Name: Jamie Penner Date: 11/1/19

Lab 4 – Observer Pattern

Objectives

Part 1 – Setting up the Project

Part 2 – Using the Observer Pattern

Background/Scenario

The observer pattern lets objects be informed when something happens elsewhere. Traditionally, observers are implemented using interfaces—with the observer registering an interest with a subject. In this lab, you will start with a “Gamma” style code base and then refactor the code to use C# delegates\events.

In this lab you will work with an interface implementation of the observer pattern and then refactor to a C# delegate/event implementation of the observer pattern. The lab is a 'Console Clock' application. Using the Framework Class Library (FCL) Console APIs, various clocks are displayed in the console window. These clocks display time at various intervals ranging from once every hundredth of a second to once every hour. (Note: not all clocks are actually implemented.)

Required Resources

- Visual Studio 2017

Part 1: Setting up the Project

Step 1: Download the open the ObserverLabStarterCode project and familiarize yourself with the GammaConsoleClockObserver project.

The clocks are implemented as observers. They implement an ITimerObserver interface and are registered with the Ticker class. The Ticker class runs on its own thread and calls the observers. You will refactor this code to use delegates and events for observer registration and notification

Run the GammaConsoleClockObserver application to see the clocks in action. You will see three clocks in the top-left corner of the console. The first one is updated each second, the second clock every tenth of a second, and the third clock every hundredth of a second.

Step 2: Make a copy of the GammaConsoleClockObserver project and name it CSharpConsoleClockObserver.

This new project is where you will be working. Close all the files that are currently open in Visual Studio and make sure you only edit the files from the CSharpConsoleClockObserver application.

You will only edit the newly created project so you can easily flip back and forth between both the old and new projects when debugging to verify you have the correct output.

Part 2: Using the Observer Pattern

Next, we will update the Ticker class.

Step 1: Creating the delegates.

In the Ticker.cs file, add three global public delegate class types—call them OnSecondsDelegate, OnTenthsDelegate, and OnHundredthsDelegate. Note: The signatures must match the methods you need to invoke in your clock classes.

For example:

```
public delegate void OnTenthsDelegate();
```

Step 2: Update the Ticker class

Add 3 delegate instance variables, one for each delegate type. Call these instances OnTenthsTick, OnHundredthsTick, and OnSecondsTick. Use the “event” key word to wrap the delegates for safer use in our Observer pattern implementation.

Delete the timers list and the code that uses it inside the Run method.

Remove the RegisterTimer and UnRegisterTimer methods.

Add a NullHandler() method. Add a Ticker constructor and initialize the Ticker

delegates to the address of the NullHandler() method for thread safety around subscribe and unsubscribe events.

Inside the Run method, add code to invoke the delegates.

Step 3: Refactor each of the Derived Clock classes

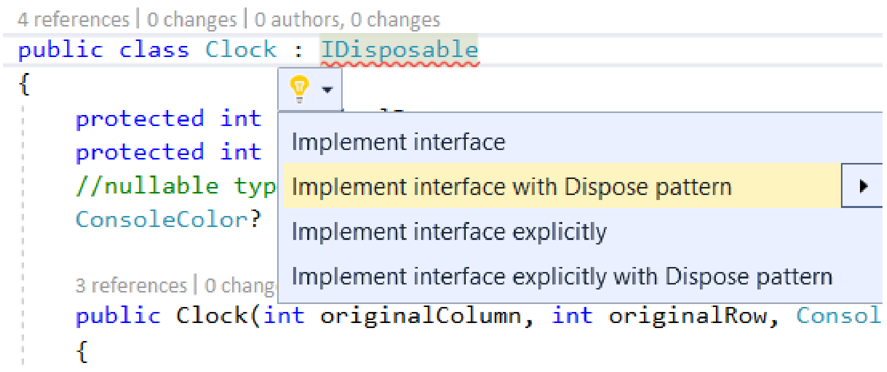
Add a Ticker parameter to the constructor. In the constructor add the appropriate method addresses to the Ticker's delegate invocation lists.

Add a private data member of type Ticker to use in the Dispose method. Initialize this private data member in the constructor.

Remove the ITimerObserver inheritance. Also, remove any methods that will not be used. Change each of the remaining methods to private access since they will only be invoked by the delegate invocation list.

Implement the IDisposable interface to remove the appropriate method(s) from the Ticker's delegate invocation lists.

Note: The Dispose pattern must be implemented in the base class as well. Look for "Implement interface with Dispose pattern."



```
4 references | 0 changes | 0 authors, 0 changes
public class Clock : IDisposable
{
    protected int
    protected int
    //nullable typ
    ConsoleColor?

    3 references | 0 chang
    public Clock(int originalColumn, int originalRow, Consol
    {
```

Implement interface
Implement interface with Dispose pattern
Implement interface explicitly
Implement interface explicitly with Dispose pattern

Step 4: Update Main

In Main, refactor the code to create an instance of Ticker. Then create an instance of each clock passing your Ticker instance as a parameter. The threading code should not be modified. Once this works, implement "using" statements to fire the Dispose() methods of each clock.

```
using (SecondClock clock1 = new SecondClock(0, 1, ConsoleColor.Yellow,
ticker))
```

Step 5: Clean up the project

Remove any files the project no longer uses.

Reflection

1. What does the "event" keyword do?

   The event keyword is an object oriented method used to subscribe observers to objects.

2. What does the "using" statement do? (Not the using statements at the top, but the ones around your clock instantiation)

   The using statement calls the dispose function to delete the clocks before exiting.

3. Why did we set the Ticker event to the NullHandler function? What would happen if we didn't set it? What design pattern is this?

   If there are no subscribers, the program could crash or throw an exception. This is called null object pattern.

4. In almost every C# event tutorial, you will see checking the event if it is null first, but in the lab, we set it to NullHandler. Both achieve the same purpose, but one is better in this case over the other. Which is better and why is that?

   Setting the ticker event to nullhandler is better. This option allows the program to iterate without a subscriber and follows the solid principle.