Introducción al módulo

Conoceremos en el presente módulo algunas características esenciales para la programación Orientada a Objetos, específicamente en las fases de **análisis y diseño** de aplicaciones basadas en dicho paradigma. En primera instancia conocer los requerimientos de sistema y planteamiento de soluciones. En segundo lugar el conjunto de métricas de diseño, que ayudarán a resolver problemas comunes que se dan en la ingeniería de Software en donde se plantean algunas recomendaciones de diseño eficiente. Desarrollar aplicaciones que cumplan con todos los requerimientos actuales es una tarea de ingeniería y tenemos la tarea de estudiar cada una de las métricas que nos ayudan a cumplir con esa tarea.

Análisis y diseño orientado a objetos.

El análisis y diseño orientado a objetos (ADOO) es un método estructurado para analizar y diseñar un sistema mediante la aplicación de conceptos orientados a objetos y desarrollar un conjunto de modelos de sistemas gráficos durante el ciclo de vida de desarrollo del software.

El ciclo de vida del software generalmente se divide en etapas que van desde descripciones abstractas del problema hasta diseños, pasando por el código y las pruebas, y finalmente hasta la implementación.

La principal diferencia entre el análisis orientado a objetos y otras formas de análisis es que mediante el enfoque orientado a objetos organizamos requisitos alrededor de objetos, que integran tanto comportamientos (procesos) como estados (datos) modelados a partir de objetos del mundo real con los que el sistema interactúa.

Las tareas principales en el análisis orientado a objetos (AOO) son:

- Describir las clases y sus relaciones usando el diagrama de clases.
- Describir la interacción entre los objetos usando el diagrama de secuencia.
- Aplicar principios de diseño de software y patrones de diseño.

La fase de análisis también se denomina a menudo "especificación de requisitos de software (ERS)"

Calidad en el paradigma Orientado a Objetos.

Algunos aspectos que implican que se haga un correcto diseño orientado a objetos, pueden ser:

1. Identificar correctamente los objetos.
2. Definir una jerarquía de clases para establecer una herencia de calidad
3. Identificar todas las relaciones entre clases
4. Definir interfaces
5. Resolver de forma general el problema de software, además de los detalles específicos

1. Respecto al tiempo

El tiempo de desarrollo en los proyectos son valiosos, por ende, muchos programadores cometen el error de no tener una etapa de análisis que ayude a que se planteen situaciones desde el inicio. Dicha etapa, implica tener el conocimiento del problema a resolver y dominar el paradigma orientado a objetos para establecer parámetros iniciales que sirvan como punto de partida para el desarrollo de software.

Aspectos a evitar en el análisis Orientado a Objetos

Algunas de las cosas que hacen que los proyectos se retrasen o prolonguen su desarrollo son los siguientes:

- Especificaciones incompletas.
- Especificaciones incorrectas.

También debemos considerar que los requerimientos de las aplicaciones cambian en relación al tiempo y que un análisis bien estructurado desde el inicio conllevara a que la evolución de las aplicaciones no sea problema alguno para el equipo de desarrollo.

Al final los costos en los proyectos de desarrollo se incrementan al no tener una planificación ni análisis inicial, la prioridad para los equipos de desarrollo recae en menor medida en el diseño y mayoritariamente en la implementación. Al final, si nos saltamos la parte del análisis, pareciera que estamos ahorrando tiempo, pero en realidad no es así, las reestructuraciones de código que aparecen por no tener esa primera etapa bien formada son bastantes, al final los costos se incrementan al verse afectados directamente por el tiempo de desarrollo.

2. Clases

Algunas de las condicionantes que hacen de una clase tener la calidad de software son las siguientes:

Métricas de medición de calidad de clases según Grady Booch:

- **Acoplamiento**: En la herencia se da un acoplamiento muy cercado, esto es, qué tan cerca pueden trabajar las clases entre sí.
- **Cohesión:** Refleja el comportamiento cercano de estado y comportamiento de la clase.
- **Suficiencia:** Debe reflejar los detalles suficientes para ser una clase útil.
- **Integridad:** Debe tener los elementos suficientes para ser una clase reusable en relación al tiempo y usuarios.
- **De carácter primitivo:** Las clases primitivas deben ser pequeñas y fáciles de entender para otros programadores.

3. Respecto al diseño.

Para una calidad del diseño en el paradigma orientado a objetos existen una serie de patrones que resuelven problemas comunes en el desarrollo de software, situaciones concurrentes en el entorno.

Patrones de diseño: Ventajas

1. Utilizar soluciones a problemas que ya han sido tratados.
2. Fomentar la estandarización de código en el desarrollo de software
3. Reutilización y optimización de código
4. Fomentar un vocabulario que ayude a definir problemas y soluciones en el desarrollo de software.

Los patrones se aplican:

Según ámbito: En Objetos y Clases

Según propósito: Creacionales, estructurales y de comportamiento.

Patrones Creacionales.

1. Abstraer el proceso de instanciación

1. Abstract Factory (Fábrica abstracta)
2. Builder (Constructor virtual)
3. Factory Method (Método de fabricación)
4. Prototype (Prototipado)
5. Singleton (Instancia única)

2. Creacional de Clase

Factory Method

3. Creacional de Objeto

Abstract Factory, Builder, Prototype y Singleton

Descripción:

A continuación una breve descripción de los patrones asemejándolos a algunas situaciones reales que pueden darse en el desarrollo de software:

Patrón Abstract Factory: Cuando se pretende crear diferentes familias de productos (ej: creación de interfaces gráficas de distinto tipo). y se prevé la inclusión de nuevas familias de productos y no se prevé añadir o modificar productos.

Patrón Singleton: Asegurar que una clase tiene una sola instancia, y brindar un punto de acceso global a ella. Controlar el proceso de instanciación, impidiendo la creación directa de objetos de esta clase, y facilitando un mecanismo de acceso a la instancia única: Un ejemplo básico en Java.

```
public class Singleton
{
	static private Singleton singleton;

	private Singleton() { }

	static public Singleton getSingleton()
	{
		if (singleton == null)
		{
			singleton = new Singleton();
		}
		return singleton;
	}
}
```

Algunos de los beneficios que podemos tener con dicho patrón:

1. Flexibilidad para realizar cambios en el proceso de instanciación.
2. Permite la especialización de la clase mediante herencia.
3. Permite la creación de un número concreto de instancias (una o unas pocas).

Patrones estructurales.

Las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades:

1. Adapter(Adaptador)
2. Bridge (Puente)
3. Composite (Objeto compuesto)
4. Decorator (Envoltorio)
5. Facade (Fachada)
6. Flyweight (Peso ligero)
7. Proxy

Patrón Adapter (Clase)

Generalmente se usa cuando es necesario transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda. Crear una nueva clase que será el Adaptador, que: Extienda el componente existente.

```
public class Empresa
{
	private List empleados;
	private Empresa()
	{
		empleados = new ArrayList();
	}

	public List devueveEmpleados()
	{
		return empleados;
	}

	public void añadeEmpleado(Empleado e)
	{
		empleados.add(e);
	}
}
```

Patrón Composite

Se usa cuando se ocupa construir objetos complejos a partir de otros más simples y similares entre sí.

Básicamente, construir el objeto complejo mediante una composición recursiva.

Patrones de Comportamiento.

Ayudan a definir la comunicación e iteración entre los objetos de un sistema, reduciendo el acoplamiento entre los objetos.

1. Chain of Responsibility (Cadena de responsabilidad)
2. Command (Orden)
3. Interpreter (Intérprete)
4. Iterator (Iterador)

5. Mediator (Mediador)
6. Memento (Recuerdo)
7. Observer (Observador)
8. State (Estado)
9. Strategy (Estrategia)
10. Template Method (Método plantilla)
11. Visitor (Visitante)

Comportamiento de Clase: Usan la herencia para distribuir el comportamiento entre clases. Los patrones que entran en este grupo: Interpreter y template method.

Comportamiento de Objetos: Describen como un grupo de objetos interaccionan para realizar una tarea. Los patrones que entran en el grupo: Chain of Responsability, Command, Iterator…

Descripción:

Patrón Chain of Responsability:

Se usan cuando es necesario establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor. Cuando en función del estado del sistema, las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.

Patrón Iterator:

Definir una interfaz que declare los métodos necesarios para acceder secuencialmente a una colección de objetos, sin necesidad de conocer la estructura interna de la colección. Métodos que podemos definir en la Interfaz Iterator son: primero(), siguiente(), haymas() , elementoactual()

Trabajando con objetos en Java

Estableciendo y obteniendo atributos:

Veamos un ejemplo en Java para la definición de las características básicas en Java para trabajar con objetos.

Crearemos un objeto persona y le estableceremos algunas propiedades. Ejecutar dicho código en el módulo de Desarrollo de la plataforma online.

```
public class Main
{
	public static void main(String[] args)
	{
		//Creando el objeto persona
		Persona persona = new Persona();
		persona.setNombre("Eduardo");
		persona.setApellido("Suarez");
		persona.setEdad (20);
		persona.setSexo ("M");

		System.out.println("Nombre: " + persona.getNombre() );
		System.out.println("apellido: " + persona.getApellido() );
		System.out.println("Edad: " + persona.getEdad() );
		System.out.println("Sexo: " + persona.getSexo() );
	}

	public static class Persona
	{
		//Private hace que los atributos sean accedidos dentro de la clase
		  private String nombre;
		  private String apellido;
		  private int edad;
		  private String sexo;

		//Métodos públicos para acceder a los datos
		  public String getNombre()
		    {
		    return nombre;
		  }
```

```
                public String getApellido()
                {
                        return apellido;
                }
                public int getEdad()
                {
                        return edad;
                }
                public String getSexo()
                {
                        return sexo;
                }

                //Métodos públicos para establecer los datos

                public void setNombre(String nombre)
                {
                        this.nombre = nombre;
                }
                public void setApellido(String apellido)
                {
                        this.apellido = apellido;
                }
                public void setEdad(int edad)
                {
                        this.edad = edad;
                }
                public void setSexo(String sexo)
                {
                        this.sexo = sexo;
                }
        }
}
```