Lab7: Patrones de diseño

1. Global Configuration Management

Patrón de diseño seleccionado: Singleton.

Usando *Singleton* se puede asegurar que existe una sola instancia de la clase de configuración y un único punto de acceso a dicha clase. Con un solo punto de acceso evitamos que existan múltiples y posiblemente incosistentes instancias de configuración.

Simulación e integración

En el archivo 1.cs se incluye un pseudocódigo de la clase propuesta. En este documento se hará referencia a dicho archivo.

- En cualquier punto de la aplicación, un componente puede recuperar la instancia del configuration manager gracias a *ConfigurationManager.Instance*.
- Cuando se llama a *ConfigurationManager.Instance* si no existe una instancia se crea una nueva, de lo contrario se devuelve la existente.
- Una vez que el componente ha recuperado la instancia global de *ConfigurationManager* tiene acceso al attributo *_settings* que podrá leer para acceder a los settings de interés, o alternativamente se pueden agregar métodos a la clase *ConfigurationManager* para recuperar configuraciones específicas.
- Con el constructor privado nos aseguramos de que componentes no tienen acceso este y solo puede ser usado una única vez.

2. Dynamic Object Creation Based on UserInput

Patrón de diseño seleccionado: Factory.

Usando el patrón *Factory* podemos tener una súperclase de la cual los diferentes elementos de interfaz hereden. Posteriormente, dependiendo de la entrada del usuario la clase *Factory* podrá decidir que tipo de objeto utilizar. Usando una clase *Factory* podemos encapsular toda la lógiac de instanciación y aislar esa lógica de la funcionalidad en cuestión y, además, respetamos el principio Open-Close de SOLID. Esto también permite crear nuevos tipos de elementos de interfaz con cambios mínimos en el código.

Simulación e Integración.

En el archivo 2.cs se incluye un pseudocódigo de la clase propuesta. En este documento se hará referencia a dicho archivo.

- La aplicación principal no se involucrará en absoluto con la instanciación del *UIElement*, solo llamará a la función *CreateUIElement* de la clase *Factory* especificando el tipo de acción que realizó el usuario.
- Si se necesitara crear un nuevo tipo de elemento, simplemente se puede crear la clase que herede de *IUElement* y adapatar el método *CreateUIElement*. La aplicación principal no necesita realizar ningún cambio.

3. State Change Notification Across System Components

Patrón de diseño seleccionado: Observador.

El patrón Observador es el ideal para escenarios como este, en el que múltiples componentes necesitán información sombre cambios en otro componentes. Tendremos entonces el objeto observado, *observable*, que notificará a sus dependientes cuándo reciba algún cambio.

Simulación e Integración

En el archivo 3.cs se incluye un pseudocódigo de la clase propuesta. En este documento se hará referencia a dicho archivo.

- Cada clase que se defina como un observador no necesita información adicional del objeto que observa, ni estar ligado a él.
- El único paso que se requiere como parte de la integración es que cada observador herede la Interfaz (como la definida en 3.cs) e implemente el método a ejecutar luego de cada cambio.
- Cuando se requiera que un nuevo objeto se subscriba a las notificiones del objeto observado, bastará con registrarla con el método RegisterObserver. Similarmente, cuando no se requira recibir más notificaciones, el objeto puede De-registrarse.

4. Efficient Management of Asynchronous Operations

Patrón de diseño seleccionado: Promises

Usaremos el patrón asíncrono de *Promises*. Sacaremos provecho de *async/await* en lenguajes como .NET para que la implementación y lectura de código sea sencilla y directa. *Async/await* facilitan el manejo de flujos complejos de llamadas asíncronas. Como además necesitamos coordinar múltiples tareas, podemos hacer uso de comandos como Task.WhenAll para intentar el inicio de todas las tareas simultáneamente y esperar a que todas sean completadas.

Simluación e integración

Este escenario no da una descripción específica, así que no propocionaremos código como para los anteriores. En cambio describimos de manera general la integración y posibles pasos a realizar.

- Se puede definir una clase coordinadora que ejecute todas las task en cuestión de manera asíncrona y haga uso de Task.WhenAll para esperar su finalización.
- Desde la aplicación principal usar el ejecutor definido en el paso anterior para lanzar todas las tareas sin bloquear el thread principal.