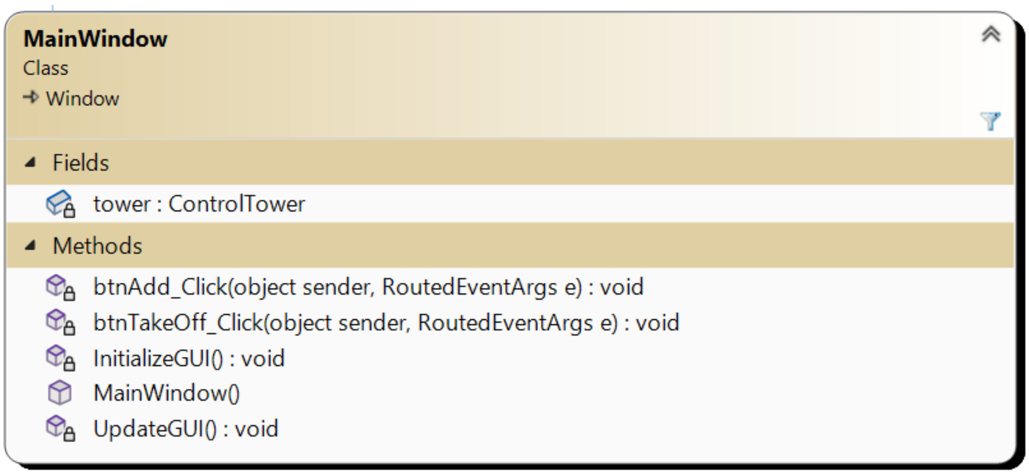
3.6 he user should not be able to order take off to an airplane that has not landed. When a plane has landed it should change its destination to “Home”. The plane can take off again when the controller orders take-off by clicking the Take-off button.

3.7 is allowed to use auto-properties in this assignment, but you are not allowed to use the keyword “**var**” for declaring variables!

3.8 Optional: you can copy the ListManager classes (see the above Solution Explorer image) from your previous assignment to this project and reuse them with List collections.

3.9 The class diagrams presented below do not have to be followed but can be used as a guideline. You can add new methods, change methods, and use you own structure.

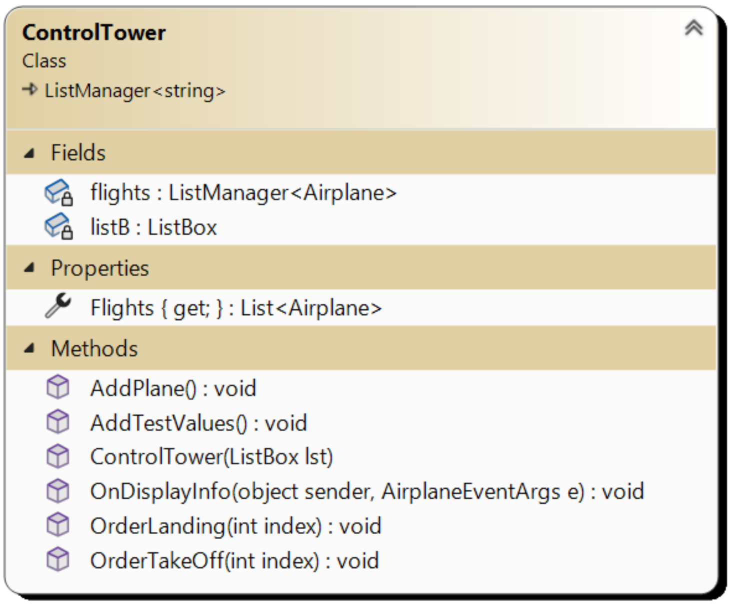
3.10 The class diagrams presented below do not have to be followed but can be used as a guideline. You can add new methods, change methods, and use you own structure.



MainWindow
Class
→ Window

Fields
- tower : ControlTower

Methods
- btnAdd_Click(object sender, RoutedEventArgs e) : void
- btnTakeOff_Click(object sender, RoutedEventArgs e) : void
- InitializeGUI() : void
- MainWindow()
- UpdateGUI() : void

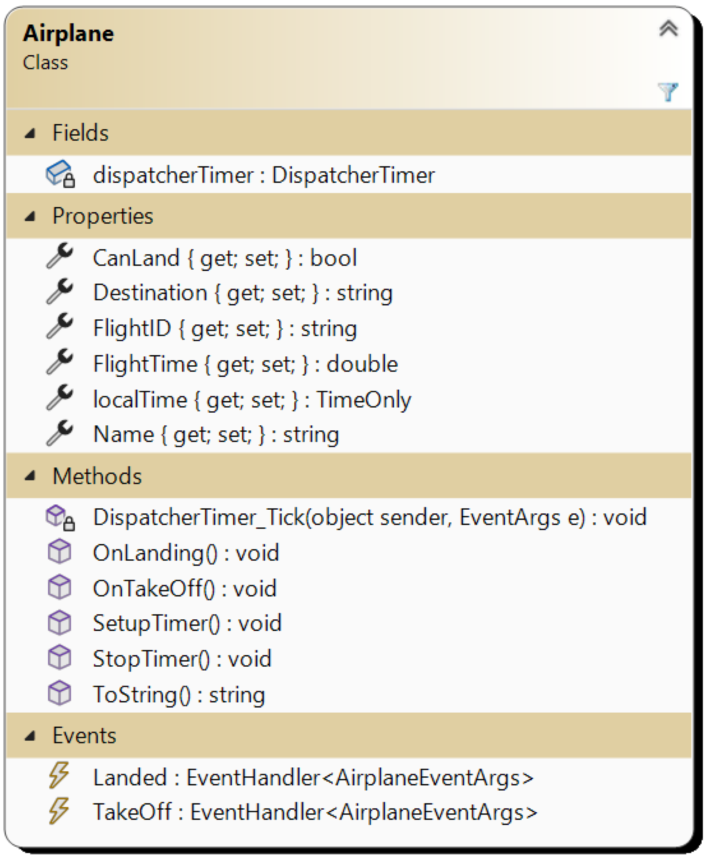


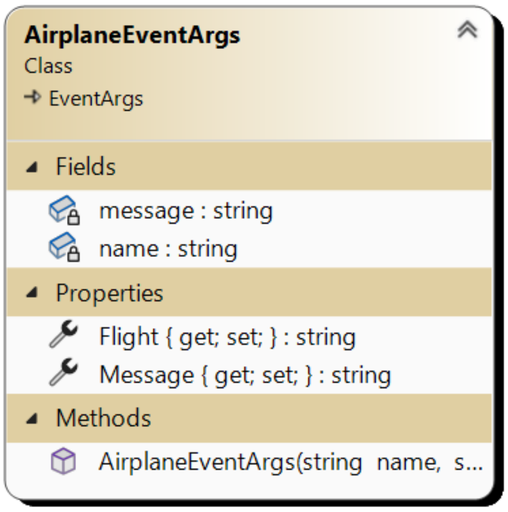
ControlTower
Class
→ ListManager<string>

Fields
- flights : ListManager<Airplane>
- listB : ListBox

Properties
- Flights { get; } : List<Airplane>

Methods
- AddPlane() : void
- AddTestValues() : void
- ControlTower(ListBox lst)
- OnDisplayInfo(object sender, AirplaneEventArgs e) : void
- OrderLanding(int index) : void
- OrderTakeOff(int index) : void


Airplane
Class
Fields
dispatcherTimer : DispatcherTimer
Properties
CanLand { get; set; } : bool
Destination { get; set; } : string
FlightID { get; set; } : string
FlightTime { get; set; } : double
localTime { get; set; } : TimeOnly
Name { get; set; } : string
Methods
DispatcherTimer_Tick(object sender, EventArgs e) : void
OnLanding() : void
OnTakeOff() : void
SetupTimer() : void
StopTimer() : void
ToString() : string
Events
Landed : EventHandler<AirplaneEventArgs>
TakeOff : EventHandler<AirplaneEventArgs>


AirplaneEventArgs
Class
EventArgs
Fields
message : string
name : string
Properties
Flight { get; set; } : string
Message { get; set; } : string
Methods
AirplaneEventArgs(string name, s...

4 Requirements for higher grades (A, B)

The following features are required only for the grades A and B (in addition to the above requirements). You can skip this section, if you are aiming only at a Pass grade as described above.

For Grade B (and A):

4.1 Add a feature that allows the Control Tower to order an airplane in the air to change altitude by a numerical value and return its new altitude. To accomplish this task, you must use a normal delegate..

For Grade A only:

4.2 Use a WPF ListView for displaying the Airplane information. You can neglect this requirement if you are using MAUI and you will still receive an A grade, depending on the structure and quality of your work.

5 Submission

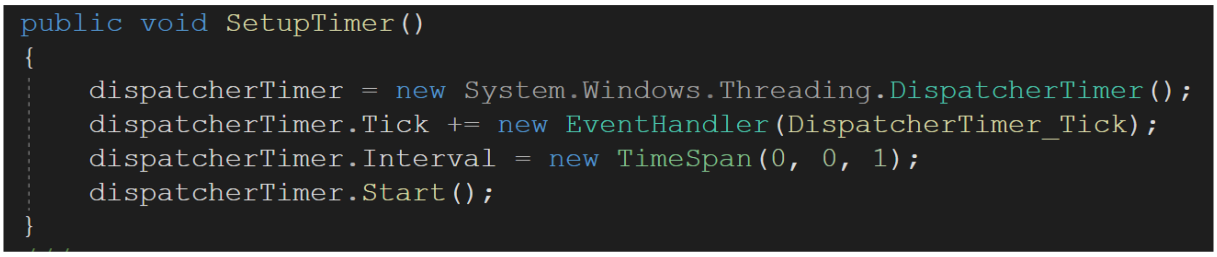
Test the system thoroughly to ensure it works as expected before you submit your project. Upload your solution to Canvas as before.

6 Help

6.1 Airplane class:

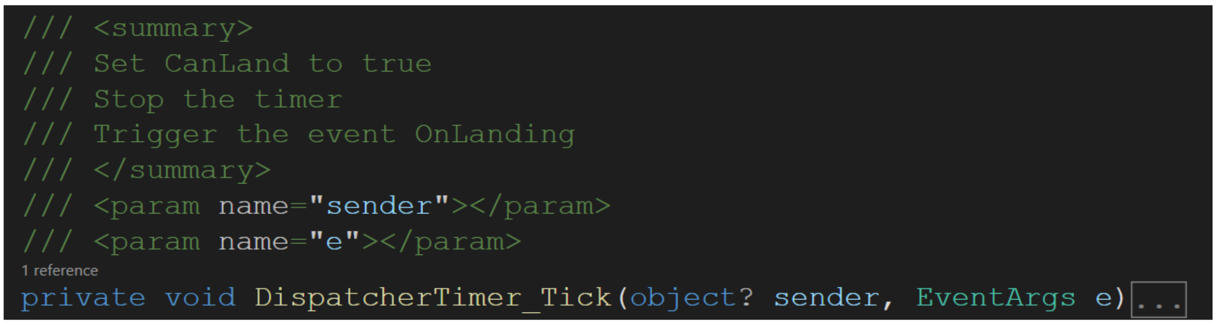
```
private DispatcherTimer dispatcherTimer;
```

6.2 Write the following method to set up the timer. The method is to be called by the Control Tower when the Take Off feature is launched.



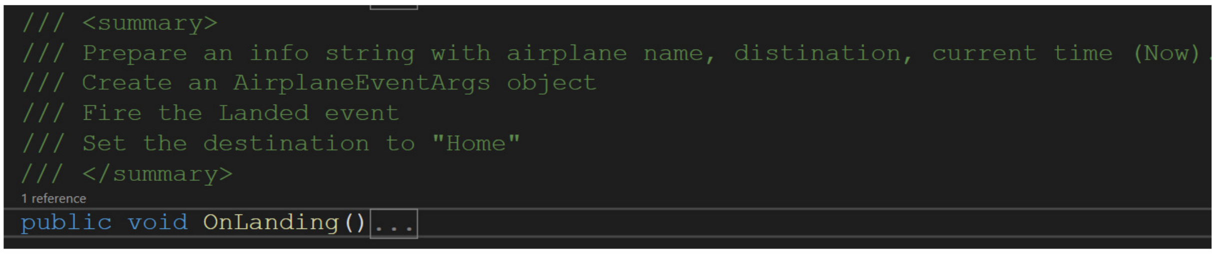
```
public void SetupTimer()
{
    dispatcherTimer = new System.Windows.Threading.DispatcherTimer();
    dispatcherTimer.Tick += new EventHandler(DispatcherTimer_Tick);
    dispatcherTimer.Interval = new TimeSpan(0, 0, 1);
    dispatcherTimer.Start();
}
```

6.3 The method **DispatcherTimer_Tick**:



```
/// <summary>
/// Set CanLand to true
/// Stop the timer
/// Trigger the event OnLanding
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void DispatcherTimer_Tick(object? sender, EventArgs e)...
```

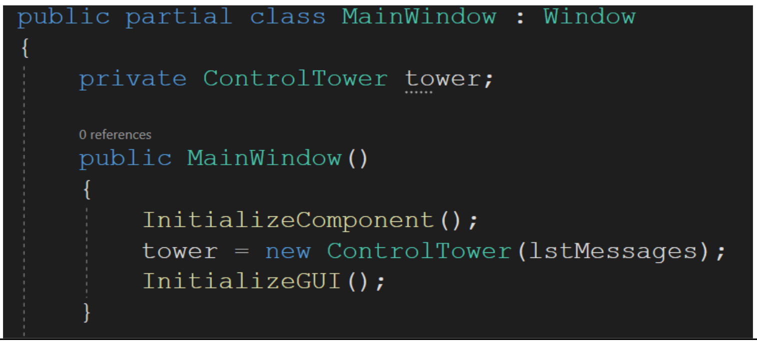
6.4 Hints on how to code the **OnLanding** method:



```
/// <summary>
/// Prepare an info string with airplane name, distination, current time (Now)
/// Create an AirplaneEventArgs object
/// Fire the Landed event
/// Set the destination to "Home"
/// </summary>
1 reference
public void OnLanding()...
```

6.5 **ControlTower updating the messages listbox:**

To simplify this issue, the MainWindow can send the reference of the ListBox to the ControlTower class when creating an object of the class, as shown below:



```
public partial class MainWindow : Window
{
    private ControlTower tower;

    0 references
    public MainWindow()
    {
        InitializeComponent();
        tower = new ControlTower(lstMessages);
        InitializeGUI();
    }
```

6.6 Study the BidHouse Exercise in the module for both delegates and WPF hints.

There is an example project, the **BidHouse** exercise, in the module that should give you a good idea. Inside the code files, it is documented which steps the publisher and the subscriber classes should take. Take a good time and try to understand the code. This assignment is actually easier than the mentioned exercise

Using delegates might seem to be difficult, and may take a while to understand when it is new to you, but once you practice with it and learn to determine the roles of the publisher and the subscribers, you will see that the mechanism makes sense.

Think of the role of the ball in a football game. When the ball is kicked, it is the ball object that gives out (publishes) the signal (event) of being kicked without knowing (or needing to know) which other objects are waiting to receive the signal. There are several objects, the referee, the players, and the goalkeeper, who would like to be notified when the ball has been kicked. These are the subscribers. Now consider the two tasks that need to be accomplished in the code: which actions are to be coded in the Ball class to fire the events and how you should code in each of the subscriber classes to get notified when an event is fired by the publisher object. These tasks are well documented in the code example. **BidHouse**.

Good Luck!

Farid Naisan,
Course Responsible and Instructor