(untagged)

Design Patterns Implementation in a Storage Explorer Application

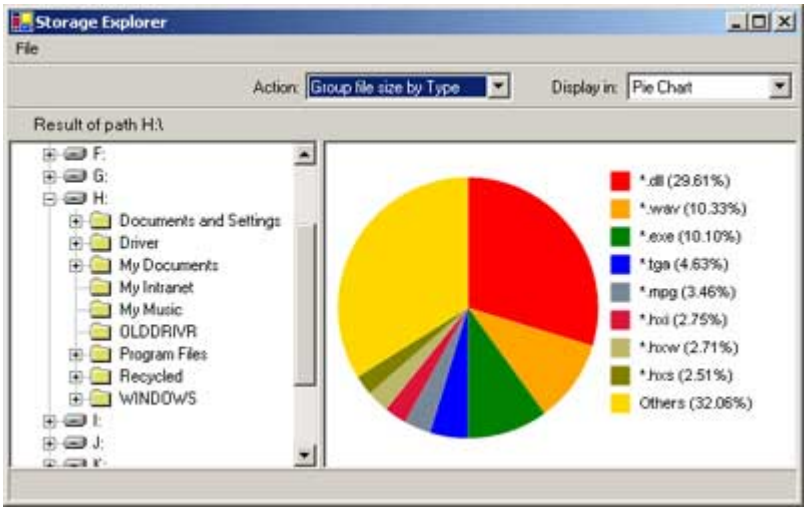
Breman Sinaga

25 Apr 2004

A design patterns approach for designing and explaining a Storage Explorer application. The application is used to explore file composition in a computer storage.

Download demo application - 14 Kb

Download source - 21 Kb



Introduction

A lot of information about GoF design patterns has been published, however most of the examples use separate illustration for each pattern, and the sample code is not an application with a real function. The Storage Explorer is a small application that is designed to use several design patterns together. While the main purpose of this submission is to share the code than to teach the reader about design patterns, the discussion about the application design itself is explained using design patterns.

This article discusses the structure of the application, the design patterns that I use, how it works and the advantages of using them. In my opinion design pattern is worth learning. Developers who have design pattern skill can recognize the intent of a program and the purpose by identifying the patterns. The classes with complex relations then can be understood better and faster than if the developers only understand basic OO design. The design patterns can be used as a thinking model for the designer and the code reader.

The application is used to explore file composition inside a computer storage. The design patterns that are used are: Strategy, Observer, Adapter, Template Method, Singleton and Wrapper Fa&#xFFFD;ade. The first five are known as GoF design patterns and the last one is a POSA pattern (POSA book volume-2).

Background

The idea of this application came when I desperately needed an application similar to Windows Explorer that can show me the file size composition inside my storage. I want to know, for example, under the Program Files folder, which folder uses what size of space, including the numbers in percentage. I want also to know how big my mp3 total size in drive D compare to my jpg files. The information I want to see should look like this:

```
C:\Program Files:
        Microsoft           445,123 KB          43%
        Adobe               234,744 KB          25%
        Symantec             98,906 KB          10%
        ...
```

And this

```
D:\
        *.mp3               602,456 KB          47%
        *.jpg               305,830 KB          30%
        *.doc               245,355 KB          20%
        ....
```

I need the information is presented in a chart to help me visualize the numbers as well.

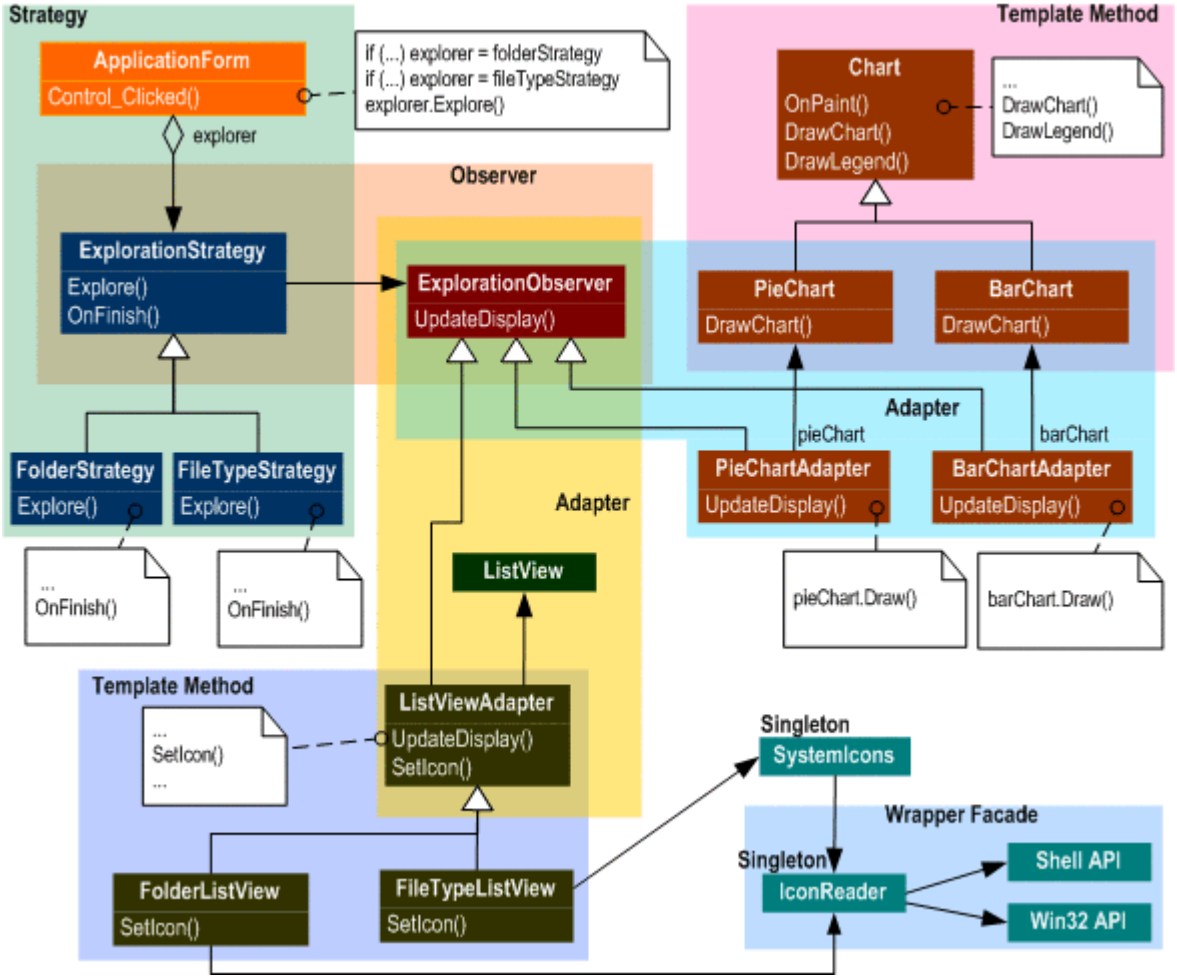
The Features

The application features are:

- A TreeView containing nodes that represents file system folders. The folders can be selected to see the information about file size structure within that path.
- It shows information about files size grouped by folders
- It shows information about files size grouped by file type
- The information is presented in listview, pie chart and bar chart on the right panel.

The Design Patterns inside the Application

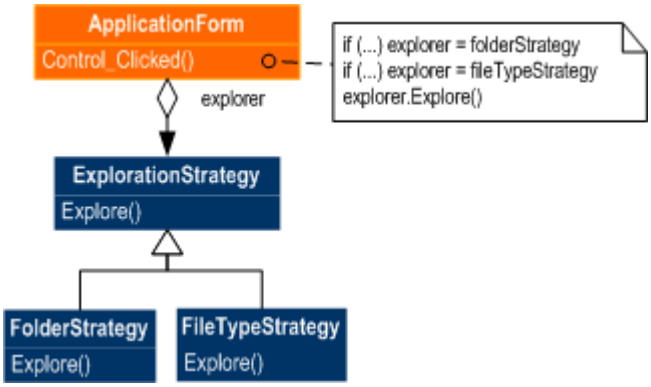
The design patterns used in this application is shown in the following figure.



The discussion for each design pattern is given below.

Strategy

Intent: "Define a family of algorithms, encapsulate each one, and make them interchangeable" (GoF).



The application explores the file system using two different algorithms, which is implemented in two strategy classes: `FolderStrategy` and `FileTypeStrategy` (see Figure above). `FolderStrategy` class implement an algorithm that explores files and sums the file size under different folders in a path. `FileTypeStrategy` class implements an algorithm that explores files and sums the file size that has the same type.

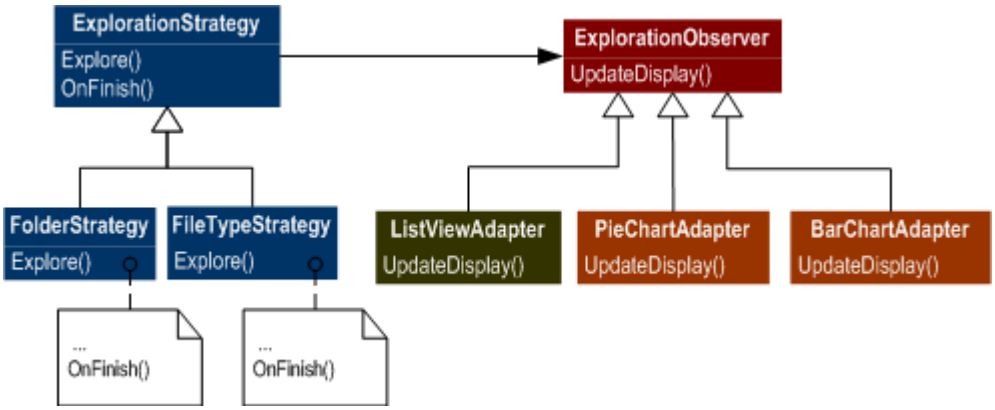
Now suppose we want to add a new algorithm. It can be done by creating a new subclass under `ExporationStrategy` class and putting the new algorithm code under the `Explore()` method. During the runtime create the instance of the new class and assign it dynamically to the explorer object. When we call `explore.Explore()` method the algorithm is run. That is the idea of the strategy pattern: encapsulate and make the algorithm interchangeable. The implementation looks like this:

C# Shrink ▲

```
public abstract class ExplorationStrategy
{
  public virtual void Explore (...){}
}

public class NewAlgorithm : ExplorationStrategy
{
  public override void Explore (...)
  {
    // the new algorithm

  }
}

public class StorageExplorerForm : System.Windows.Forms.Form
{
  // Somewhere in the initialization section

  ExplorationStrategy explorer;
  ExplorationStrategy folderStrategy = new FolderStrategy ();
  ExplorationStrategy fileTypeStrategy = new FileTypeStrategy ();
  ExplorationStrategy newStrategy = new NewAlgorithm ();

  // Somewhere else, an algorithm is selected

  switch (...)
  {
    case 0: explorer = folderStrategy; break;
    case 1: explorer = fileTypeStrategy; break;
    case 2: explorer = newStrategy; break;
  }

  // Execute the algorithm

  explorer.Explore (...);
}
```

Observer

Intent: Define a one-to-many dependency between object so that when one object changes state, all its dependent are notified and updated automatically (GoF).



In the application, this pattern creates a relationship between the subjects (concrete `ExplorationStrategy` object) and the observers (concrete `ExplorationObserver` object), so that when an exploration process finishes, the subject notifies the observer, and then the observer display the result (see Figure above). The concrete subject (`FolderStrategy` or `FileTypeStrategy`) notify the observers (`ListViewAdapter`, `PieChartAdapter` and `BarChartAdapter`) by calling `OnFinish()` method, which in turn generates an event. The event is sent to the observers and then handled by them. The relationship is defined at the abstract class and the concrete classes then inherits it.

The object and observer relationship is established by registering the observer `UpdateDisplay()` method to `ExplorationStrategy.Finish` event, as shown below:

C#

```
public abstract class ExplorationObserver
{
  public void SubscribeToExplorationEvent (ExplorationStrategy obj)
  {
    obj.Finish += new ExplorationFinishEventHandler (UpdateDisplay);
  }
}
```

And during application initialization in the client (the application form), the concrete observer object call `SubscribeToExplorationEvent()` method and pass the instance of the concrete strategy as the parameter. Since then the subject-observer relationship has been established. It shows as folows:

C#

```
// Initialization in the client:

// Create the concrete subject

ExplorationStrategy folderStrategy = new FolderStrategy ();
ExplorationStrategy fileTypeStrategy = new FileTypeStrategy ();

// Create the concrete observer

ExplorationObserver pieChart = new PieChartAdapter ();

// Subscribe the concrete observer object to the concrete strategy object

pieChart.SubscribeToExplorationEvent (folderStrategy);
pieChart.SubscribeToExplorationEvent (fileTypeStrategy);
```

Now let see the beauty of this pattern. Suppose we want to change the application to accommodate a new requirement. We want to save the exploration result into a file in addition to displaying it on the screen. For that purpose, inherits a new class from `ExplorationObserver` class and put the code that saves the result to a file inside `UpdateDisplay()` method. Create the instance of new class then call `SubscribeToExplorationEvent()` with the concrete `ExplorationStrategy` object as the parameter. Finish. When the application is run, information will be sent to the display and a file as well. The code is shown below.

C#

```
public class NewConcreteObserver : ExplorationObserver
{
  public override void UpdateDisplay (object o, ExplorationFinishEventArgs e)
  {
    Hashtable result = e.ExplorationResult;

    // Write the result to a file

  }
}

// Somewhere, during the initialization in the client

...
ExplorationObserver fileSaver = new NewConcreteObserver ();
fileSaver.SubscribeToExplorationEvent (folderStrategy);
fileSaver.SubscribeToExplorationEvent (fileTypeStrategy);
```

Observer design pattern is actually always used in the event driven programming. The idea is often used without being realized. However knowing the conceptes is still important if we want to collaborate it with another pattern like Adapter, as explained in the next section.

Adapter

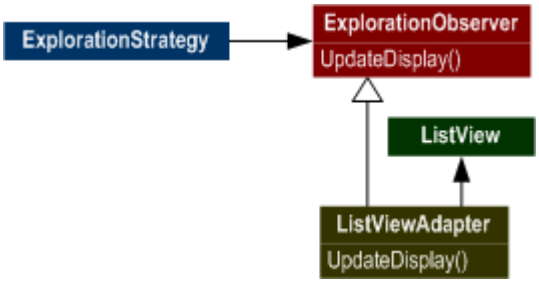
Intent: Convert the interface of a class into another interface client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces (Gof).

Now here is the case: I want the exploration result to be displayed in a .NET `ListView` class. However I need to modify the `ListView` capability so it can receive auto notification when an exploration finishes. It means I need to change `ListView` class to have `ExplorationObserver` class signatures and therefore I can override the `UpdateDisplay()` method, which is called when the `Finish` event comes. Unfortunately I cannot modify .NET ListView class. The solution is to create a new class, namely `ListViewAdapter`, that uses `ListView` capability while still has `ExplorationObserver` class signatures. Here adapter pattern plays the role. The `ListViewAdapter` class is an Adapter. In GoF terms: the `ListViewAdapter` class modify the interface of `ListView` class to have `ExplorationObserver` interface that `ExplorationStrategy` expects.

The adapter pattern has two forms: class adapter and object adapter. The class adapter is implemented by creating `ListViewAdapter` that inherits from `ListView` and `ExplorationObserver` class. Because C# does not support multiple inheritances then this is not possible. Actually this can be

possible if `ExplorationObserver` is an interface instead of an abstract class. `ListViewAdapter` then could implement the interface and inherit from the`ListView` class. But in this case `ExplorationObserver` has some implementation inside the class, which makes being an interface is not an option.

The second form, the object adapter, which is used in this application, uses object composition (see Figure below).

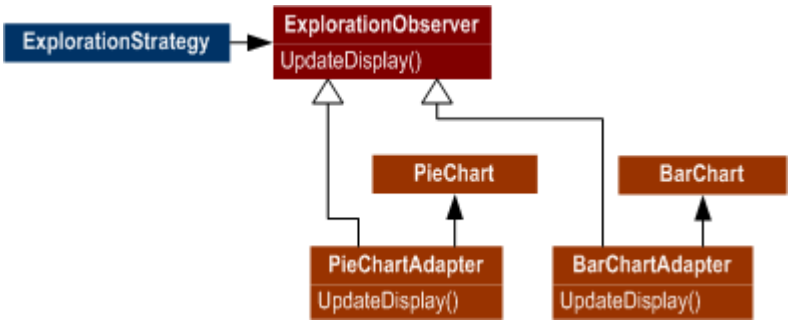


The `ListViewAdapter` class inherits the `ExplorationObserver` class. To have a `ListView` capability, the adapter creates a `ListView` object and use the object when necessary. The advantage of this approach is the `ListView` object and its members is hidden from the `ListViewAdapter` user. The code is shown below:

C#

```
public class ListViewAdapter : ExplorationObserver
{
  protected ListView listView;

  public ListViewAdapter()
  {
    // Create a ListView object and initialze it

    listView = new ListView();
    ...
  }
}
```

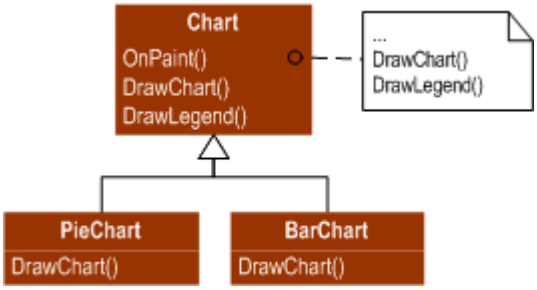
The same approach is applied to the `PieChartAdapater` and `BarChartAdapter` class. Again I assume that I have already had those chart classes and I do not want to modify them. The solution is again to create an adapter class that inherits from `ExplorationObserver` class and create the chart object inside the adapter (see Figure below).



Template Method

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain step of an algorithm without changing the algorithm structure (GoF).

The template method itself is a method in a class that calls other primitive method(s), so that when the subclass overrides the primitive method(s), the template method in the descendant class produces different results. Now let see the template method implementation in this application (see Figure below).



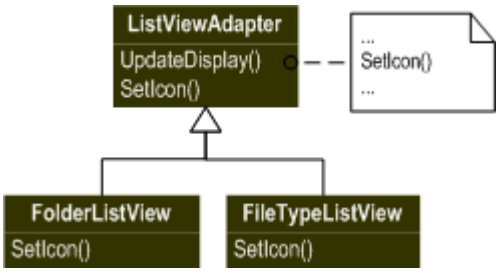
The template method inside the Chart is the `OnPaint()` method. Inside the `OnPaint()` there has been defined some steps to draw a chart. The steps include the process to initialize the Graphics object, draw the legend and draw the chart image. If we want to inherits this class to be some different types of chart like `PieChart` class and `BarChart` class , then inside the `OnPaint()` method convert the drawing chart operation to become a `DrawChart()` method call. In the `Chart` class itself the `DrawChart()` is a blank method. In the `PieChart` class, which inherits from the `Chart` class, the `DrawChart()` contains implementation to draw a pie images, and in the `BarChart`, the `DrawChart()` draws a bar images. We can do the same thing to the drawing the legend, the title, and every steps involved in the drawing chart process. The code is shown below:

C# Shrink ▲

```
public abstract class Chart : Control
{
  // OnPaint is a template method. It contains steps

  // necessary to draw a complete chart

  public OnPaint ()
  {
    ...
    DrawLegend (...);
    DrawChart (...);
  }

  protected abstract void DrawChart (...)
  protected virtual void DrawLegend (...) { // Draw legend }

}

public abstract class PieChart : Chart
{
  protected virtual void DrawChart (...)
  {
    // Draw the pie chart image

  }
```

```
}

public abstract class BarChart : Chart
{
  protected virtual void DrawChart (...)
  {
    // Draw the bar chart image

  }
}
```

I use the same approach with the `ListViewAdapter` class. It has two descendant class, `FolderListView` and `FileTypeListView`. Both classes have very similar steps when displaying information, except a step to set the icon. The `FolderListView` set a single icon representing folder image while the `FileTypeListView` select the icon from the series of image in the `ImageList` . Because the difference, that step is deferred to the descendant class (see Figure below)



The key to use the Template Method is to define a skeleton of an algorithm as general as possible in the base class and defer the steps that could vary to the descendant class. Also in the base class, we need to create some default implementation of the deferred step in case the descendant classes do not want to override the steps. Once we have a good template and default implementation, then we can enjoy creating variety of descendant classes with less effort.

Singleton

Intent: ﷺEnsure a class only has one instance, and provide a global point of access to itﷺ [GoF].

In this application there are two classes in which each of them needs only one instance to run: `FileIcons` class and `IconReader` class. The`FileIcon` class is used to retrieve and buffer the shell icons from the Windows Shell. This class represents the collection of the icon in the system. From the definition we know that it needs only one instance of the class running in the application. The `IconReader` class is used to retrieve the icon from the system without buffering. It calls the Shell API function. We need only one instance of this class as well and therefore it is implemented as Singleton.

MSDN provides an example on how to implement the singleton in .NET using C#. In this application the .NET singleton is implemented as follows:

C#

```
sealed class FileIcons
{
  ...

  // By making it private, the class cannot be explicitly created

  private FileIcons () {}

  // This is the statement that ensures only one instance exists

  public static readonly FileIcons Instance = new FileIcons();

  public int GetIconIndex (...)
  {
    ...
  }
}
```

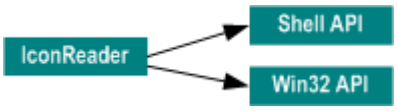
To use the `FileIcons` class we just use `FileIcons.Instance` property without needing to create the class, like this:

C#

```
int iconIndex = FileIcons.Instance.GetIconIndex (...);
```

Wrapper Facade

Intent: • Encapsulate the functions and data provided by existing non-object oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces• (POSA volume 2).



To access the shell icons from the windows operating system we must call the Shell API and Win API functions. The function call involves strict usage of data structure and complex parameter. To encapsulate that complexity I created an `IconReader` class. POSA says this is different from Facade pattern since the Facade encapsulates relationship complexity among classes, while the Wrapper Facade encapsulates non OO functions. Also this pattern is more specific in its purpose. For example, the Shell API function that is used to retrieve icon from the Windows Shell is the `SHGetFileInfo()` function. This function can be used to do a lot of job, not only for icon retrieval. Because the `IconReader` class is specific for Icon related jobs then I create only methods specific for it, they are `GetSmallFileTypeIcon()` method (to retrieve icon representing file extension) and `GetClosedFolderIcon()` (to retrieve icon representing a closed folder). The icon retrieval also needs to call Win32 API `DestroyIcon()` function. The idea behind this pattern is: don't look at to the complexity behind the process, as long as it is for delivering robust and cohesive functionalities, then use all the resources needed, and present only the cohesive interface to the class user.

License