**Observer Design Pattern**

Integrantes

- Cristopher Núñez Del Prado Mansilla ---------- cnpm18  
- Diana Lucía Rodríguez Delgado -------------------- dianalu  
- Claudia Milagros Zevallos Rivera ---------------- claumzr

**Actividades**

1. Utilice las versiones anteriores de su aplicación de repositorio: debe garantizar la inclusión de la capa DataAccess.

2. Agregar el proyecto Core para los models.

3. Implemente una capa de UI para hacer cambio en la data.

4. Implemente el patrón Observer para que las entidades de datos, automáticamente refresquen la base de datos ante cambios realizados en UI.

5. Debe mantener la capa core, totalmente desacoplada.

6. Incluya en su informe, dos estrategias para implementar el patrón Observer en el lenguaje de su elección. Qué facilidades brinda el lenguaje para facilitar la implementación.

El lenguaje escogido para la implementación del patrón Observer es C# puesto que facilita el desarrollo en los siguientes aspectos:

* La primera estrategia es que el objeto crear es **Observador**, que es una interfaz que se le llama **IObservador**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | using System;    namespace ConsoleApplication1  {      // Observador es una interfaz que define que método que el ObservadorConcreto debe implementar      interface IObservador      {          void Actualiza(string valor);      }  } |

El segundo elemento esta es la clase **Observador concreto** que implementará el método Actualiza de la interfaz. Define una variable **valor** que se guarda en cada observador como una copia de por el resto de los otros Observadores concretos y que debe sincronizarse en caso de un cambio por parte del Sujeto con tal de que todos los Observadores tengas el mismo valor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | using System;    namespace ConsoleApplication1  {    /\* Esta clase es la clase conceptualmente es un Observador real que implementa el método definido en la interfaz.  Tiene la propiedad compartida por el resto de los observadores (en este caso una propiedad llamada Valor) y que es la que los otros observadores quieren monitorear, también actualiza su valor e imprime por pantalla el cambio. \*/      class ObservadorContreto: IObservador      {          string nombre;          string valor = "0"; //por defecto            //Constructor: definimos un nombre para el observador real          public ObservadorContreto(string nombre)          {              this.nombre = nombre;          }            // Notifica y actualiza el nuevo valor en este ObservadorConcreto e imprime en pantalla.          public void Actualiza(string valor)          {              // Actualizamos el valor con tal de sincronizar              this.valor = valor;                Console.WriteLine("Se notifica a {0} que Sujeto ha cambiado el Valor a {1}", nombre, valor);          }      }  } |

El tercer elemento es la clase Sujeto, que es abstracta, y usa métodos del Observador concreto creado recientemente:

using System;

using System.Collections;

namespace ConsoleApplication1

{

/\* Clase abstracta que almacena una colección de observadores subscritos y los notifica de algún cambio, le dice a cada ObservadorConcreto subscrito que actualize el valor e imprima en pantalla el cambio.

Es responsabilidad de cada ObservadorConcreto imprimir en pantalla, pero es responsabilidad del Sujeto indicarles a ellos que lo hagan. \*/

abstract class Sujeto

{

ArrayList coleccionObservadores = new ArrayList();

public void RegistraObservador(ObservadorContreto observador)

{

coleccionObservadores.Add(observador);

}

public void DesRegistraObservador(ObservadorContreto observador)

{

coleccionObservadores.Remove(observador);

}

public void NotificaAObservadores(string valor)

{

foreach (ObservadorContreto observadores in coleccionObservadores)

{

observadores.Actualiza(valor);

}

}

}

}

* La segunda estrategia es el elemento es la clase Sujeto concreto o real, que hereda de Sujeto:

using System;

namespace ConsoleApplication1

{

// Clase real que implementa el Sujeto. Esta clase es usada para indicar que cualquier cambio afecta a los otros observadores

class SujetoContreto: Sujeto

{

public void CambiaValor(string valor)

{

NotificaAObservadores(valor);

}

}

}

El elemento es la clase Programa o Main, que utiliza la clase SujetoConcreto para registrar Observadores y cambiar un valor que se replicará o sincronizará en cada uno de ellos y se notificará en pantalla. Luego de des-registran 2 Observadores y cambia de nuevo el valor para ver a quien informa:

using System;

namespace ConsoleApplication1

{

class Programa

{

static void Main(string[] args)

{

SujetoContreto sujetoConcreto = new SujetoContreto();

// Se crea 4 observadores concretos

ObservadorContreto observdor1 = new ObservadorContreto("");

ObservadorContreto observdor2 = new ObservadorContreto("");

ObservadorContreto observdor3 = new ObservadorContreto("");

ObservadorContreto observdor4 = new ObservadorContreto("");

/\* SujetoConcreto le dice a Sujeto que agrege un nuevo ObservadorConcreto a su colección. Esto lo hace el Sujeto sin derivarle esta tarea a nadie más\*/

sujetoConcreto.RegistraObservador(observdor1);

sujetoConcreto.RegistraObservador(observdor2);

sujetoConcreto.RegistraObservador(observdor3);

sujetoConcreto.RegistraObservador(observdor4);

/\*Por medio de SujetoConcreto decimos que cambiará el valor, pero lo que por abajo pasa es que el SujetoConcreto le dice al Sujeto que actualice el valor, y el Sujeto lo que hace es decirle a cada ObservadorConcreto que lo haga\*/

sujetoConcreto.CambiaValor("10");

Console.WriteLine("");

sujetoConcreto.DesRegistraObservador(observdor3);

sujetoConcreto.DesRegistraObservador(observdor4);

Console.Read();

}

}

}

**Qué facilidades brinda el lenguaje para facilitar la implementación.**

C# brinda múltiples facilidades para la implementación por ejemplo:

* IList

Representa una colección de objetos no genéricos a los que se puede obtener acceso por índice

[ComVisibleAttribute(true)]

public interface IList : ICollection, IEnumerable

* Clases implementadas

En c# para implementar el patrón de diseño Observador se apoya en dos interfaces que ya están definidas en el lenguaje, son las clases IObservable e IObserver, de las cuales se hereda y se puede implementar clases concretas que implementen las clases definidas, dándole así funcionalidad de añadir y quitar observadores en la clase concreta Observable e implementando la función Update en la clase concreta Observer se ejecuta cada vez que se actualiza el estado del Observable.

public interface Subject

{

    public void registerObserver(Observer observador);

    public void removeObserver(Observer observador);

    public void notifyObservers();

}

* Referencias Inválidas

A la hora de implementar este patrón se debe ser cuidadoso cuando un elemento observable desaparece. En ciertos lenguajes será el gestor de memoria el que cada cierto tiempo debe de limpiará las referencias liberadas, pero si un observable que sigue siendo observado puede no liberarse nunca. Para solucionar este problema en C# puede crearse una función “destruir (destroy) ”, que notifique al gestor que el elemento observable va a desaparecer y si no estamos usando la variante del gestor el observable directamente des-registrará a sus observadores. Antes de esto hay que eliminar las referencias a este elemento, por tanto, hay que eliminar a los observadores antes de eliminar al observable, ya que así evitaremos tanto que aparezcan referencias inválidas al objeto una vez éste haya sido eliminado, como que se produzcan operaciones inválidas intentando invocarlo.