


## GUI Observer - Using the Observer pattern to handle data updates to complex linked controls

By [ChrisLee1](#)

Use the Observer pattern to encapsulate control update code (adding items to lists etc.) in separate classes outside of the form, and handle it when the code is called.

7 votes for this article. 

Popularity: 3.38. Rating: 4 out of 5.

 [Download example project - 30.3 Kb](#)

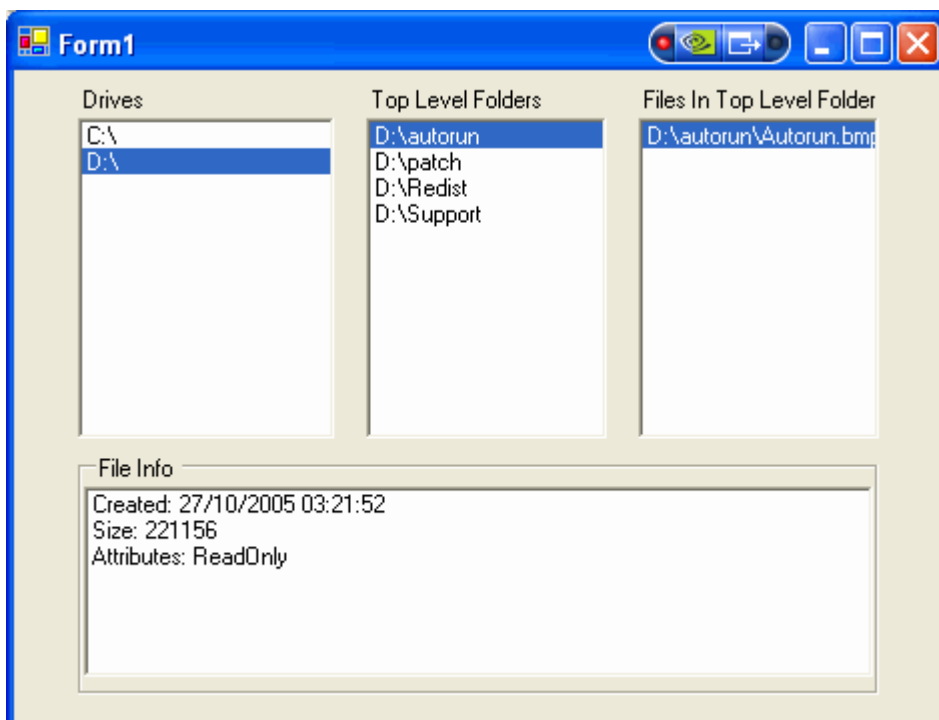
### Introduction

I searched for a long time to find an existing pattern/process/solution on the web to a long standing issue I have.

When writing a WinForms application (or VB6 or below, for that matter), the IDE encourages you to place a lot of code and logic in (or triggered from) control events. This is not (in my experience and opinion) a good solution, events can often be triggered for reasons the programmer didn't initially anticipate and lead to all sorts of problems. Also, it doesn't feel very object-oriented having lots of methods in forms that don't actually having anything to do with the form itself (being rather for controls on the form). Logic for the data relationships is often found in the control event or population code, when it is nearly always modeling an existing data relationship of some sort.

Let me explain further, with a trivial example application.

Imagine a simple little application designed to allow some very basic browsing of your top level hard disk directories. The main form for this application might look something like this.



The *Drives* list is initialized on startup, *Top Level Folders* populates when a Drive is selected, *Files in Top Level Folder* populates when a *Top Level Folder* is selected, and *File Info* is populated when a *File* is selected.

A set of common sense business rules for this screen might look like this:

*Drive* list:

- Filled at startup, nothing selected by default.
- When a *Drive* is selected, the *Top Level Folders* is changed to represent the folders on that *Drive*.
- No *Top Level Folder* will be selected by default, and the *Files in Top Level Folder* and *File Info* areas should be cleared.

*Top Level Folder* list

- Filled when the *Drive* list changes to contain the *Top Level Folders* on that *Drive*.
- No *Top Level Folder* will be selected by default, and the *Files in Top Level Folder* and *File Info* areas should be cleared.

*Files in Top Level Folder* list

- Filled when the *Top Level Folder* selection changes to contain the files in that selected folder.
- No file will be selected by default, and *File Info* should be cleared.

*File Info*

- Filled when the *Files in Top Level Folder* list changes to contain info on the selected file.

A traditional approach, in a WinForms type application, would probably look something like the code in *Traditional.vb* in the example project supplied, where code in the `SelectedIndexChanged` events of the three list boxes would be used to trigger the population code for the other lists.

```
Private Sub lstDrives_SelectedIndexChanged(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles lstDrives.SelectedIndexChanged
    populateFolders()
End Sub

Private Sub populateFolders()
    Dim folders() As String
    Dim folder As String
    Dim drive As String

    If (lstDrives.SelectedIndex >= 0) Then
        drive = lstDrives.SelectedItem
        folders = Directory.GetDirectories(drive)
        ' Initialise the drives box only
        lstFolders.Items.Clear()
        For Each folder In folders
            lstFolders.Items.Add(folder)
        Next
    End If
    populateFiles()
End Sub
```

This approach doesn't scale very well as the form, controls, and the relationships become more numerous and complex. The main issues for me have always been:

- As it stands, the population code must sit in the form itself as it requires access to values in controls sitting on the form.
- The logic for the relationship between the Drives, Top Level Folders, and Files is found across different control events and population code. Whilst in this example, where the relationship basics and fundamentals (folders and file) are never likely to change, the events and population code are probably not the best place for the relationship rules to be enforced. This means that if any of the

selected indexes are changed programmatically, without triggering the `SelectedIndexChanged` events (for example, when `SuspendLayout()` has been called), **nothing** will be updated, requiring more calls to the 'populate' methods from other places in the code, reproducing logic already found in other pieces of code.

A better approach perhaps, would be if each control knew what data it represented, and somehow received a notification of when that data changed. Furthermore, it would be useful if as part of that notification, the data that changed was also supplied. With this information, a control could typically repopulate itself. Additionally, if the control is populating itself, it can also decide whether something is selected by default or not, and if it does, that it should also be able to inform, any other control that uses this information, which item has been selected (or that no item has been selected, as the case maybe).

If all of that was true, then we would simply need to update the underlying data the controls watch, in anyway we want or need, and all the controls that use that data either directly or indirectly will be updated.

I effectively used those last two paragraphs as my specification for the "update" part of the GUI I've been working on, the GUIObserver is the result, and the example application supplied is the test bed for the ideas.

The example contains two projects: an Observer pattern library (CPObserver, written in C#), and the example project (written in VB.NET). I nearly always do my libraries in C# and my WinForms in VB.NET, just because I enjoy working in both languages and it keeps me fresh in both. I shall convert both examples, should demand require it.

## Background

The article is based around the Observer pattern. To summarize this pattern:

A Subject (a piece of data/information) allows Observers (something that needs to know when a Subject changes) to attach themselves to it so it can notify them when it changes. A single Subject can have many Observers, and a single Observer may monitor many Subjects.

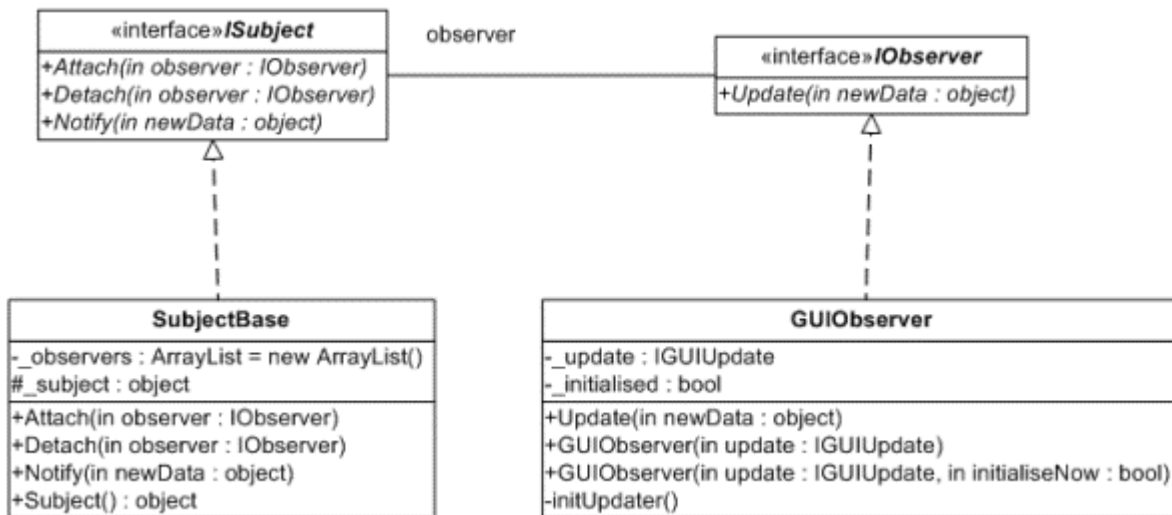
For more detailed information on the Observer pattern, try the following links:

- [Observer Design Pattern in C# and VB.NET](#)
- [Implementing Observer in .NET](#)
- [Observer Pattern](#)

The GUIObserver pattern was developed for an application I'm developing that contains many different related views of large amounts of related data, with several different types of visual relationships between the data, depending on how it's being viewed. The GUIObserver is meant to fulfill one distinct part of an application's architecture: updating control contents. The supplied example project was developed to prototype ideas before committing them to the main application (which has now been done with great success).

## Observer Pattern

The classic Observer pattern doesn't quite support my requirements as it doesn't implement any method of passing information back with the notification of change. As I needed to send some data with the notification, I simply modified the Observer pattern to allow the Subject to send a single piece of data (as a `System.Object`) when it notifies Observers. Additionally, my concrete Observer (`GUIObserver`) uses a generalised `GUIUpdate` object that contains the code to physically update controls. The `IGUIUpdate` interface and its implementation will be discussed below.



Note: I did originally have the notifications sending instances of `ISubject` around, which seemed to make complete sense, but I've since changed it to `System.Object` for flexibility.

With our new Observer pattern implemented (see `Observer.cs` in the CPPatterns library example project), we can now create some simple Subjects that we can Observe. There is also the `SubjectBase` class that handles the storage of the actual subject content (as a `System.Object`) and some further derived classes to create basic data type subjects, i.e., `StringSubject`, `LongSubject`, `BoolSubject` etc.

We have three subjects that require watching, and they are all strings, so we set up three `StringSubjects` to represent the data we want to observe.

```

Private _selectedDrive As New StringSubject("")
Private _selectedFolder As New StringSubject("")
Private _selectedFile As New StringSubject("")
  
```

Now, we can modify the `SelectedIndexChanged` events in our `ListBoxes` to use our new Subjects instead of calling population methods. For the *Top Level Folders* list, the event code would now read:

```

Private Sub lstFolders_SelectedIndexChanged(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles lstFolders.SelectedIndexChanged
    _selectedFolder.Subject = lstFolders.SelectedItem
End Sub
  
```

The same can be applied to other `ListBoxes`' events, and we have now successfully detached the user's interactions with the GUI from any direct logic other than storing the user's selection.

Using this pattern, we could now extend a control by having it implement the `IObserver` interface and fill the control on the Notify implementation, but we can generalize the case further to allow us to use the new Observer pattern on any control, group of controls, form, or any object.

We accomplish this by defining an interface that contains all the members required by a control to update itself. This interface could then be implemented on objects that a generic observer could use. The interface is called `IGUIUpdate`, and the generic observer is `GUIObserver`.

## IGUIUpdate

«interface» <b>IGUIUpdate</b>
<b>+ControlSubject()</b> : <i>SubjectBase</i> <b>+NewData()</b> : <i>object</i> <b>+Init()</b> <b>+Reset()</b> <b>+Fill()</b>

**IGUIUpdate** defines the members needed for a control to update itself, these include:

- **Init()** which initializes the control as required (called only once).
- **Reset()** which resets the control back to its initialized state.
- **Fill()** which populates the control.
- **ControlSubject**, a property that may optionally contain a Subject instance that can be updated to reflect the selection of a control.
- **NewData**, a property that may optionally contain the new value of the Subject that caused the notification.

A Concrete/Base class, **GUIUpdateBase**, is available which offers basic implementation of the properties. Naturally, the methods are abstract as they are specific to the control being updated.

With this interface and base class defined, we can now develop a simple class to handle a specific control. The code shown is from the *GUIObserver.vb* form in the example project, and handles the *Top Level Folder* listbox in the example.

```
Private Class FoldersGUIUpdate
    Inherits GUIUpdateBase
    Private _list As ListBox
    Public Sub New(ByVal listBox As ListBox, _
        ByVal controlSubject As SubjectBase)
        Me.ControlSubject = controlSubject
        _list = listBox
    End Sub
    Public Overrides Sub Fill()
        Dim folders() As String
        Dim folder As String
        Dim drive As String

        drive = Me.NewData
        If Not IsNothing(drive) And Directory.Exists(drive) Then
            folders = Directory.GetDirectories(drive)
            ' Initilise the drives box only
            _list.SuspendLayout()
            For Each folder In folders
                _list.Items.Add(folder)
            Next
            _list.ResumeLayout()
            ControlSubject.Subject = ""
        End If
    End Sub

    Public Overrides Sub Init()
        _list.BackColor = Color.Yellow
    End Sub

    Public Overrides Sub Reset()
        _list.Items.Clear()
    End Sub
End Class
```

As you can see, we inherit from the **GUIUpdateBase** and implement the three methods (**Init**, **Reset**, and **Fill**) required.

In our constructor, we store a pointer to the listbox we will update, and a pointer to a subject this listbox selection represents.

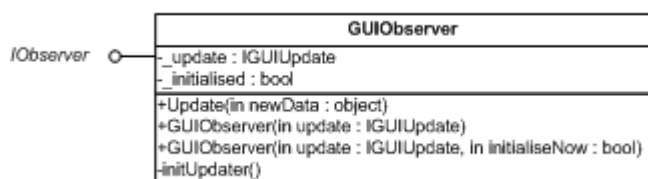
The `Init()` method is not that useful for `ListBoxes`, but I find it useful for more complex controls like grids and `ListView`s that may require some further runtime set-up (such as column/row headers etc.) before being populated.

The `Reset()` method simply returns the list to its initialized state.

The `Fill()` method is responsible for populating the control. In our example, the control needs a list of the folders at the root level of a specified drive. The specified drive is available in the `Me.NewData` reference because it's the `SelectedDrive` subject that this control is linked to. You can also see in the example that after repopulating the list, it uses the `ControlSubject.Subject = ""` line to ensure everything else knows that no folder is selected.

When the `GUIObserver` receives a notification, it first calls `Init()` if it hasn't already been called, followed by `Reset()`, and finally `Fill()`.

## GUIObserver



We now have the code to physically update the control in our `GUIUpdate` object, now we need to wrap it in an Observer so it can be called from a Subject.

The `GUIObserver` class implements our custom `IObserver` interface, and is initialized with an object that implements `IGUIUpdate` so that when `Observer.Update` is called, it can call the `IGUIUpdate`'s `Init()`, `Reset()`, and `Fill()` methods as required. It can also set the `NewData` property to the value supplied in the notification so that `IGUIUpdate` methods can access the new value that caused them to be triggered.

Using the `GUIObserver`, we can now link our Subjects to the instances of generalized observers (`GUIObserver`) that contain their own references to objects that implement the `IGUIUpdate`. The code below comes from the `GUIObserver.vb` example, and shows the initialization of the `GUIObserver` and `IGUIUpdate` objects and their attachment to the subjects we created earlier.

```

Dim obDrive As GUIObserver = _
    New GUIObserver(New DrivesGUIUpdate(lstDrives, _selectedDrive))
Dim obFolder As GUIObserver = _
    New GUIObserver(New FoldersGUIUpdate(lstFolders, _selectedFolder))
Dim obFile As GUIObserver = _
    New GUIObserver(New FilesGUIUpdate(lstFiles, _selectedFile))
Dim obFileView As GUIObserver = _
    New GUIObserver(New FileInfoGUIUpdate(lstInfo))

_selectedDrive.Attach(obFolder)
_selectedFolder.Attach(obFile)
_selectedFile.Attach(obFileView)
  
```

Now that everything is in place, the runtime process flow (after initialization) is as follows:

- User selects a Drive from the list.
- The `SelectedIndexChanged` event fires and sets the `_selectedDrive` subject to the new value.
- The `_selectedDrive` subject notifies any Observers attached to it.
- The `FoldersGUIUpdate` instance created and attached to the Drive subject (`_selectedDrive`) is accessed and its methods called, it refreshes the folder list according to the Drive supplied in the notification via the `NewData` property.
- After refreshing the list, no items are selected, so it sets its `ControlSubject` (which is the `_selectedFolder` subject supplied at construction) to `Nothing`.

- The `_selectedFolder` notifies its observers (the `FilesGUIUpdate` class) and a similar process ensues.

## Conclusions

Over the years, I have used several approaches to implement this part of an application's architecture, and my current one feels like the best :). Seriously though, I looked around for a pattern/process etc. to fulfill this particular niche, and (Model-View-Controller and Command patterns aside) was surprised I couldn't find anything (please point me in the direction of anything I've missed).

I have implemented this pattern in a much more substantial application with some quite complex visual relationships as well as used it for some simpler screens, and I have noticed the following positive things:

- My form code is much cleaner and easier to follow, mainly because it's not cluttered up with loads of code to populate controls on the form.
- Logic and code for populating individual controls is no longer sitting in a form itself, and can be placed in other classes/namespaces, allowing me to be much more flexible with my code organization.
- I can modify and switch whichever implementation I want to use for populating individual controls cleanly and easily, by implementing new `GUIUpdate` objects. I can also switch the implementations at run time by removing and attaching different `GUIObserver` instances that contain different `IGUIUpdate` objects. This can allow controls (and screen space) to be reused with great ease.
- Logic for the visual relationships between data is mostly modeled in one area, the attachment of the Observers to Subjects.
- It can easily be extended to cater for very specific cases. For example, I have a `GUIUpdate` object that is registered with an observer that contains three private `GUIUpdate` objects of its own, for different types of output in a `ListView`. The main `GUIUpdate` object, when it receives a `Notify`, checks the data that was supplied, and based on this, it can identify which of the private `GUIUpdate` objects to use.
- `GUIUpdate` objects need not be responsible for just one control, it's completely up to the developer to define the constructor and contents of a `GUIUpdate` object. I have one that needs eight `Labels`, and out of pure laziness, I simply pass a pointer to the form in to get at them rather than the eight `Labels` individually.

Nothing is perfect though, and the following are still on my list to look in to:

- There is no inherent logic in the pattern to know whether an update is truly required (i.e., is the control actually visible), in my main app. Many of the controls are on tabs, and therefore, not always truly visible, but they are always updated. This can mean a slow update time when selecting certain (high level in my data relationships) items, as this triggers lots of child data control updates. The response time to show individual tabs is naturally quick but changing high level items can be slow. There is currently no inherent way in the design to offer a JIT mechanism for updates to offer quicker response times. Even if we do work out whether our specific control is actually visible, there is no way to delay the update call until it is visible. A possible solution (just thought of while writing this) is to simply have `SelectedTab` subjects that have to be set as an additional "key" before a control updates itself.
- The data relationships are not really "modeled", although the visual ones are, and centrally located more or less. I still feel there could be improvements to this part of the implementation. Ideas?
- I'm still not completely happy with the way external data is supplied to the `GUIUpdate` classes. Currently, it is supplied via the `IGUIUpdate.NewData` property which is set to the new value of the Subject that notifies the control. For simple data, this is fine, but when a control requires more than one piece of data (i.e., two database keys), then it must watch more than one Subject and cater for the fact that it might not have all pieces of data at once and therefore remember pieces that come in until it has a complete set. Without adding some form of call back request using delegates/events so that a `GUIUpdate` object can request data from its client (i.e., the form), which I don't want because events are optional, there is no enforcement of supply of data at compile time level. I couldn't think of another way to supply the objects with external data, ideas?

## History


- v1.0 - 31/05/2006 - Submission to CodeProject.

## ChrisLee1

Click [here](#) to view ChrisLee1's online profile.



## Discussions and Feedback

 **15 comments** have been posted for this article. Visit <http://www.codeproject.com/dotnet/GUIObserver.asp> to post and view comments on this article.

[All Topics](#), [.NET](#), [C#](#) >> [.NET](#) >> [Patterns and Practices](#)  
Updated: 31 May 2006 12:25

Article content copyright ChrisLee1, 2006  
everything else Copyright © [CodeProject](#), 1999-2006.