EJERCICIOS PATRONES DE DISEÑO #2

(Respuestas al final)

EJERCICIOS SACADOS DE LA GUIA

1. Winforms

La aplicación de ventas de un comercio cuenta con una calculadora entre sus utilidades, que permite al personal del local hacer ciertas cuentas simples que forman parte de sus tareas diarias. Sin embargo cada vez que la invocan a través del menú, se abre una nueva ventana en lugar de usar una ya existente, lo que resulta en decenas de ventanas abiertas durante el día.

a. ¿Qué patrón podría utilizar para evitar esta situación?

b. Codifique la clase calculadora haciendo hincapié en los cambios que haría a dicha clase

2. Feeds

Un concentrador de contenidos web necesita recolectar información de varios orígenes para presentarlos en su portal, en principio están interesados en poder recibir feeds en formato XML y JSON.

a. Elabore el diagrama de clases básico para representar esta situación.

b. Como modificaría la estructura anterior si necesitaran agregar además html, atom, rss, rdf y txt.

3. Biblioteca privada

Suponga un sistema de biblioteca de una universidad privada en la cual el alumno tiene la posibilidad de solicitar libros para préstamo. Las condiciones que se exigen son que sea un alumno regular, tenga la cuota al día y no adeude ningún libro. Para averiguar las dos primeras condiciones debe comunicarse con el subsistema de alumnado y el subsistema administrativo respectivamente.

a. Cómo podría facilitar el acceso a los diferentes subsistemas y a la clase de préstamos propia del subsistema de biblioteca.

b. Realice un diagrama de clases y sus dependencias con los distintos subsistemas.

4. Facebook

Considere una aplicación de Facebook en la cual debe implementar una función de invitación personalizada, mostrando en una lista todos los amigos de determinado usuario, recuerde que estás imágenes deben cargarse de manera remota y existen usuarios que pueden poseer hasta 5000 amigos.

a. ¿Qué patrón podría utilizar para evitar la carga inicial de imágenes en forma innecesaria?

b. Realice el diagrama de clases y el diagrama de secuencia respectivo.

5. Juego

Imagínese una maquina de estados simple perteneciente al personaje de un juego, que puede moverse, atacar y detenerse, pasando por diferentes estados tal como lo muestra el siguiente diagrama de estados.

a. Realice el diagrama de clases usando el patrón adecuado

b. Implemente el diagrama de clases utilizando algún lenguaje.

EJEMPLOS SACADOS DE EXÁMENES:

1. Teniendo en cuenta, que todos los jugadores inscriptos en el torneo están registrados en el sistema, se contempla la situación de que se les notifique cuando se finalizó la inscripción de su equipo, se les notifique el fixture de los partidos a jugar, que puedan recibir las novedades en cuanto a cambios de días y horarios de partidos pautados, como así también de los resultados de los partidos del torneo en el que están inscriptos.

a. ¿Qué patrón podría implementarse?

b. Realice el diagrama de clases correspondiente.

2. Si se quisiera implementar un aviso a los clientes con recomendaciones de que hay nuevos videos cargados que son de su preferencia.

a. Qué patrón de diseño utilizaría?

3. Teniendo en cuenta que debe haber una única conexión a la BD por la cual encauzar las peticiones.

a. Plantee qué patrón debería utilizarse.

b. Realice el código en el lenguaje de programación que prefiera enfocándose en las características particulares del patrón.

RESPUESTAS:

Aquí tienes las posibles respuestas a cada escenario planteado, con la lógica detrás de la elección del patrón y breves explicaciones:

Guía de Trabajo Práctico

1. **Winforms**
   - **Patrón Aplicado**: Singleton
     - **Por qué**: Para garantizar que solo exista una única instancia de la calculadora y evitar múltiples ventanas abiertas.
     - **Cambios en la Clase**:

```
public class Calculadora
{
    private static Calculadora instanciaUnica;
    private static readonly object candado = new object();

    private Calculadora() { }

    public static Calculadora ObtenerInstancia()
    {
        lock (candado)
        {
            if (instanciaUnica == null)
            {
                instanciaUnica = new Calculadora();
            }
        }
        return instanciaUnica;
    }

    public void Abrir()
    {
        // Lógica para mostrar la calculadora
    }
}
```

2. **Feeds**
   - **Patrón Aplicado**: Factory Method
     - **Por qué**: Para manejar la creación de los objetos feed sin acoplar la clase principal a los tipos específicos.
     - **Diagrama Inicial**:
       - Clases: `Feed`, `FeedXML`, `FeedJSON`.

        - Método Factory: `Feed CrearFeed(string tipo);`.
    - **Modificación para nuevos formatos**:
        - Agregar subclases como `FeedHTML`, `FeedAtom`, y actualizar el método factory para soportar los nuevos formatos.

3. **Biblioteca privada**
    - **Patrón Aplicado**: Facade
        - **Por qué**: Para simplificar el acceso a múltiples subsistemas (alumnado, administrativo) y centralizar la lógica en una única clase.
        - **Estructura**:
            - Clase `BibliotecaFacade`: métodos como `VerificarPrestamo()`.
            - Interacciones con `SubsistemaAlumnado`, `SubsistemaAdministrativo`, y `ClasePrestamos`.

4. **Facebook**
    - **Patrón Aplicado**: Proxy
        - **Por qué**: Para manejar la carga diferida (lazy loading) de las imágenes de los amigos.
        - **Estructura**:
            - Clase `ProxyImagen` que carga las imágenes solo cuando se necesitan.
            - Clase `ImagenReal` que maneja la carga real.
        - **Diagrama de Secuencia**:
            - Usuario → ProxyImagen (carga bajo demanda) → ImagenReal.

5. **Juego**
    - **Patrón Aplicado**: State
        - **Por qué**: Para gestionar los comportamientos del personaje según su estado (Mover, Atacar, Detenerse).
        - **Estructura**:
            - Contexto: `Personaje`.
            - Estados concretos: `EstadoMover`, `EstadoAtacar`, `EstadoDetener`.
        - **Implementación en Código**:

```
public interface IEstado
{
    void EjecutarAccion();
}

public class EstadoMover : IEstado
```

```
{
    public void EjecutarAccion() => Console.WriteLine("Moviéndose");
}

public class EstadoAtacar : IEstado
{
    public void EjecutarAccion() => Console.WriteLine("Atacando");
}

public class Personaje
{
    private IEstado estadoActual;

    public void CambiarEstado(IEstado nuevoEstado)
    {
        estadoActual = nuevoEstado;
    }

    public void RealizarAccion()
    {
        estadoActual.EjecutarAccion();
    }
}
```

Ejemplos de Examen

1. **Notificaciones de Torneo**
   - **Patrón Aplicado**: Observer
     - **Por qué**: Para notificar automáticamente a todos los jugadores inscritos sobre eventos.
     - **Estructura**:
       - Clase `Torneo` (Sujeto).
       - Clases `Jugador` (Observadores).
       - Método `Notificar()` .

2. **Aviso de nuevos videos**
   - **Patrón Aplicado**: Observer
     - **Por qué**: Para enviar recomendaciones a los clientes automáticamente cuando se suben nuevos videos.

3. **Única conexión a la base de datos**

- **Patrón Aplicado**: Singleton
  - **Por qué**: Para garantizar que solo exista una instancia de conexión a la base de datos.
  - **Código**:

```
public class ConexionBaseDatos
{
    private static ConexionBaseDatos instanciaUnica;
    private static readonly object candado = new object();

    private ConexionBaseDatos() { }

    public static ConexionBaseDatos ObtenerInstancia()
    {
        lock (candado)
        {
            if (instanciaUnica == null)
            {
                instanciaUnica = new ConexionBaseDatos();
            }
        }
        return instanciaUnica;
    }

    public void EjecutarConsulta(string consulta)
    {
        // Lógica para ejecutar consulta
    }
}
```

Si necesitas diagramas adicionales o más ejemplos, ¡avísame! 😊