Carrera: Licenciatura en Sistemas Materia: Orientación a Objetos II Año: 2018

Equipo docente:
Titular: Prof. María Alejandra Vranić alejandravranic@gmail.com
Ayudantes: Prof. Leandro Ríos leandro.rios.unla@gmail.com
Prof. Gustavo Siciliano gussiciliano@gmail.com
Prof. Romina Mansilla romina.e.mansilla@gmail.com

Patrones de diseño

Patrón Composite (Compuesto):

El patrón Composite utiliza una estrategia de composición recursiva. Con esto nos permite crear clases "complejas" a partir de clases "sencillas" y así formar una estructura en árbol para, mediante una interfaz o superclase, establecer un comportamiento que nos permitan tratar a todos los objetos de la misma manera.

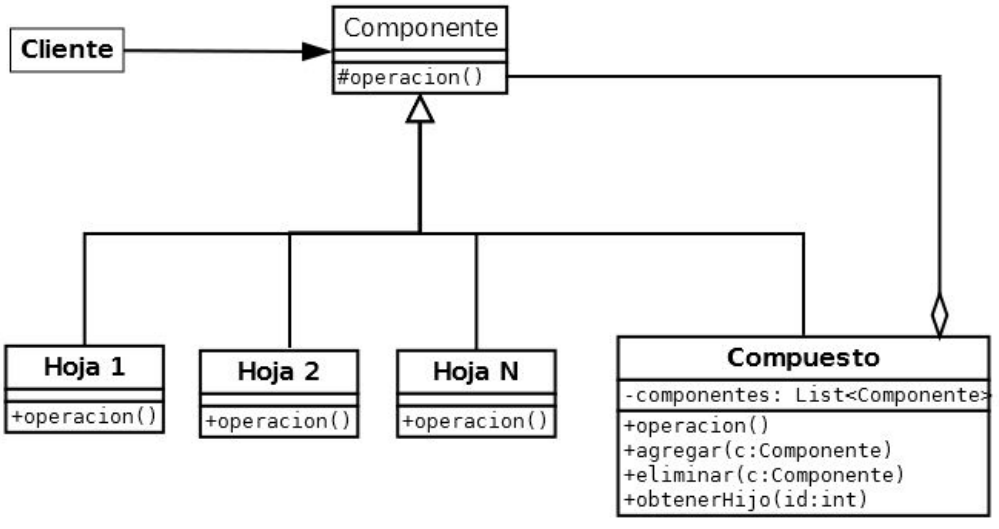
Este patrón tiene los siguientes componentes:

1. Componente: Clase que nos indica cual será el comportamiento de sus "Hojas".
2. Hojas: Clases que implementen o hereden el comportamiento, cada una tendrá su propia implementación.
3. Composite: Clase que define el objeto compuesto de hojas.
4. Cliente: Clase que utilice la implementación de este patrón.

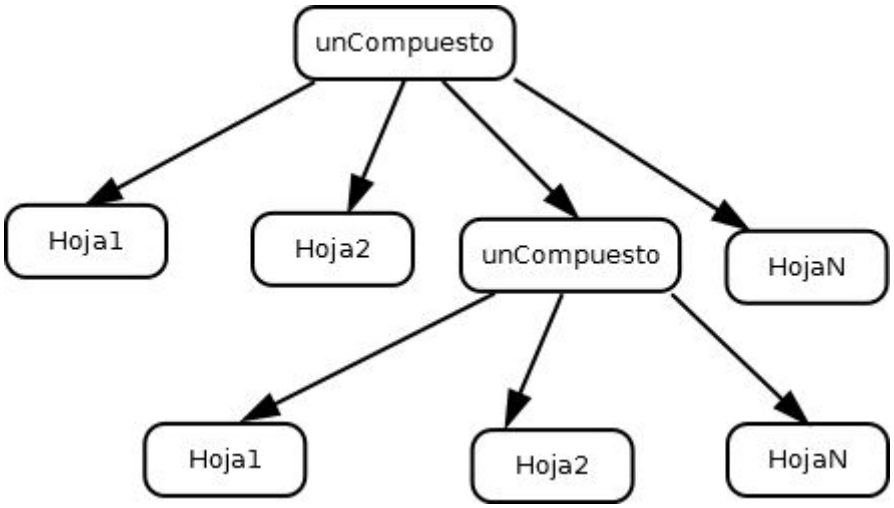
Cuando aplicar este patrón:

1. Para representar jerarquías de objetos parte-todo.
2. Cuando se necesite que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

Estructura:



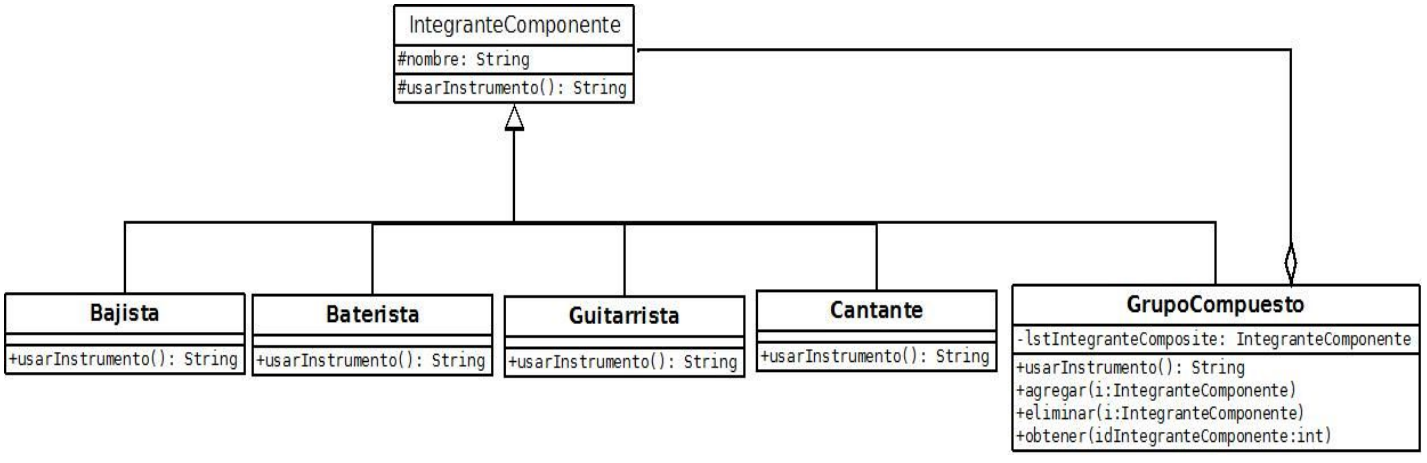
Una estructura de objetos común:



Conclusiones:

1. El patrón define jerarquías de clases formadas por objetos hojas y compuestos. Los objetos hojas pueden componerse en otros objetos más complejos que a su vez pueden ser compuestos y así de manera recurrente. Entonces la operacion() puede recibir un objeto hoja o un objeto compuesto.
2. Simplifica al cliente. Los clientes pueden tratar uniformemente las estructuras compuestas y la objetos individuales. Los clientes no conocen si se está de una hoja o un compuesto. Esto simplifica el código del cliente porque evita tener que escribir funciones con instrucciones if() anidadas en las clases que definen la composición.
3. Facilita añadir nuevos tipos de componentes. Si se definen nuevas subclases: Compuesto u Hoja, estas funcionarán automáticamente con las estructuras el código cliente existentes. No hay que cambiar los clientes para nuevas clases compuesto.
4. Puede hacer que un diseño sea demasiado general. La desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto. El diseño tiene la responsabilidad de cumplir las condiciones necesarias para agregar nuevos componentes.

Ejemplo simple:



Preguntas:

1. ¿Porque la clase GrupoCompuesto tiene una lista del tipo IntegranteComponente?
2. ¿Porque la clase GrupoCompuesto hereda de IntegranteComponente?

Supongamos un sistema que nos permita registrar bandas musicales.

- **Componente:**

```
package composite;

public abstract class IntegranteComponente {
	protected String nombre;

	public IntegranteComponente(String nombre) {
		this.nombre = nombre;
	}

	protected abstract String usarInstrumento();
}
```

- **Hojas:**

```
package composite;

public class Bajista extends IntegranteComponente {
	public Bajista(String nombre){
		super(nombre);
	}

	@Override
	public String usarInstrumento() {
		return "\nBajo: "+nombre;
	}

}
```

```
package composite;

public class Baterista extends IntegranteComponente {
	public Baterista(String nombre){
		super(nombre);
	}

	@Override
	public String usarInstrumento() {
		return "\nBatería: "+ nombre;
	}

}
```

```java
package composite;

public class Cantante extends IntegranteComponente {
	public Cantante(String nombre){
		super(nombre);
	}

	@Override
	public String usarInstrumento() {
		return "\nCantante: "+nombre;
	}

}
```

```java
package composite;

public class Guitarrista extends IntegranteComponente {
	public Guitarrista(String nombre){
		super(nombre);
	}

	@Override
	public String usarInstrumento() {
		return "\nGuitarra: "+nombre;
	}

}
```

- **Compuesto:**

```java
package composite;

import java.util.ArrayList;
import java.util.List;

public class GrupoCompuesto extends IntegranteComponente {

	private List<IntegranteComponente> lstIntegranteComponente;

	/*---------- Constructor ----------*/
	public GrupoCompuesto(String nombre){
		super(nombre);
		this.setLstIntegranteComponente(new ArrayList<IntegranteComponente>());
	}

	/*---------- Methods ----------*/
	@Override
	public String usarInstrumento() {
		String cancion = "\nGrupoCompuesto: "+nombre;
		for(IntegranteComponente integrante: this.getLstIntegranteComponente()){
			cancion = cancion.concat(integrante.usarInstrumento().concat(" "));
		}
		return cancion;
	 }
```

```
	public void addIntegranteComponente(IntegranteComponente integrante){
		this.getLstIntegranteComponente().add(integrante);
	}

	public void removeIntegranteComponente(IntegranteComponente integrante)
	{
		this.getLstIntegranteComponente().remove(integrante);
	}

	public List<IntegranteComponente> getLstIntegranteComponente() {
		return lstIntegranteComponente;
	}

	public void setLstIntegranteComponente(
			List<IntegranteComponente> lstIntegranteComponente) {
		this.lstIntegranteComponente = lstIntegranteComponente;
	}

}
```

-

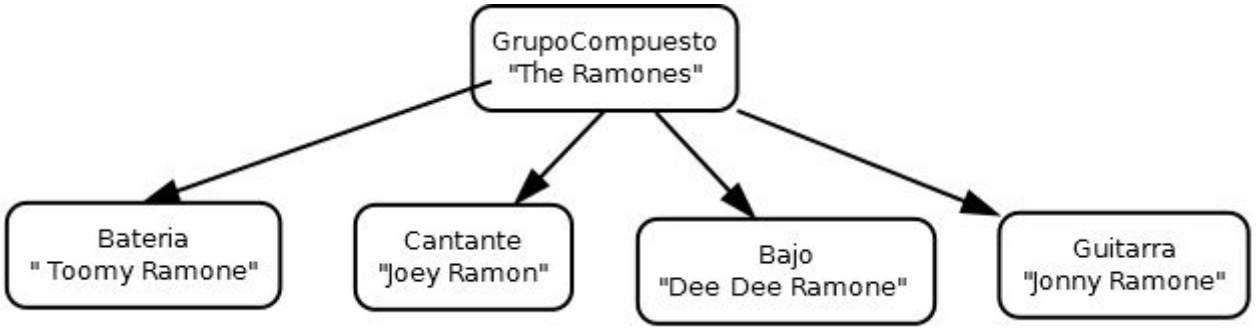
Test:

```
package test;

import composite.Bajista;
import composite.Baterista;
import composite.Cantante;
import composite.Guitarrista;
import composite.GrupoCompuesto;

public class Test {

	public static void main(String[] args) {


		GrupoCompuesto theRamones = new GrupoCompuesto("The Ramones");
		theRamones.addIntegranteComponente(new Baterista("Toomy Ramone"));
		theRamones.addIntegranteComponente(new Cantante("Joey Ramone"));
		theRamones.addIntegranteComponente(new Guitarrista("Jonny Ramone"));
		theRamones.addIntegranteComponente(new Bajista("Dee Dee Ramone"));
		System.out.println(theRamones.usarInstrumento());

		//Otro ejemplo pero con una banda con dos guitarristas
		GrupoCompuesto acdc = new GrupoCompuesto("AC-DC");
		GrupoCompuesto guitarristas = new GrupoCompuesto("Guitarristas");
		guitarristas.addIntegranteComponente(new Guitarrista("Angus Young"));
		guitarristas.addIntegranteComponente(new Guitarrista("Malcolm Young"));
		acdc.addIntegranteComponente(guitarristas);
		acdc.addIntegranteComponente(new Baterista("Chris Slade"));
		acdc.addIntegranteComponente(new Cantante("Brian Johnson"));
		acdc.addIntegranteComponente(new Bajista("Cliff Williams"));
		System.out.println(acdc.usarInstrumento());
	}

}
```

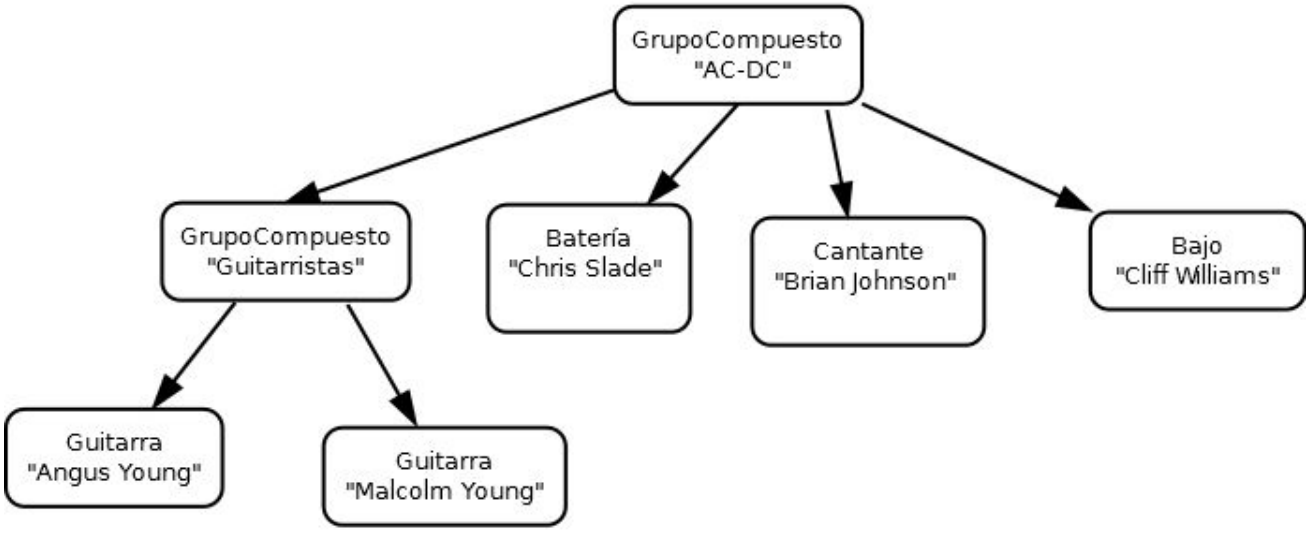
- **Run:**

```
GrupoCompuesto: The Ramones
Batería: Toomy Ramone
Cantante: Joey Ramone
Guitarra: Jonny Ramone
Bajo: Dee Dee Ramone

GrupoCompuesto: AC-DC
GrupoCompuesto: Guitarristas
Guitarra: Angus Young
Guitarra: Malcolm Young
Batería: Chris Slade
Cantante: Brian Johnson
Bajo: Cliff Williams
```

Estructura de objetos para The Ramones



Estructura de objetos para AC-DC



En el proyecto de Sistema francés:

Imaginemos que para nuestro Sistema francés de préstamos es necesario agregar una forma de búsqueda, ya que contamos con una cantidad grande de préstamos y se desea desea buscar los préstamos por estado o por fecha. Una opción sería agregar métodos con estos comportamientos, pero podría ocurrir que luego de unos días el gerente del banco requiere búsquedas por estado del préstamo. Más adelante vuelve con un nuevo requerimiento búsquedas por otro tipo de criterio, como montos del préstamo, cantidad de cuotas, que prestamos se encuentran deudores a una fecha o una combinación de cualquiera de los anteriores o el cruce de información que se les ocurra.

Teniendo en cuenta esto, si seguimos agregando nuevo métodos de búsqueda el sistema se volvería difícil de mantener, siempre modificando el comportamiento de clases y agregando funcionalidad sin saber exactamente cuándo se va a terminar. A fin de evitar estos problemas, usamos el patrón de diseño Composite para poder agrupar el comportamiento general de búsqueda en un clase superior y delegar las diferencias específicas a sus clases dependientes. De este modo podemos hacer que el sistema crezca con facilidad sin tener que modificar demasiado el contexto y siempre manteniendo una regla de comportamiento.

Vamos plantear cuatro tipos de búsquedas, por clientes, por estado, por “morosos” y por un período de tiempo.

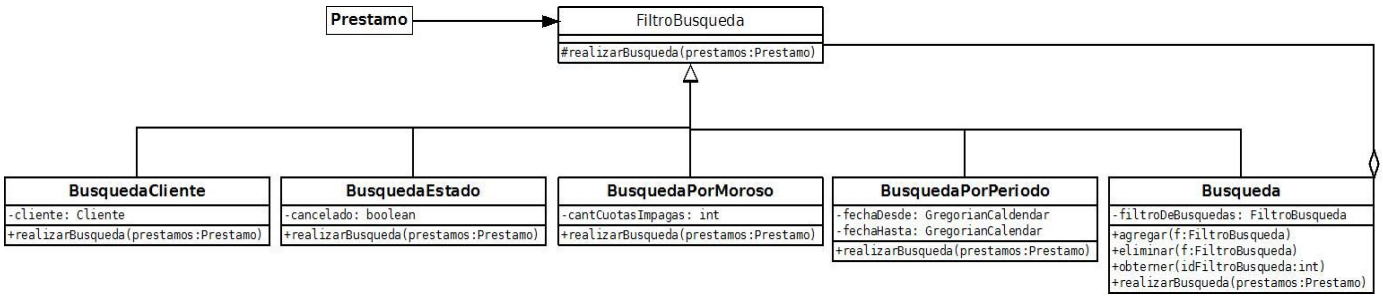
BusquedaCliente: Trae todos los préstamos de un cliente es específico.

BusquedaEstado: Trae todos los préstamos según el valor de cancelado, si cancelado es true significa que todas sus cuotas están pagas.

BusquedaPorMoroso: Trae todos los préstamos donde sus clientes sean morosos.

BusquedaPorPeriodo: Para el intervalo cerrado de dos fechas traer todos los préstamos cuya fecha de otorgamiento pertenezca al intervalo.

Utilizando Composite para resolver estos problemas, el diagrama de clases que planteamos es:



Preguntas:

1. ¿Cómo resolvería el diseño ante el requerimiento de los préstamos activo7s de un cliente?
2. ¿Cómo resolvería este diseño ante el requerimiento de conocer los préstamos morosos en una cuota en el mes de Marzo del 2018?

Referencias utilizadas:

Título: Patrones de Diseño

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

Editorial: Pearson Addison Wesley

SourceMaking https://sourcemaking.com/design_patterns