Licenciatura en Sistemas - Orientación a Objetos II – 2020

Equipo docente: Mg. María Alejandra Vranic alejandravranic@gmail.com
Lic. Romina Mansilla romina.e.mansilla@gmail.com
Lic. Gustavo Siciliano gussiciliano@gmail.com

Patrones de diseño

Los desarrolladores no siempre tienen que encarar aplicaciones a partir de cero, muchas tareas y acciones se repiten en diversas formas a lo largo de cada problema, permitiendo reutilizar metodologías de trabajo anteriores. Los Patrones forman una base para la producción de soluciones a los temas recurrentes en los grandes desarrollos. Cada patrón registra una manera de resolver un problema, incluyendo cómo reconocer la presencia del mismo y cómo generar la solución para que se ajuste al contexto. Los patrones conducen naturalmente uno al otro, formando una especie de tela flexible de las decisiones que pueden resolver problemas a gran escala.

Christopher Alexander (1977): "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y describe la esencia de la solución a ese problema, de tal modo que pueda utilizarse esta solución un millón de veces más, sin siquiera hacerlo de la misma manera dos veces".

Vamos a dar un paso por los siguientes tipos de Patrones de Diseño:

1- Patrones estructurales:
   a. Composite.
   b. Facade.
2- Patrones de comportamiento:
   a. State.
   b. Observer.
3- Patrones creacionales:
   a. Singleton.

Patrón Facade (Fachada):

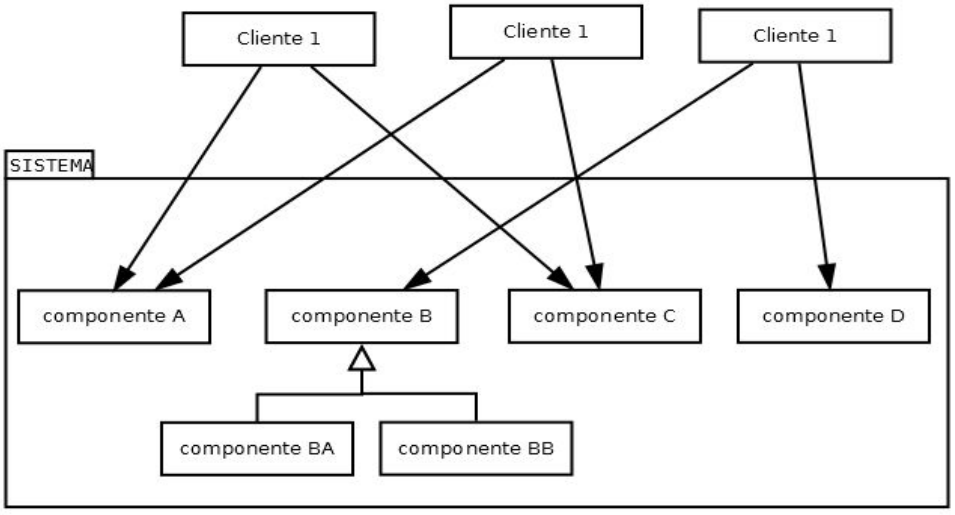
El patrón de Facade proporciona una interfaz unificada a un conjunto de interfaces de un subsistema. Esta interfaz es de alto nivel y hace que el subsistema sea más fácil de usar. De esta manera se envuelve un subsistema complicado con una interfaz más sencilla.

Este patrón tiene los siguientes componentes:

1. Facade: Clase que se transforma en el nexo entre el/los cliente/s y las clases que componen al subsistema.
2. Subsistema: Son las clases que implementan la funcionalidad. Estás realizan el trabajo solicitado por el Facade, sin conocerlo en verdad.
3. Cliente: Es el componente que realiza las peticiones al Facade

Problema:

Un Sistema en Subsistemas facilita el diseño y el mantenimiento, pero puede ocurrir que existan dependencias entre los subsistemas, como muestra la siguiente figura:



Cuando aplicar este patrón:

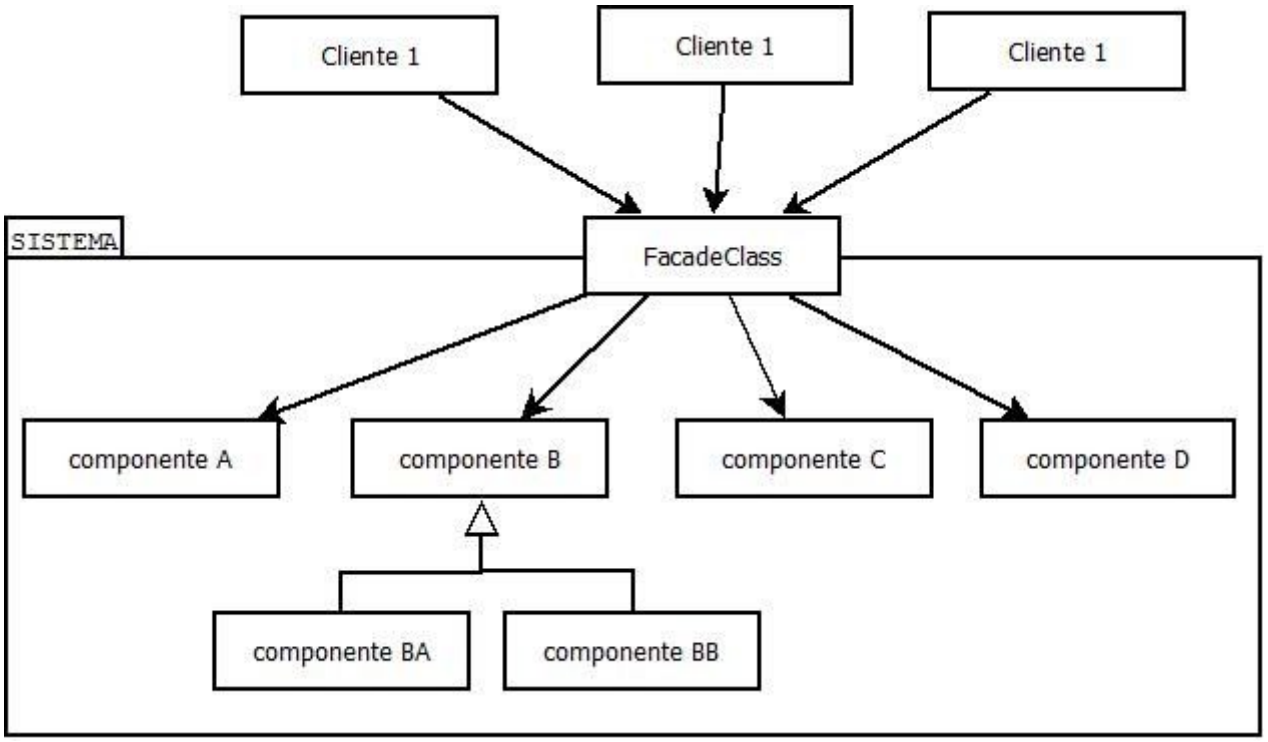
Cuando los subsistemas se vuelven complejos y existen muchas dependencias entre los clientes y los subsistemas que implementan, se diseña una Facade para minimizar el acoplamiento de los clientes y el subsistema, logrando así independencia y la portabilidad.

Podemos diseñar nuestro sistema en capas y proporcionar una interfaz Facade para definir un punto de entrada en cada nivel logrando que se comuniquen y cumpliendo el objetivo de minimizar las dependencias. De este modo también se minimizan la comunicación y las dependencias entre subsistemas. La interfaz Fachada se encarga de distribuir los servicios a los subsistemas.

Este patrón define una interfaz unificada de nivel superior a un subsistema que hace que éste sea más fácil de usar. Los consumidores se encuentran con una fachada del pedido y al realizar una petición reciben una interfaz de algún componente del subsistema.

Con el objetivo de minimizar la comunicación y dependencias entre subsistemas se introduce una interfaz más sencilla unificada para el subsistema o componente. Al diseñar una clase 'envoltorio' que encapsula el subsistema, esta clase captura la complejidad y la colaboración del componente, y los delegados de los métodos apropiados. El conocimiento necesario para relacionar los distintos módulos de nuestro sistema entre sí queda encapsulado entonces dentro de la clase Facade, la que nos ofrece puntos de entrada (métodos) que, con los datos mínimos necesarios, instancian a las distintas clases del modelo e invocan a sus métodos, devolviendo sólo la información que queremos obtener. El cliente usa únicamente el Facade, como muestra la siguiente figura:

Estructura:



Conclusiones:

1. Los clientes no ven los componentes del subsistema, minimizando la cantidad de objetos que se relacionan con el cliente lo que facilita el uso del subsistema.
2. Reduce el acoplamiento entre los clientes y el subsistema.
3. No presenta la limitación que las aplicaciones utilicen clases del subtema en caso que sea necesario, pudiendo elegir la generalidad o facilidad de uso.

Patrón Singleton (Único):

La idea del patrón es asegurar que una clase tenga sólo una instancia, y proporcionar un punto de acceso global a ella. Hay que hacer a la clase de la instancia de objeto único responsable de la creación, inicialización, el acceso y la aplicación. Se debe declarar la instancia como un miembro de datos estáticos privado y proporcionar una función pública que encapsula todo el código de inicialización, y proporciona acceso a la instancia. El cliente llama a la función de acceso (utilizando el nombre de la clase y el operador) siempre que se requiera una referencia a la instancia única.

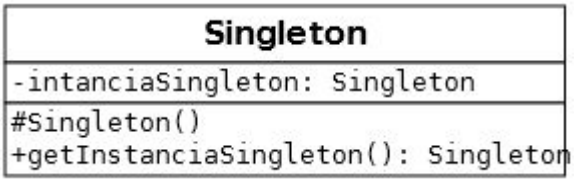
Este patrón tiene los siguientes componentes:

1. Singleton: Clase que tiene el valor estático.
2. Consumidores: Clases que hacen uso de ese valor estático de la clase Singleton.

Cuando aplicar este patrón:

En el caso de ser necesario garantizar la creación de sólo una instancia de una clase y que los clientes puedan acceder de cualquier lugar de acceso.

Estructura:



Conclusiones:

El patrón controla la creación de una única instancia.

Ejemplo:

```
package datos;

public class Empresa {
	private static Empresa instanciaEmpresa;
	protected String nombre;
	protected String email;
	protected Empresa(){
		this.inicializar();
	}
	public static Empresa getInstanciaEmpresa() {
		 if (instanciaEmpresa== null) {
		     instanciaEmpresa = new Empresa();
		    }
		 return instanciaEmpresa;
	}
	public String getEmail() {
		 return email;
	}
	public String getNombre() {
		 return nombre;
	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	private void inicializar(){ //pueden leer la instancia de un archivo xml
		this.setNombre("Soft Argentina");
		this.setEmail("softargentina@unla.edu.ar");
	}

}
```

```
package test;
import datos.Empresa;
public class TestSingletonEmpresa {
	 public static void main(String[] args) {
			System.out.println( "Objeto empresa");
			Empresa empresa=Empresa.getInstanciaEmpresa();
			System.out.println( empresa.getNombre());
			System.out.println( empresa.getEmail());
			 System.out.println( "Objeto empresa1");
			Empresa empresa1=Empresa.getInstanciaEmpresa();
			System.out.println( empresa1.getNombre());
			System.out.println( empresa1.getEmail());
System.out.println("empresa.equals(empresa1)="+empresa.equals(empresa1));
			}
		}
```

Preguntas:

¿Por qué está protegido el constructor?

¿Por qué el método getInstanciaEmpresa() necesita ser estático?

Referencias utilizadas:

Título: Patrones de Diseño

Autores: Erich Gamma - Richard Helm - Ralph Johnson -John Vlissides

Editorial: Pearson Addison Wesley

SourceMaking https://sourcemaking.com/design_patterns