

Patrones


▼ Creacionales

▼ Factory

Proporciona una interfaz para la creación de objetos en superclases, pero permite a las subclases alterar el tipo de objetos que se van a crear.

Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

 <https://refactoring.guru/design-patterns/factory-method>



REFACTORING
· GURU ·

▼ Abstract Factory

Permite producir familias de objetos relacionados sin especificar sus clases concretas.

Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

 <https://refactoring.guru/design-patterns/abstract-factory>




REFACTORING
· GURU ·

▼ Builder

Permite construir objetos complejos paso por paso. Habilita el producir diferentes tipos y representaciones de un objeto usando el mismo código de construcción. Se logra con la creación de una clase constructora que tiene todos los pasos para ese tipo de objeto.

Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

 <https://refactoring.guru/design-patterns/builder>



REFACTORING
· GURU ·

▼ Ejemplos y correcciones

▼ Problema del sandwich

```

public class Sandwich
{
    public string Bread { get; set; }
    public string Cheese { get; set; }
    public string Meat { get; set; }
    public string Vegetables { get; set; }
    public string Condiments { get; set; }

    public Sandwich(string bread, string cheese, string meat,
        string vegetables, string condiments)
    {
        Bread = bread;
        Cheese = cheese;
        Meat = meat;
        Vegetables = vegetables;
        Condiments = condiments;
    }

    public override string ToString()
    {
        return $"Sandwich with {Bread} bread, {Cheese} cheese,
            {Meat} meat, {Vegetables} vegetables, and {Condiments}
condiments.";
    }
}

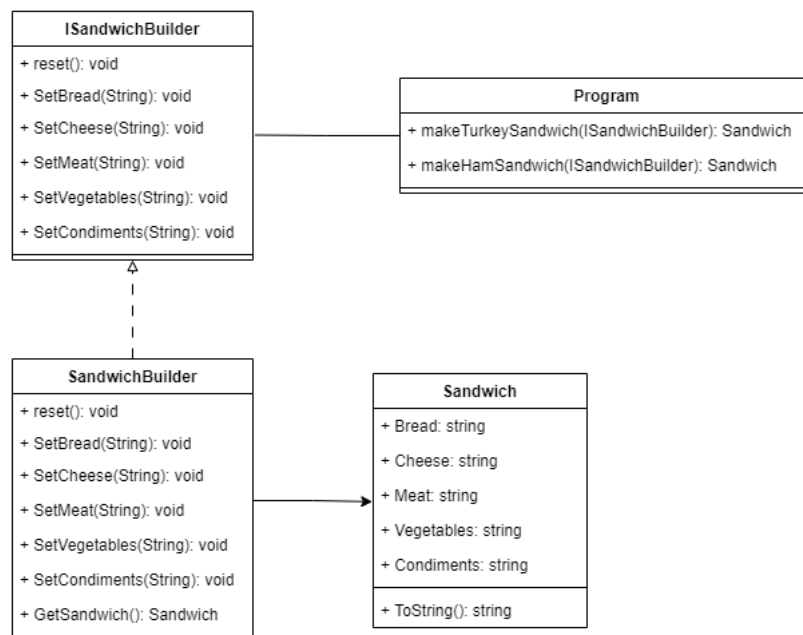
class Program
{
    static void Main(string[] args)
    {
        Sandwich hamSandwich = new Sandwich("White", "Swiss", "Ham",
            "Lettuce, Tomato", "Mayo, Mustard");
        Sandwich turkeySandwich = new Sandwich("Wheat", "Cheddar",
            "Turkey", null, "Mayo");

        Console.WriteLine(hamSandwich);
        Console.WriteLine(turkeySandwich);
    }
}

```

El patron builder se puede utilizar acá con el fin de facilitar la construcción del objeto sandwich y reducir el volumen de parametros que recibe el constructor y lograr así mayor legibilidad.

▼ UML



▼ Código

```

public class Sandwich
{
    public string Bread { get; set; }
    public string Cheese { get; set; }
    public string Meat { get; set; }
    public string Vegetables { get; set; }
    public string Condiments { get; set; }

    public Sandwich()
    {
    }

    public override string ToString()
    {
        return $"Sandwich with {Bread} bread, {Cheese} cheese, {Meat} meat, {Vegetables} vegetables, and {Condiments} condiments";
    }
}

public interface ISandwichBuilder
{
    void reset();
    void SetBread(string bread);
    void SetCheese(string cheese);
    void SetMeat(string meat);
    void SetVegetables(string vegetables);
    void SetCondiments(string condiments);
}

public class SandwichBuilder
{
    private Sandwich sandwich;

    SandwichBuilder()
    {
        this.reset();
    }

    void reset()
    {
        this.sandwich = new Sandwich();
    }

    void SetBread(string bread)
    {
        this.sandwich.Bread = bread;
    }

    void SetCheese(string cheese)
    {
        this.sandwich.Cheese = cheese;
    }

    void SetMeat(string meat)
    {
        this.sandwich.Meat = meat;
    }

    void SetVegetables(string vegetables)
    {
        this.sandwich.Vegetables = vegetables;
    }

    void SetCondiments(string condiments)
    {
        this.sandwich.Condiments = condiments;
    }

    Sandwich GetSandwich()
    {
        Sandwich sandwich = this.sandwich;
        this.reset();

        return sandwich;
    }
}

class Program
{
    static void Main(string[] args)
    {
        SandwichBuilder builder = new SandwichBuilder();
        makeHamSandwich(builder);
        Sandwich hamSandwich = builder.GetSandwich();
    }
}

```

```

        makeTurkeySandwich(builder);
        Sandwich hamSandwich = builder.GetSandwich();

        Console.WriteLine(hamSandwich);
        Console.WriteLine(turkeySandwich);
    }

    static Sandwich makeTurkeySandwich(ISandwichBuilder builder)
    {
        builder.SetBread("Wheat");
        builder.SetCheese("Cheddar");
        builder.SetMeat("Turkey");
        builder.SetVegetables(null);
        builder.SetCondiments("Mayo");
    }

    static Sandwich makeHamSandwich(ISandwichBuilder builder)
    {
        builder.SetBread("White");
        builder.SetCheese("Swiss");
        builder.SetMeat("Ham");
        builder.SetVegetables("Lettuce, Tomato");
        builder.SetCondiments("Mayo, Mustard");
    }
}


```

▼ Prototype

Permite copiar objetos existentes sin hacer que el código dependa de sus clases, delegando la responsabilidad de crearlos a las clases de dichos objetos.

Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

 <https://refactoring.guru/design-patterns/builder>



REFACTORING
· GURU ·

▼ Ejemplos y correcciones

▼ Problema GameUnit

```

public abstract class GameUnit
{
    public int Health { get; set; }
    public int Attack { get; set; }
    public int Defense { get; set; }

    // Simula la carga de recursos costosos como modelos 3D, texturas, etc.
    public virtual void LoadResources()
    {
        Console.WriteLine("Loading resources...");
    }
}

public class Archer : GameUnit
{
    public Archer()
    {
        LoadResources();
        Health = 100;
        Attack = 15;
        Defense = 5;
    }
}

public class Knight : GameUnit
{
    public Knight()
    {
        LoadResources();
        Health = 200;
        Attack = 20;
        Defense = 10;
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Creating original Archer...");
        Archer originalArcher = new Archer();

        Console.WriteLine("Copying Archers manually...");
        Archer copiedArcher1 = new Archer
        {
            Health = originalArcher.Health,
            Attack = originalArcher.Attack,
            Defense = originalArcher.Defense
        };

        Archer copiedArcher2 = new Archer
        {
            Health = originalArcher.Health,
            Attack = originalArcher.Attack,
            Defense = originalArcher.Defense
        };

        Console.WriteLine("Creating original Knight...");
        Knight originalKnight = new Knight();

        Console.WriteLine("Copying Knights manually...");
        Knight copiedKnight1 = new Knight
        {
            Health = originalKnight.Health,
            Attack = originalKnight.Attack,
            Defense = originalKnight.Defense
        };

        Knight copiedKnight2 = new Knight
        {
            Health = originalKnight.Health,
            Attack = originalKnight.Attack,
            Defense = originalKnight.Defense
        };
    }
}

```

Como se hacen copias de los objetos `Archer` y `Knight`, el patron que ma se puede adecuar para lograr de mejor forma lo que se quiere, es el prototype. Delegando la responsabilidad de clonar/copiar a las mismas clases de las que se necesita la copia, `Archer` y `Knight`

▼ UML


▼ Código

▼ Singleton

Asegura que una clase solo tenga una instancia, y a la vez provee un acceso global a la misma.

Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

 <https://refactoring.guru/design-patterns/singleton>

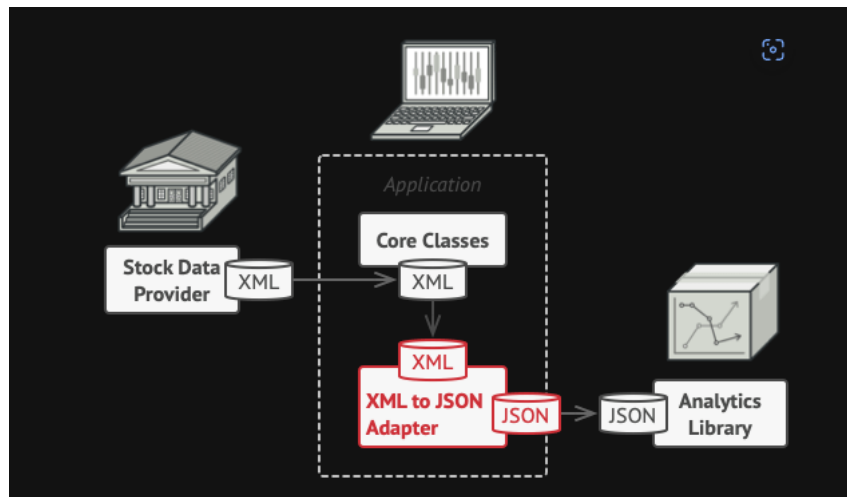


REFACTORING
· GURU ·

▼ Estructurales


▼ Adapter

Permite a los objetos con interfaces incompatibles colaborar. Lo puede hacer convirtiendo la data de uno en formatos que sean aceptados por el otro otro.



Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

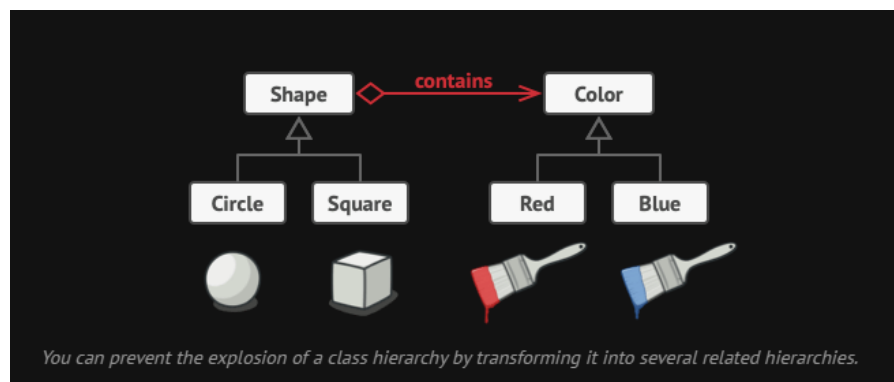
 <https://refactoring.guru/design-patterns/adapter>



REFACTORING
• GURU •

▼ Bridge

Permite dividir una clase grande o un conjunto de clases fuertemente relacionada en dos jerarquías separadas (abstracción e implementación) que podrán ser desarrolladas independiente la una de la otra.



Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

 <https://refactoring.guru/design-patterns/abstract-factory>



REFACTORING
• GURU •


▼ Composite

Permite componer objetos en estructuras de árbol y luego trabajar con estas estructuras como si fueran objetos individuales.

Usar este patron solo tiene sentido cuando el modelo de la app puede ser representado como un arbol.

Composite

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

 <https://refactoring.guru/design-patterns/composite>




REFACTORING
• GURU •

▼ Decorator

Permite adjuntar nuevos comportamientos a los objetos colocando estos objetos dentro de objetos envolventes especiales que contienen los comportamientos.

Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

 <https://refactoring.guru/design-patterns/builder>




REFACTORING
· GURU ·

▼ Facade

Provee una interfaz sencilla para una librería, framework, o algún otro conjunto complejo de clases.

Facade

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

 <https://refactoring.guru/design-patterns/facade>



REFACTORING
· GURU ·


▼ Flyweight

Permite ajustar mas objetos en la memoria RAM disponible compartiendo parte del estado entre los multiples objetos, en vez de mantener toda la data en cada uno.

La data inherent de cada objeto (la que es igual) es la que se suele compartir.

Flyweight

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

 <https://refactoring.guru/design-patterns/flyweight>



REFACTORING
· GURU ·

▼ Proxy

Permite brindar un sustituto de un objeto. Un proxy controla el acceso hacia un objeto, permitiendo ajustar cosas antes o despues de una request hecha al objeto original.

Proxy

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

 <https://refactoring.guru/design-patterns/proxy>



REFACTORING
· GURU ·

▼ Comportamiento

▼ Chain of Responsibility

Permite pasar una request a lo largo de una cadena de handlers. Una vez recibida una request cada handler decide si procesar la request o pasarla al siguiente handler de la cadena.

Chain of Responsibility

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

 <https://refactoring.guru/design-patterns/chain-of-responsibility>




REFACTORING
· GURU ·

▼ Command

Convierte una request en un objeto independiente. Esta transformacion permite pasar la request como un argumento de metodos, retrasar o poner en cola la ejecución de una solicitud, y admitir operaciones imposibles de hacer.

Command

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

 <https://refactoring.guru/design-patterns/command>




REFACTORING
· GURU ·

▼ Iterator

Permite recorrer elementos de una colección sin exponer su representación subyacente (pila, tress, linkedlist, etc)

Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

 <https://refactoring.guru/design-patterns/iterator>




REFACTORING
· GURU ·

▼ Mediator

Permite reducir dependencias caóticas entre objetos. Se restringe la comunicación directa entre los objetos, y se fuerza a que colaboran solo mediante el objeto mediador.

Mediator

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

 <https://refactoring.guru/design-patterns/mediator>




REFACTORING
· GURU ·

▼ Memento

Permite guardar y restaurar el estado previo de un objeto, sin revelar los detalles de su implementación.

Facade

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

 <https://refactoring.guru/design-patterns/facade>



REFACTORING
· GURU ·

▼ Ejemplos y correcciones

- ▼ Problema del GameCharacter


```

class Program
{
    static void Main()
    {
        var gameCharacter = new GameCharacter
        {
            Name = "John",
            Health = 100,
            Mana = 50
        };

        Console.WriteLine("Estado inicial:");
        gameCharacter.DisplayStatus();

        Console.WriteLine("\nGuardando estado...");
        var savedState = gameCharacter;

        Console.WriteLine("\nCambiando estados...");
        gameCharacter.Health -= 30;
        gameCharacter.Mana += 20;
        gameCharacter.DisplayStatus();

        Console.WriteLine("\nRestaurando estado...");
        gameCharacter = savedState;
        gameCharacter.DisplayStatus();
    }
}

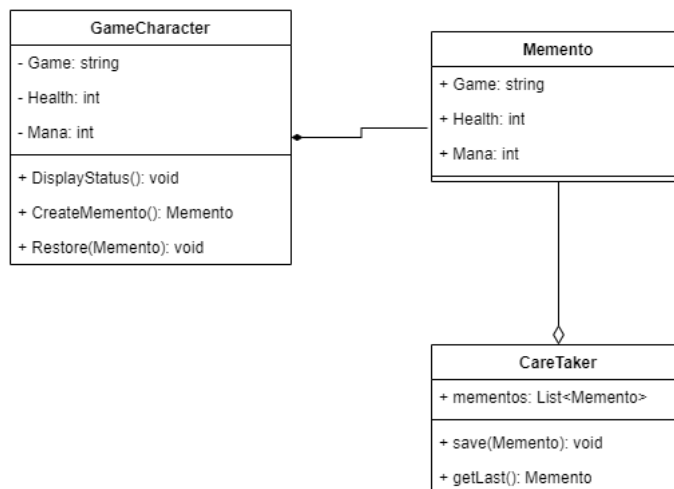
class GameCharacter
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int Mana { get; set; }

    public void DisplayStatus()
    {
        Console.WriteLine($"{Name} tiene {Health} de salud y {Mana} de mana.");
    }
}

```

El patrón que resuelve el problema es memento ya que se desea guardar estados, cambiarlos y restaurarlos. Y memento permite tener copias de objetos en estados anteriores.

▼ UML



La clase **Memento** es la que tiene guarda la información del objeto en un momento dado (Es de buena practica tener una clase inmutable como almacenadora). La clase **CareTaker** es la que almacena las distintas versiones del objeto a lo largo del tiempo, se encarga de manejar el historial de cambios.

▼ Código

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        var gameCharacter = new GameCharacter
        {
            Name = "John",
            Health = 100,
            Mana = 50
        };
    }
}

```

```

    };

    Console.WriteLine("Estado inicial:");
    gameCharacter.DisplayStatus();

    var caretaker = new Caretaker();
    // Guardar el memento
    caretaker.AddMemento(gameCharacter.CreateMemento());

    Console.WriteLine("\nCambiando estados...");
    gameCharacter.Health -= 30;
    gameCharacter.Mana += 20;
    gameCharacter.DisplayStatus();

    // Restaurar el memento
    gameCharacter.RestoreMemento(caretaker.GetLast());
    gameCharacter.DisplayStatus();
}
}

class GameCharacter
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int Mana { get; set; }

    public void DisplayStatus()
    {
        Console.WriteLine($"{Name} tiene {Health} de salud y {Mana} de mana.");
    }

    public Memento CreateMemento()
    {
        return new Memento(Name, Health, Mana);
    }

    public void RestoreMemento(Memento memento)
    {
        Name = memento.Name;
        Health = memento.Health;
        Mana = memento.Mana;
    }
}

class Memento
{
    public string Name { get; }
    public int Health { get; }
    public int Mana { get; }

    public Memento(string name, int health, int mana)
    {
        Name = name;
        Health = health;
        Mana = mana;
    }
}

class Caretaker
{
    private List<Memento> Mementos;

    public Caretaker()
    {
        Mementos = new List<Memento>();
    }

    public void AddMemento(Memento memento)
    {
        Mementos.Add(memento);
    }

    public Memento GetLast()
    {
        return Mementos.Last();
    }
}


```

▼ Observer

Permite definir un mecanismo de subscripción para notificar multiples objetos sobre cualquier evento que pase en el objeto que están observando.

Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

 <https://refactoring.guru/design-patterns/observer>



REFACTORING
GURU

▼ Ejemplos y correcciones

▼ Problema del examen

“Este código es bastante básico y no es escalable. Por ejemplo, si queremos notificar a más estudiantes o si queremos que los estudiantes se suscriban a las notificaciones de diferentes exámenes, este diseño no sería adecuado.”

```
class Program
{
    static void Main()
    {
        var exam = new Exam("Matemáticas");
        var student1 = new Student("Alice");
        var student2 = new Student("Bob");

        exam.NotifyStudents(student1, student2);
    }
}

class Exam
{
    public string Subject { get; }

    public Exam(string subject)
    {
        Subject = subject;
    }

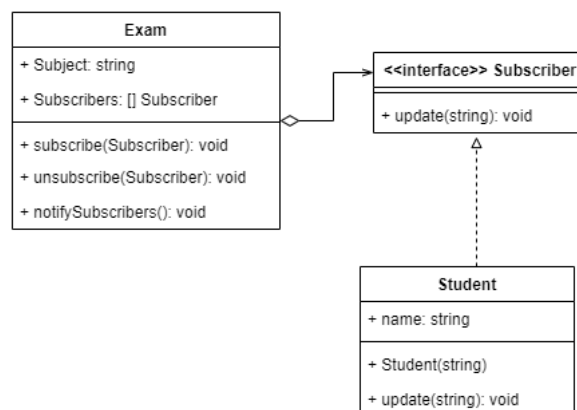
    public void NotifyStudents(params Student[] students)
    {
        foreach (var student in students)
        {
            Console.WriteLine($"{student.Name}, hay un nuevo examen de {Subject}!");
        }
    }
}

class Student
{
    public string Name { get; }

    public Student(string name)
    {
        Name = name;
    }
}
```

El patrón que resuelve el problema es observer ya que permite notificar al objeto Estudiantes y en base a eso actualizar su estado si corresponde. Además, teniendo una interfaz para aquellos objetos que pueden ser notificados, si en un futuro se agrega otro objeto, como profesor, al cual se le quiere notificar, no habría que cambiar código alguno del examen ni estudiante, simplemente extender agregando otra clase que implemente la interfaz correspondiente, por lo que cumple con OCP.

▼ UML



Eventualmente, la clase `Exam`, puede implementar una interfaz `IObservable` que tenga los metodos de `subscribe`, `notify`, y `unsubscribe`, asi como la lista de `subscribers`.

▼ Código

```
public interface Subscriber
{
    void Update(string subject);
}

public class Student : Subscriber
{
    private string name;

    public Student(string name)
    {
        this.name = name;
    }

    public void Update(string subject)
    {
        Console.WriteLine($"Student {name} received message: {subject}");
    }
}

public class Exam //Eventualmente puede implmentar la interfaz IObservable
{
    private List<Subscriber> subscribers = new List<Subscriber>();
    private string subject;

    public void Subscribe(Subscriber subscriber)
    {
        subscribers.Add(subscriber);
    }

    public void RemoveSubscriber(Subscriber subscriber)
    {
        subscribers.Remove(subscriber);
    }

    public void NotifySubscribers()
    {
        foreach (Subscriber subscriber in subscribers)
        {
            subscriber.Update(subject);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Exam exam = new Exam();
        Student student1 = new Student("Alice");
        Student student2 = new Student("Bob");

        exam.Subscribe(student1);
        exam.Subscribe(student2);

        exam.NotifySubscribers();
    }
}
```

▼ State

Permite a un objeto cambiar su comportamiento cuando su estado interno cambia. Aparenta como que el objeto cambio de clase.

State

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

 <https://refactoring.guru/design-patterns/state>



REFACTORING
GURU

▼ Ejemplos y correcciones

▼ Problema de la television

```

class Program
{
    static void Main()
    {
        var television = new Television();

        string input = "";
        while (input != "exit")
        {
            Console.WriteLine("Escriba 'on' para encender, 'off' para apagar, 'volumeup' para subir volumen, 'volumedown' para bajar volumen, 'exit' para salir.");
            input = Console.ReadLine();

            switch (input)
            {
                case "on":
                    television.TurnOn();
                    break;
                case "off":
                    television.TurnOff();
                    break;
                case "volumeup":
                    television.VolumeUp();
                    break;
                case "volumedown":
                    television.VolumeDown();
                    break;
            }
        }
    }
}

```

```

}

class Television
{
    private bool isOn = false;
    private int volume = 1;

    public void TurnOn()
    {
        isOn = true;
        Console.WriteLine("Televisión encendida.");
    }

    public void TurnOff()
    {
        isOn = false;
        Console.WriteLine("Televisión apagada.");
    }

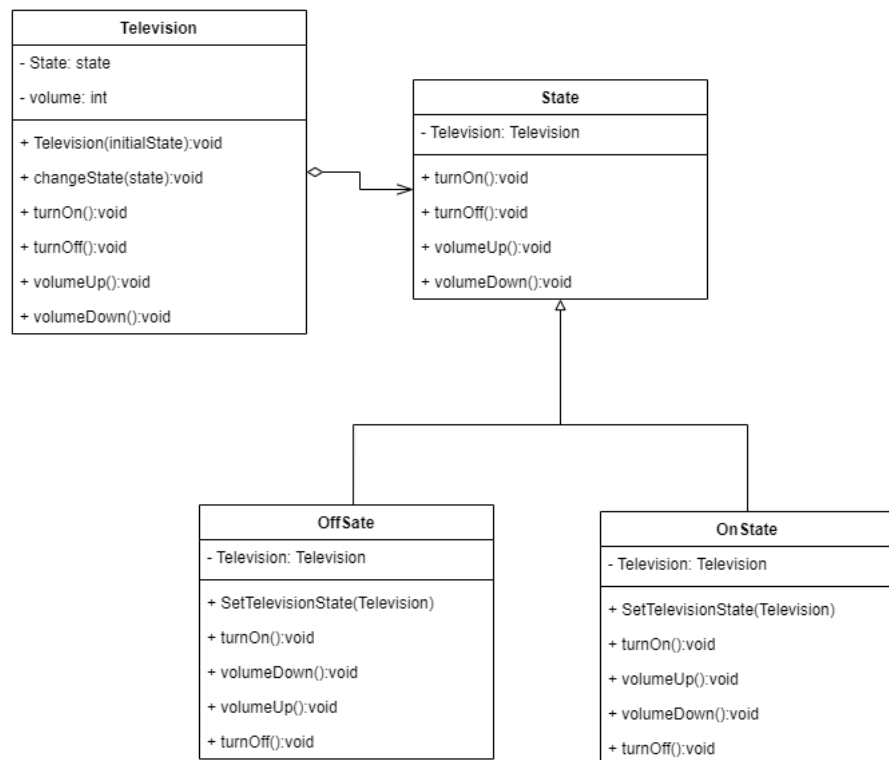
    public void VolumeUp()
    {
        if (isOn)
        {
            volume++;
            Console.WriteLine($"Volumen: {volume}");
        }
    }

    public void VolumeDown()
    {
        if (isOn)
        {
            volume--;
            Console.WriteLine($"Volumen: {volume}");
        }
    }
}

```

Es State porque en base a condiciones cambia el estado del objeto y el comportamiento del mismo.

▼ UML



El comportamiento de la `Television` cambia en base a si esta prendida o apagada, por lo que esos serían sus estados. En el caso de que tenga el estado de `OffState`, subir y bajar el volumen no hacen nada.

▼ Código

```
public class Television {
    private State state;
    private int volume;

    public Television() {
        this.state = new OffState(this);
        this.volume = 0;
    }

    public void changeState(State state) {
        this.state = state;
    }

    public void turnOn() {
        this.state.turnOn();
    }

    public void turnOff() {
        this.state.turnOff();
    }

    public void volumeUp() {
        this.state.volumeUp();
    }

    public void volumeDown() {
        this.state.volumeDown();
    }
}

abstract class State {
    protected Television tv;

    public State(Television tv) {
        this.tv = tv;
    }

    public abstract void turnOn();
    public abstract void turnOff();
    public abstract void volumeUp();
    public abstract void volumeDown();
}

class OnState extends State {
    public OnState(Television tv) {
        super(tv);
    }

    @Override
    public void turnOn() {}

    @Override
    public void turnOff() {
        tv.changeState(new OffState(tv));
    }

    @Override
    public void volumeUp() {
        tv.volume++;
    }

    @Override
    public void volumeDown() {
        tv.volume--;
    }
}

class OffState extends State {
    public OffState(Television tv) {
        super(tv);
    }

    @Override
    public void turnOn() {
        tv.changeState(new OnState(tv));
    }

    @Override
    public void turnOff() {}
}
```

```

@Override
public void volumeUp() {}

@Override
public void volumeDown() {}
}

public class Program {
    public static void main(String[] args) {
        Television tv = new Television();
        tv.turnOn();
        tv.volumeUp();
        tv.turnOff();
    }
}

```

▼ Strategy

Permite definir una familia de algoritmos, poner cada uno en una clase separada, u hacer cada objeto intercambiable.

Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

 <https://refactoring.guru/design-patterns/strategy>



REFACTORING
·GURU·

▼ Ejemplos y correcciones

▼ Problema del saludo

```

class Program
{
    static void Main()
    {
        GreetingSystem greetingSystem = new GreetingSystem();

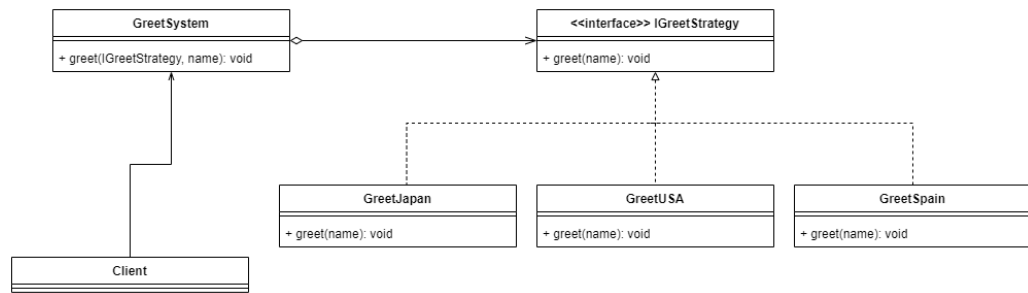
        greetingSystem.Greet("USA", "John");
        greetingSystem.Greet("Spain", "Juan");
        greetingSystem.Greet("Japan", "Yuki");
    }
}

class GreetingSystem
{
    public void Greet(string nationality, string name)
    {
        if (nationality == "USA")
        {
            Console.WriteLine($"Hello, {name}!");
        }
        else if (nationality == "Spain")
        {
            Console.WriteLine($"¡Hola, {name}!");
        }
        else if (nationality == "Japan")
        {
            Console.WriteLine($"こんにちは, {name}!");
        }
        else
        {
            Console.WriteLine("Nationality not supported.");
        }
    }
}

```

El patrón que soluciona es Strategy porque permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hace sus objetos intercambiables. Por lo que sería fácil agregar nuevas formas de saludo sin tener que cambiar el código existente.

▼ UML



Con strategy cada logica por tipo de saludo para a ser una clase que implemente la interfaz `IGreetStrategy` para tener el propio método de saludo que tendrá cada lógica en específico. Esto facilita agregar nuevos saludos, y tambien intercambiar los tipos de saludo de manera facil y eficiente, haciendolo mas mantenible.

▼ Código

```

using System;

public interface IGreetStrategy {
    void Greet(string name);
}

public class GreetJapan : IGreetStrategy {
    public void Greet(string name) {
        Console.WriteLine("こんにちは、 " + name);
    }
}

public class GreetUsa : IGreetStrategy {
    public void Greet(string name) {
        Console.WriteLine("Hello, " + name);
    }
}

public class GreetChina : IGreetStrategy {
    public void Greet(string name) {
        Console.WriteLine("你好, " + name);
    }
}

public class Greet {
    private IGreetStrategy greetStrategy;

    public Greet(IGreetStrategy greetStrategy) {
        this.greetStrategy = greetStrategy;
    }

    public void Greet(string name) {
        greetStrategy.Greet(name);
    }
}

class Program {
    static void Main(string[] args) {
        Greet greet = new Greet(new GreetJapan());
        greet.Greet("小明");
        greet = new Greet(new GreetUsa());
        greet.Greet("Tom");
        greet = new Greet(new GreetChina());
        greet.Greet("小明");
    }
}
  
```

▼ Template Method

Define un esqueleto de un algoritmo en la superclass, pero permite a las subclasses sobrescribir pasos específicos del algoritmo sin cambiar su estructura.

Template Method

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

 <https://refactoring.guru/design-patterns/template-method>




REFACTORING
· GURU ·

▼ Visitor

Permite separar algoritmos de los objetos en los que operan. Proporciona una forma de agregar nuevas operaciones a la estructura de objetos sin modificar las clases de esos objetos.

Visitor

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

 <https://refactoring.guru/design-patterns/visitor>



REFACTORING
· GURU ·

▼ Ejemplos y correcciones

▼ Problema de los animales

```
class Program
{
    static void Main()
    {
        Animal[] animals = { new Lion(), new Monkey(), new Elephant() };

        foreach (var animal in animals)
        {
            animal.Feed();
            // Nota: Con el tiempo, aquí tendrás que agregar más
operaciones, // lo que hará que el código sea menos mantenible.
        }
    }
}

abstract class Animal
{
    public abstract void Feed();
}

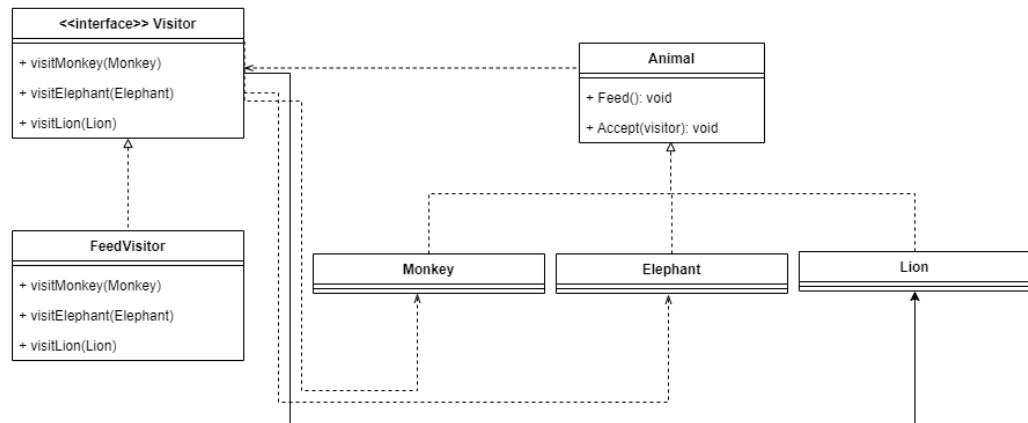
class Lion : Animal
{
    public override void Feed()
    {
        Console.WriteLine("El león está siendo alimentado con carne.");
    }
}

class Monkey : Animal
{
    public override void Feed()
    {
        Console.WriteLine("El mono está siendo alimentado con bananas.");
    }
}

class Elephant : Animal
{
    public override void Feed()
    {
        Console.WriteLine("El elefante está siendo alimentado con pastito
.");
    }
}
```

El patrón que resuelve es Visitor ya que este permite agregar funcionalidades sin tener que modificar `Animal`. Permite separar los algoritmos de los objetos en donde operan.

▼ UML



Eventualmente, si se quiere agregar otras funcionalidades como limpiar al animal, lo que se tendría que hacer sera crear otra clase `WashVisitor` que implemente `Visitor`, y en esa clase se hace la logica correspondiente de lavar a cada animal.

▼ Código

```

public interface Visitor {
    void visitMonkey(Monkey monkey);
    void visitElephant(Elephant elephant);
    void visitLion(Lion lion);
}

public class FeedVisitor implements Visitor {
    public void visitMonkey(Monkey monkey) {
        monkey.feed();
    }
    public void visitElephant(Elephant elephant) {
        elephant.feed();
    }
    public void visitLion(Lion lion) {
        lion.feed();
    }
}

public class WashVisitor implements Visitor {
    public void visitMonkey(Monkey monkey) {
        System.out.println("Monkey wash");
    }
    public void visitElephant(Elephant elephant) {
        System.out.println("Elephant wash");
    }
    public void visitLion(Lion lion) {
        System.out.println("Lion wash");
    }
}

abstract class Animal {
    public abstract void accept(Visitor visitor);
    public abstract void feed();
}

public class Monkey extends Animal {
    public void accept(Visitor visitor) {
        visitor.visitMonkey(this);
    }
    public void feed() {
        System.out.println("Monkey feed");
    }
}

public class Elephant extends Animal {
    public void accept(Visitor visitor) {
        visitor.visitElephant(this);
    }
    public void feed() {
        System.out.println("Elephant feed");
    }
}

public class Lion extends Animal {
    public void accept(Visitor visitor) {
        visitor.visitLion(this);
    }
}
  
```

```

    }
    public void feed() {
        System.out.println("Lion feed");
    }
}

public class Program {
    public static void main(String[] args) {
        Animal[] animals = new Animal[] {
            new Monkey(),
            new Elephant(),
            new Lion()
        };
        Visitor visitor1 = new FeedVisitor();
        Visitor visitor2 = new WashVisitor();
        for (Animal animal : animals) {
            animal.accept(visitor1);
            animal.accept(visitor2);
        }
    }
}

```