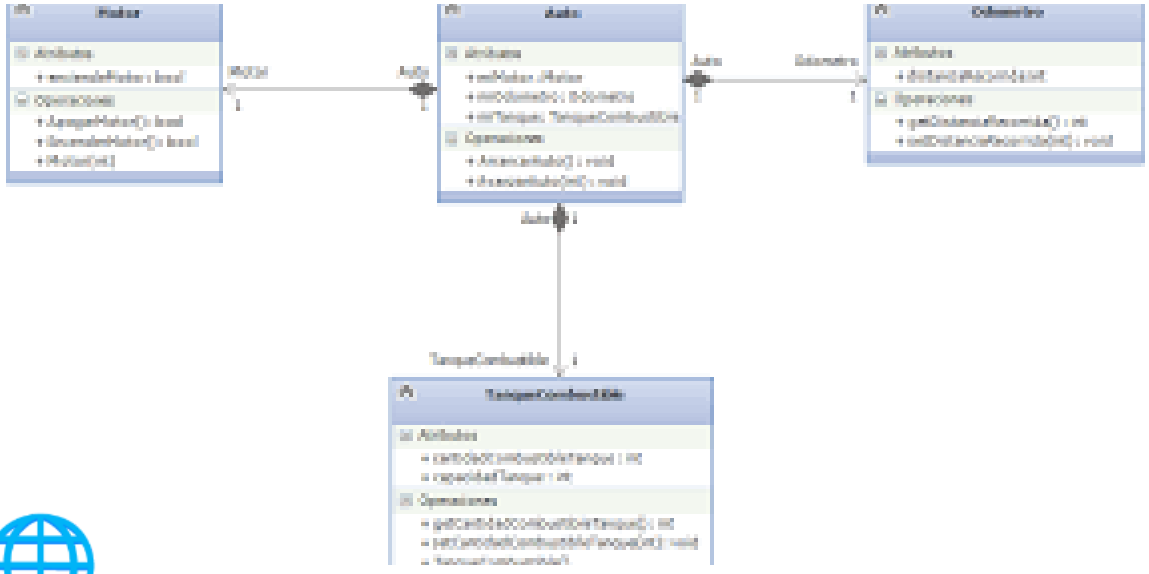

COMPOSICIÓN-DELEGACIÓN

> Cuando dos clases tienen métodos iguales o similares, especialmente si implementan el mismo tipo, es ideal reutilizar los métodos ya existentes para evitar duplicar el trabajo. Para usar los métodos de una clase en otra que implementa el mismo tipo, se puede usar **composición**: agregar objetos de una clase con los métodos deseados a la nueva clase, o crear una clase específica para contener esos métodos.

COMPOSICION:

La composición es una asociación fuerte entre una clase compuesta y una clase componente en la que instancias de la clase componente no suelen existir independiente de instancias de la clase compuesta.

> Cada método de la primera clase envía un mensaje al objeto de la segunda; esto es conocido como **delegación**.

DELEGACION:

La delegación es un mecanismo en programación por el cual cuando un objeto recibe un mensaje para realizar una operación, no la realiza él mismo, si no que la encarga a otro objeto.

El objeto que recibe el mensaje es un objeto compuesto, y el otro objeto al que se le delega el mensaje es un objeto componente

> La clase compuesta sólo necesita conocer el tipo de la clase componente para poder hacer la delegación. La composición y delegación permiten reutilizar código basándose exclusivamente en los tipos de los objetos.

COMPOSICION Y REUTILIZACION:

La composición y delegación es una forma de reutilización de código pues permite crear nuevas clases a partir de clases existentes.

> La **composición** reduce dependencias indeseables al basarse solo en las interfaces públicas, favoreciendo clases encapsuladas y enfocadas en una tarea específica. Esto mantiene las jerarquías de clases simples y manejables. Por ello, la composición promueve la **modularidad**, que es la organización de un sistema en módulos altamente cohesivos y con bajo acoplamiento.

COHESION:

La cohesión es la forma y el grado en el que las responsabilidades de una clase o de las clases contenidas en un paquete están relacionas unas con otras.

Cuando la cohesión es alta es mejor.

ACOPLAMIENTO:

El acoplamiento es la forma y el grado de interdependencia entre clases y entre paquetes.

Cuando el acoplamiento es bajo es mejor.

MODULARIDAD

La modularidad es una propiedad de las clases y paquetes cuando son altamente cohesivos y están poco acoplados.

PRESENTACION:

La implementación de **Car** de los miembros de **IElectric** se muestra a continuación. Vean que **Car** implementa los miembros de **IElectric** sin pedir colaboración a ninguna otra clase.

```
public class Car : IElectric
{
private bool isOn;
```

```
public Carv1(String model)
{
    this.Model = model;
    this.isOn = false;
}

public String Model { get; }

public Boolean IsOn
{
    get
    {
        return this.isOn;
    }
}

public void TurnOn()
{
    this.isOn = true;
}

public void TurnOff()
{
    this.isOn = false;
}
```

- En una clase como `Lamp`, que implementa la interfaz `IElectric`, podríamos reutilizar la lógica de la clase `Car` en lugar de repetir el código, siguiendo el principio **DRY** (Don’t Repeat Yourself). Este principio asegura que cada pieza de conocimiento tenga una única representación en el sistema, facilitando cambios consistentes y predecibles.
- Todos los objetos de tipo `IElectric`, como autos y lámparas, comparten la funcionalidad de encenderse y apagarse. Además, tanto en autos como en lámparas, la "llave" es un componente esencial que forma parte del objeto, permitiendo su encendido y apagado.

Sea una clase Switch definida así:

```
public class Switch
{
private Status status;
```

```
private enum Status
{
    On = 0,
    Off = 1
}

public Switch()
{
    this.status = Status.Off;
}

public bool IsOn
{
    get
    {
        return this.status == Status.On;
    }
}

public void Toggle()
{
    this.status = this.status == Status.On ? Status.Off : S
tatus.On;
    // La sentencia anterior es equivalente a:
    // if (this.status == Status.On)
    // {
    // this.status = Status.Off;
    // }
    // else
    // {
    // this.status = Status.On;
```

```
    // }
}
```

- La clase `Lamp` implementa la interfaz `IElectric`, lo que requiere agregar los métodos `TurnOn`, `TurnOff` y la propiedad `IsOn`. Dado que estas funciones también las realiza un interruptor, se puede agregar un objeto `Switch` en `Lamp` para manejar el encendido y apagado.

La instancia de `Switch` se almacena en la variable privada `onOff`, que se usa para implementar los métodos de la interfaz. Esta instancia puede crearse al declarar `onOff` o en el constructor de `Lamp`.

```
public class Lamp : IElectric
{
private Switch onOff = new Switch();
```

```
public Boolean IsOn
{
    get
    {
        return this.onOff.IsOn;
    }
}

public void TurnOn()
{
    if (!this.onOff.IsOn)
    {
        this.onOff.Toggle();
    }
}
```

```
o
public void TurnOff()
{
```

```
if (!this.onOff.IsOn)
{
this.onOff.Toggle();
}
}
}
```

- Los objetos de la clase `Lamp` están compuestos por objetos de la clase `Switch`, los cuales se crean y existen solo mientras el objeto `Lamp` existe. `Lamp` delega las operaciones de la interfaz `IElectric` en el objeto `Switch`, es decir, `Lamp` no realiza el trabajo directamente, sino que envía mensajes al `Switch` para que lo haga.
- El polimorfismo permite configurar dinámicamente tanto los datos como los algoritmos que se usarán para procesarlos. Por ejemplo, en un programa que ordena números, se puede seleccionar en tiempo de ejecución el algoritmo adecuado (como quicksort o heapsort) según las circunstancias, haciendo la aplicación más flexible y adaptable.

GUIAS PARA LA ASIGNACION DE RESPONSABILIDADES

> Un objeto compuesto puede delegar responsabilidades a un objeto componente, sin necesidad de conocer su clase, solo su tipo. Esto permite que el objeto compuesto utilice diferentes instancias de clases del componente, incluso en tiempo de ejecución, aportando flexibilidad. Al cambiar el componente, también cambia el comportamiento del compuesto, manteniendo principios clave como encapsulación, cohesión y bajo acoplamiento.

INVERSIÓN DE DEPENDENCIAS

- El principio de inversión de dependencias, introducido por Robert C. Martin, busca evitar interdependencias indeseadas entre clases, que provocan problemas como rigidez, fragilidad e inmovilidad del diseño. Para solucionar esto, Martin propone que las clases de alto y bajo nivel dependan de abstracciones (interfaces o clases abstractas), no de implementaciones concretas.
- En composición y delegación, esto significa que una clase compuesta debe depender solo del tipo del componente, no de su clase específica. Este principio es aplicable a cualquier

relación de dependencia entre clases, no solo en composición.

Martin dice que una de las causas de los malos diseños son las interdependencias indeseadas entre clases. Eso provoca:

- Que el diseño sea difícil de cambiar, porque cada cambio afecta muchas partes del programa -rigidez-.
- Cuando se hace un cambio, partes inesperadas del programa pueden dejar de funcionar -fragilidad-.
- Es más difícil usar clases de un programa en otro, porque no se pueden “desenredar” del programa actual -inmovilidad-.

CREATOR

- En composición y delegación, surge la pregunta de quién crea el objeto componente. Aunque normalmente es el objeto compuesto, no siempre es así. Para determinar qué clase debe crear un objeto, se asigna la responsabilidad a una clase B si cumple con alguna de estas condiciones: agrega, contiene, guarda, usa estrechamente o tiene los datos necesarios para crear objetos de la clase A. En estos casos, B es el "creador" de A.
- Si hay varias opciones, se prefiere la clase que agrega o contiene instancias de A.