Patrones

1. Patrones de Creación: Se centran en la creación de objetos o clases.
2. Patrones Estructurales: Tratan de cómo se componen los objetos para formar estructuras más grandes.
3. Patrones de Comportamiento: Se centran en la interacción y responsabilidades entre objetos y clases

Qué son los antipatrones?

Los anti patrones, son soluciones ineficientes o contraproducentes para problemas comunes en el desarrollo de software o en otras áreas de la ingeniería. Estos patrones son considerados "anti" son soluciones que parecen ser válidas a primera vista, pero en realidad conducen a problemas, dificultades o resultados no deseados a largo plazo.

Los anti patrones pueden surgir debido a diversas razones, como la falta de comprensión de los requisitos, malas prácticas de diseño, falta de comunicación, falta de experiencia o simplemente por la presión de cumplir plazos ajustados. Algunos ejemplos comunes de anti patrones incluyen el "código espagueti" (spaghetti code), que se refiere a un código fuente desorganizado y difícil de mantener; el "dios objeto" (god object), que concentra demasiada funcionalidad en una sola clase.

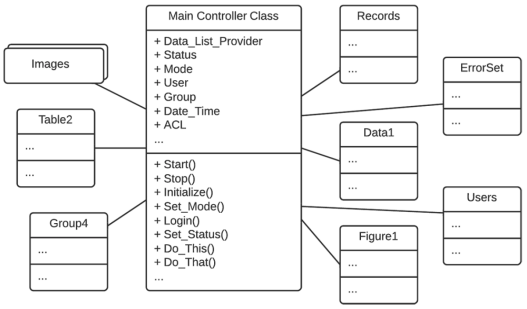
La identificación y comprensión de los anti patrones es importante en el campo del desarrollo de software, ya que permite a los desarrolladores reconocer y evitar soluciones ineficientes o problemáticas. Al aprender de estos patrones negativos, los profesionales pueden mejorar sus habilidades de diseño y desarrollo, y trabajar hacia la implementación de soluciones más efectivas y robustas.

The Blob

"The Blob", se refiere a una clase o componente que acumula una cantidad excesiva de responsabilidades y datos. Esto viola el principio de responsabilidad única y puede dificultar la comprensión, el mantenimiento y la extensibilidad del software.

Para evitar el antipatrón "The Blob", es importante aplicar principios de diseño sólidos, como la responsabilidad única y la separación de preocupaciones. Esto implica dividir las responsabilidades en clases o componentes más pequeños y especializados, cada uno con su propia área de responsabilidad claramente definida. Además, se pueden utilizar patrones de diseño como el principio SOLID, el patrón de diseño Composite o el patrón de diseño Decorator para mejorar la estructura y la gestión de la funcionalidad en el sistema.

Consecuencias de Usar The Blob
• Dificulta la mantenibilidad y comprensión del código.
• Reduce la reusabilidad de las clases.
• Puede afectar el rendimiento debido a la carga excesiva en una sola clase. • Cómo Evitar o Resolver The Blob
• Refactorizar: Dividir la clase gigante en clases más pequeñas y especializadas.
• Redistribuir responsabilidades entre clases basándose en principios de diseño orientado a objetos, como el Principio de Responsabilidad Única.
• Asegurar que cada clase tenga un propósito claro y bien definido.



Lava Flow

El antipatrón de diseño "Lava Flow" (Flujo de Lava) se refiere a la presencia de código muerto o código obsoleto en un sistema de software. Este antipatrón se produce cuando el código existente, que ya no es utilizado ni mantenido, se mantiene en el sistema sin ser eliminado.

El término "Lava Flow" se asemeja a la imagen de un flujo de lava que se solidifica y queda atrapado en el sistema, sin contribuir ni agregar valor, pero aún presente y ocupando espacio. El código muerto puede ser resultado de iteraciones anteriores del desarrollo del software, cambios en los requisitos o decisiones de diseño malas.

Los efectos negativos del "Lava Flow" incluyen:

1. Dificultad para entender el sistema: El código muerto puede confundir a los desarrolladores nuevos o incluso a los desarrolladores actuales, ya que no está claro si ese código es relevante o puede eliminarse.
2.
3. Aumento de la complejidad: El código muerto puede introducir complejidad innecesaria en el sistema, lo que dificulta el mantenimiento y las modificaciones futuras.
4.
5. Aumento del tamaño del código: El código muerto ocupa espacio en el sistema, lo que puede aumentar el tamaño del código fuente sin proporcionar ningún beneficio.

Para abordar el antipatrón "Lava Flow", es importante realizar una revisión y limpieza periódica del código base. Esto implica identificar y eliminar el código muerto o inactivo que ya no se utiliza. También es recomendable mantener un proceso de desarrollo ágil y seguir buenas prácticas de gestión de configuración para evitar la acumulación de código muerto en el futuro.

Eliminar el código muerto no solo ayuda a reducir la complejidad y mejorar la comprensión del sistema, sino que también facilita el mantenimiento y el desarrollo futuro al enfocar los esfuerzos en partes activas y relevantes del código.

Golden Hammer

• Definición: cuando un equipo de desarrollo o individuo depende demasiado de una herramienta o tecnología familiar, a expensas de posiblemente mejores alternativas.
• Características:
• Sobreutilización de una herramienta o tecnología en particular.
• Reluctancia para considerar o adoptar alternativas.
• Creencia de que la herramienta familiar

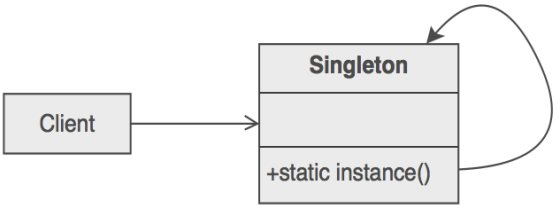
Spaghetti Code •
Definición: el código fuente de un programa tiene una estructura compleja y enredada, parecida a un plato de espaguetis, lo que lo hace difícil de leer, mantener y depurar. Características: – Falta de estructura y organización. – Uso excesivo de saltos incondicionales (GOTOs). – Difícil de seguir el flujo de control. – Ausencia de modularidad y encapsulación.

Cut−and−Paste P • Definición: se refiere a la práctica de copiar y pegar bloques de código dentro de una aplicación, en lugar de crear funciones o módulos reutilizables. • Características: • Duplicación de código. • Falta de modularidad y abstracción. • Dificultad en la implementación de cambios en lógicas duplicadas.

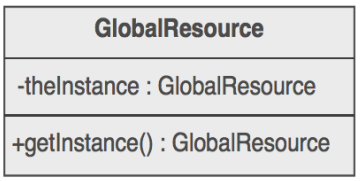
Design Patterns (Gang of Four) •
Son soluciones comunes a problemas relacionados con la creación de objetos en el diseño de software.
• Estos patrones se centran en proporcionar formas flexibles y reutilizables de crear y configurar objetos •
Los patrones creacionales abordan diferentes escenarios de creación de objetos, como garantizar una única instancia de una clase, crear objetos flexibles o construir objetos complejos paso a paso.

Singleton

Se utiliza para garantizar que una clase solo tenga una única instancia en todo el programa, proporcionando un punto de acceso global a esa instancia



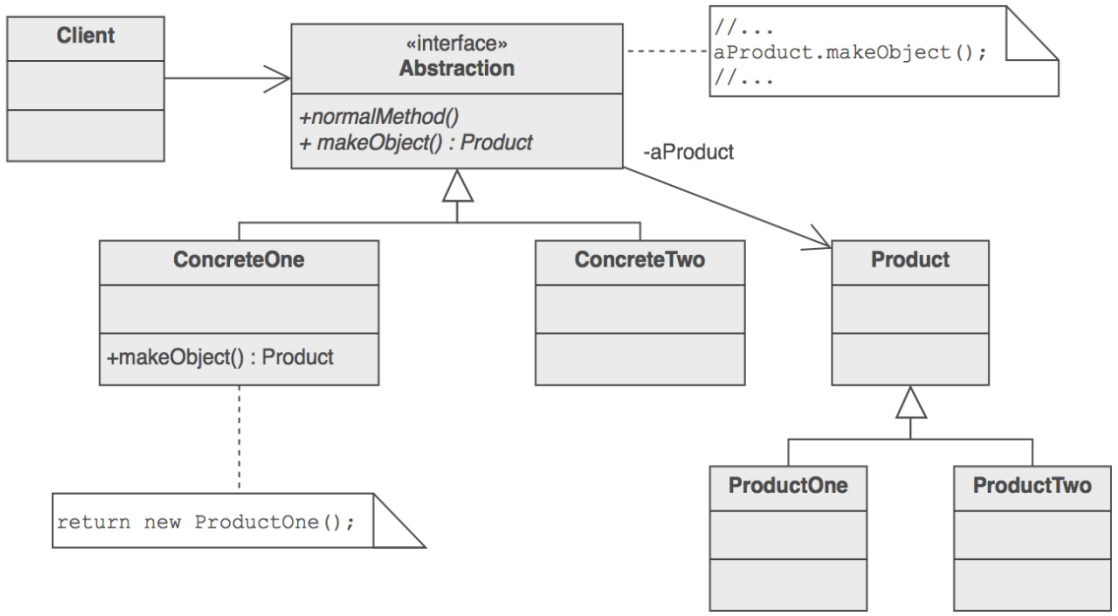
Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.



Factory Method

Se utiliza para encapsular la creación de objetos en una clase separada, permitiendo delegar la responsabilidad de la creación de objetos a la clase “Fábrica” Factory Method
• Proporciona una interfaz para crear objetos en una superclase y permite a las subclases alterar el tipo de objetos que se desean crear.
• Separa el código de construcción de producto del código que hace uso del producto.
• Por ello, es más fácil extender el código de construcción de producto de forma independiente al resto del código

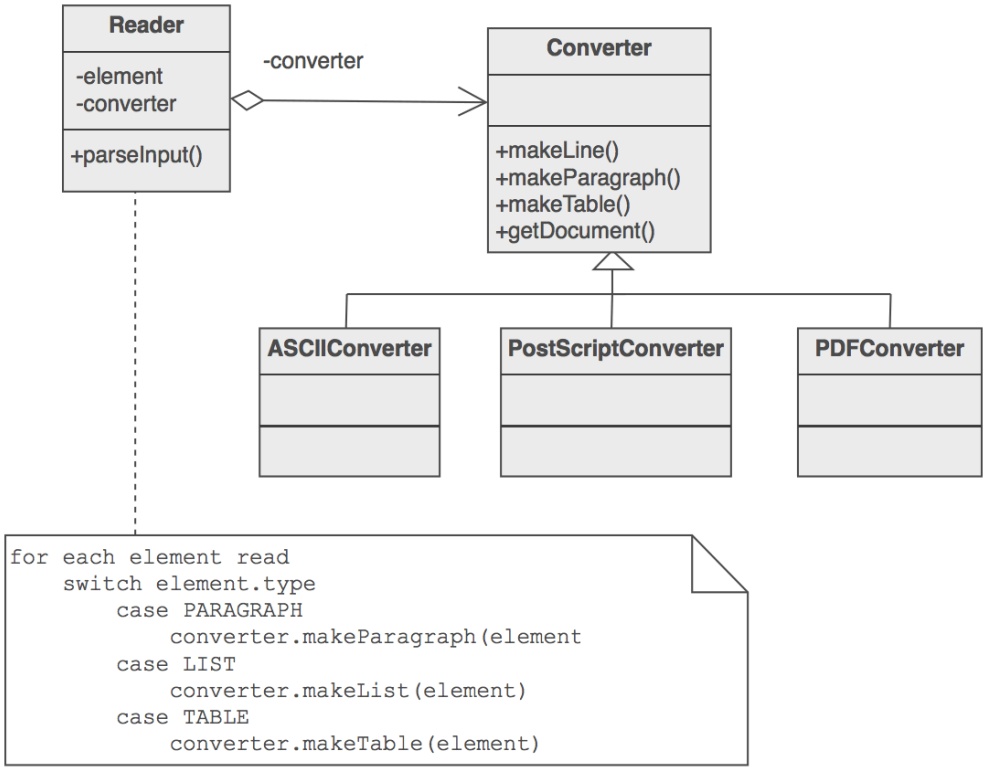
Factory Method (Usos) • Cuando se trabaja con jerarquías de clases y se necesita crear objetos polimórficos
• Cuando se busca desacoplar el código entre el cliente y las clases concretas
• Promueve la extensibilidad del código al permitir que nuevas subclases se agreguen fácilmente para crear nuevos tipos de objetos



Builder Permite construir objetos complejos paso a paso, permitiendo también producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

El Builder abstrae el proceso de construcción del objeto final de su representación interna.
• El Builder permite la creación de diferentes variaciones o configuraciones del objeto final
• Separa la lógica de construcción del objeto de la clase del objeto en sí.

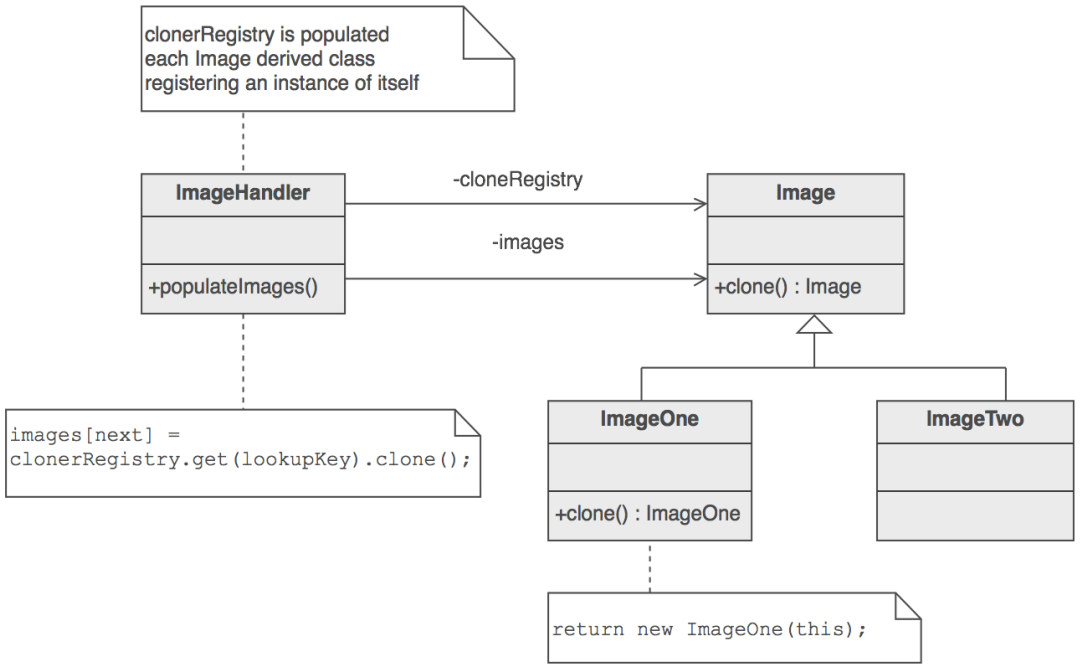
Builder (Usos) •
Es especialmente útil cuando existe un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados •
El patrón Builder facilita la modificación y ampliación del proceso de construcción •
Se suele puede aplicar cuando la construcción de varias representaciones de un producto requiera de pasos similares que sólo varían en los detalles.



Prototype
Se utiliza para crear nuevos objetos clonando un objeto existente, evitando la creación de objetos desde cero.

Permite crear nuevos objetos a través de la clonación de un objeto existente.
• Proporciona flexibilidad al permitir la creación de nuevos objetos con diferentes configuraciones o variantes sin tener que crear nuevas clases específicas para cada caso
• Permite modificar los atributos y comportamientos del objeto clonado sin afectar al objeto prototipo original.
• Algunos de los lenguajes lo implementan de forma nativa, por ejemplo: C# o C++

Cuando se busca evitar dependencia de clases concretas de objetos que se necesiten copiar. (Por ejemplo, objetos pasados por códigos de terceras personas a través de una interfaz)
• Cuando se necesitan gestionar diferentes estados o versiones de un “mismo” objeto, puede ser una alternativa a la creación de subclases
• Si la creación de un objeto es costosa, Prototype puede mejorar la eficiencia. En lugar de repetir el proceso de creación completo cada vez, se puede clonar un objeto ya creado



Prototype (Estructura)

• Debe existir una interfaz “Prototype” que declara los métodos de clonación, por ejemplo: clonar()

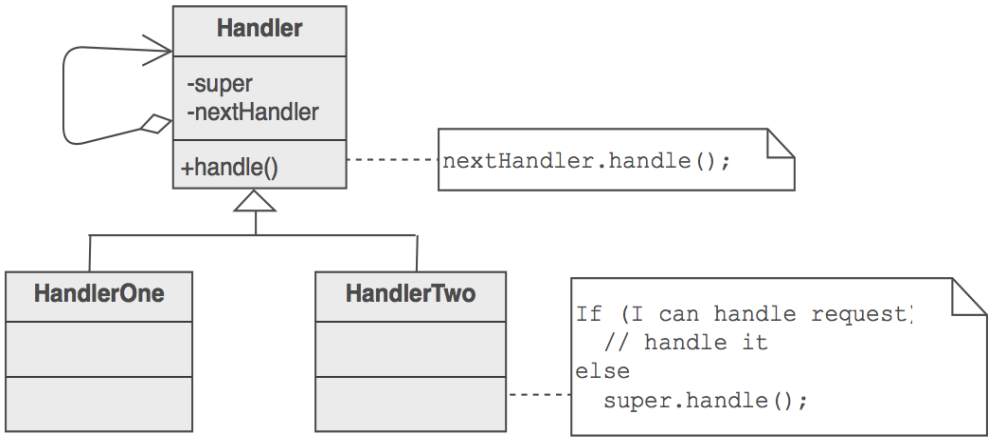
• Existe una clase que implementa “Prototype” que llamaremos “ConcretePrototype”

• La clase “ConcretePrototype” implementa el método clonar()

• Si el método clonar() copia las referencias de la clase “ConcretePrototype”, el clonado se denomina cómo superficial

• De lo contrario, si copia los objetos de las referencias, estamos ante una clonación profunda

Chain of responsability
Encapsula elementos de procesamiento dentro de una abstracción de “pipeline" y permite que los clientes envíen sus solicitudes a la entrada de esta ahí sin tener que gestionar cómo serán procesadas.
• En este patrón, los objetos que reciben la solicitud están encadenados entre sí. Una solicitud es pasada de un objeto a otro a lo largo de la cadena hasta que encuentra un objeto que es capaz de manejarla. – No es necesario saber de antemano el número y tipo de objetos manejadores, ya que pueden configurarse dinámicamente.
• Simplifica las interconexiones entre objetos. En lugar de que los emisores y receptores mantengan referencias a todos los receptores candidatos, cada emisor mantiene una sola referencia a la cabeza de la cadena, y cada receptor mantiene una sola referencia a su sucesor inmediato en la cadena.



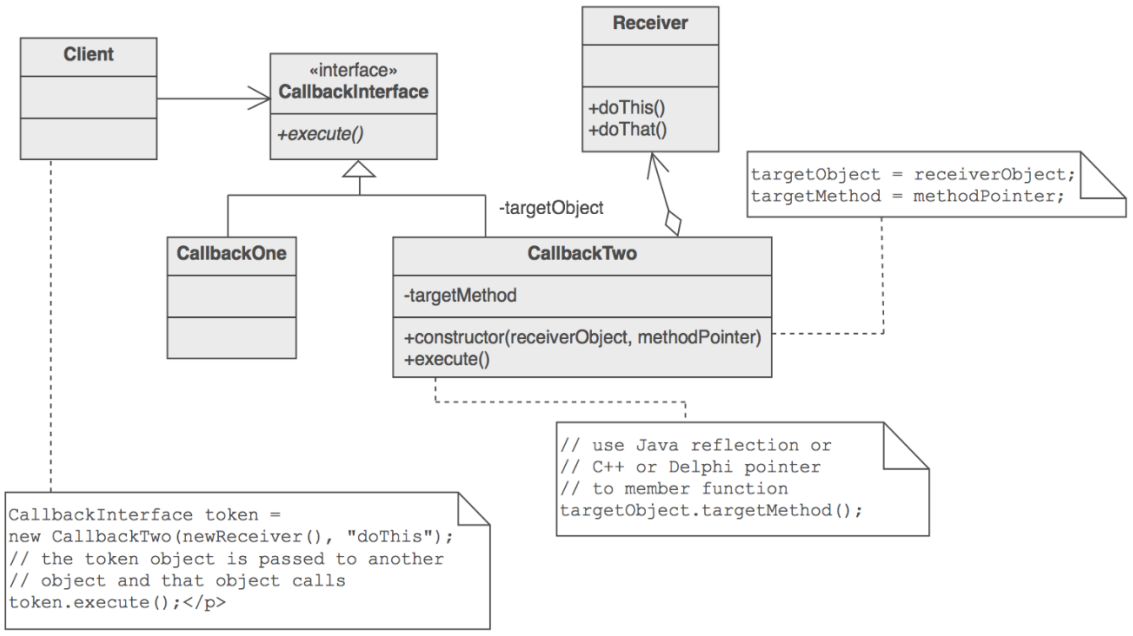
Command
Transforma una solicitud en un objeto independiente con toda la información sobre la solicitud. Esto permite parametrizar los objetos con operaciones, la idea central detrás del Patrón Command es la encapsulación de una solicitud de una acción a ser llevada a cabo en nombre de un objeto con los parámetros de esa acción.

Útil cuando necesitas desacoplar un objeto que invoca una operación del objeto que conoce cómo llevar a cabo dicha operación. • Cuando se quiere hacer cola de solicitudes, o se necesita mantener un historial de solicitudes.

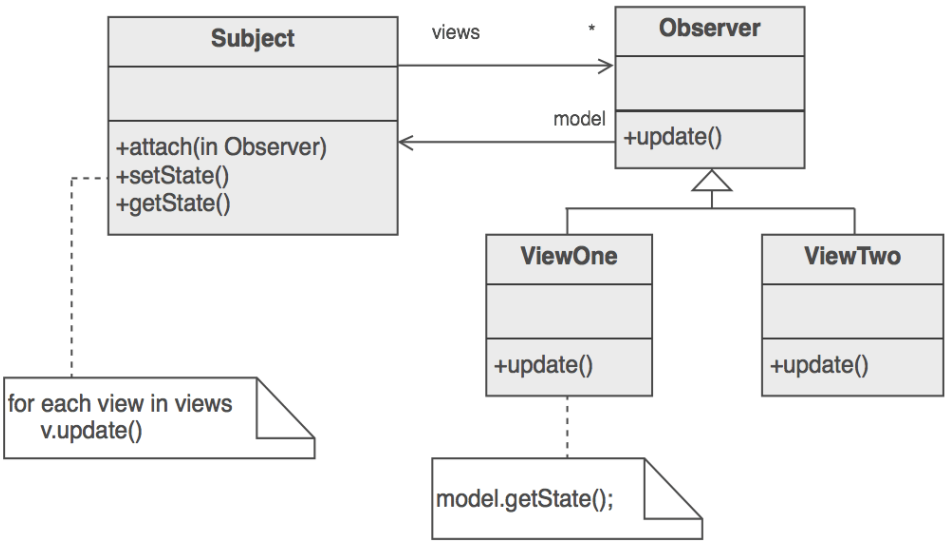
```
// Step 4: Create the invoker class
public class Invoker
{
    private ICommand _command;

    public void SetCommand(ICommand command)
    {
        _command = command;
    }

    public void ExecuteCommand()
    {
        _command.Execute();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Create instances
        Calculator calculator = new Calculator();
        ICommand add = new AddCommand(calculator, 10);
        ICommand subtract = new SubtractCommand(calculator, 5);

        // Create invoker
        Invoker invoker = new Invoker();

        // Execute commands
        invoker.SetCommand(add);
        invoker.ExecuteCommand();

        invoker.SetCommand(subtract);
        invoker.ExecuteCommand();
    }
}
}
```

```
using System;

namespace CommandPatternExample
{
    // Step 1: Create the command interface
    public interface ICommand
    {
        void Execute();
    }

    // Step 2: Create concrete command classes
    public class AddCommand : ICommand
    {
        private readonly Calculator _calculator;
        private readonly int _number;

        public AddCommand(Calculator calculator, int number)
        {
            _calculator = calculator;
            _number = number;
        }

        public void Execute()
        {
            _calculator.Add(_number);
        }
    }
```

```
public class SubtractCommand : ICommand
{
    private readonly Calculator _calculator;
    private readonly int _number;

    public SubtractCommand(Calculator calculator, int number)
    {
        _calculator = calculator;
        _number = number;
    }

    public void Execute()
    {
        _calculator.Subtract(_number);
    }
}

// Step 3: Create the receiver class
public class Calculator
{
    private int _value = 0;

    public void Add(int number)
    {
        _value += number;
        Console.WriteLine($"Added {number}, current value: {_value}");
    }

    public void Subtract(int number)
    {
        _value -= number;
        Console.WriteLine($"Subtracted {number}, current value: {_value}");
    }
}
```



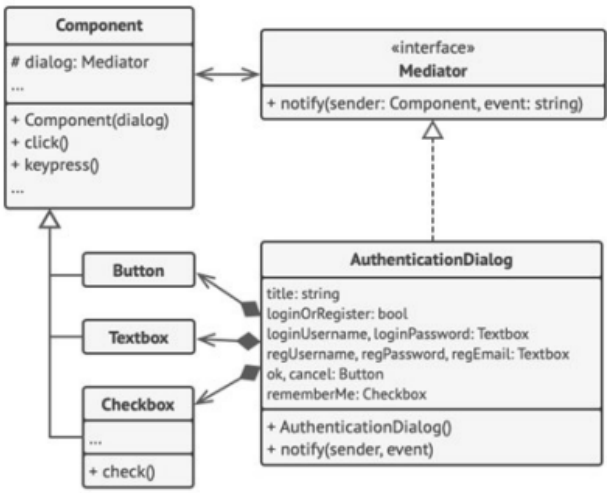
Observer • También conocido como Publisher-Subscriber, define una dependencia uno a muchos entre objetos. • Cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Esto es útil en situaciones donde un cambio en un objeto requiere cambios en otros objetos, sin saber cuántos objetos necesitan ser cambiados.



Mediator • Se utiliza para reducir la comunicación compleja entre objetos estrechamente relacionados. En lugar de que los objetos se comuniquen directamente entre sí, estos objetos interactúan a través de un objeto mediador central. • Esto es útil cuando tienes un conjunto de objetos que están estrechamente relacionados y que necesitan comunicarse entre sí de maneras complejas. En lugar de tener un número creciente de conexiones entre cada par de objetos, solo necesitas conectar cada objeto con el objeto mediador

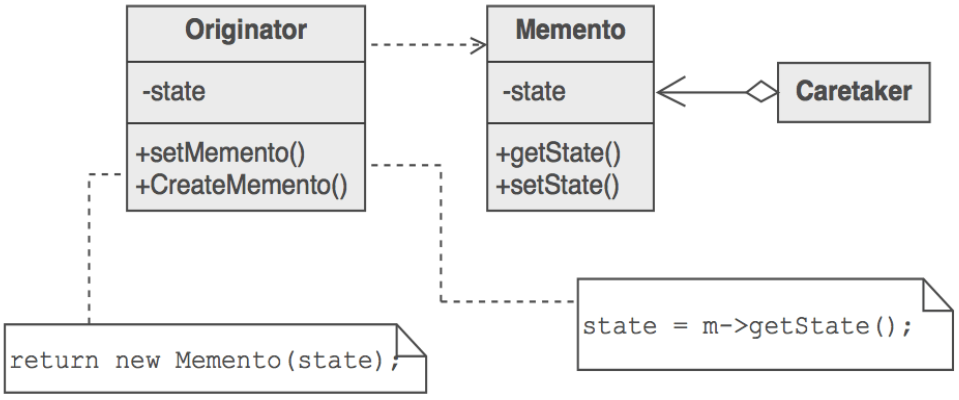
Los componentes principales del patrón Mediator son:

1. Mediator (Mediador): Es una interfaz que define cómo los objetos pueden comunicarse con el mediador.
2. ConcreteMediator (Mediador Concreto): Es una clase que implementa la interfaz Mediator y coordina la comunicación entre objetos colegas. Mantiene referencias a los objetos colegas y puede tener lógica compleja para coordinar estos objetos.
3. Colleague (Colega): Es una interfaz (o clase base) que define cómo los objetos colegas pueden comunicarse con el Mediador. A menudo, los objetos colegas tienen una referencia al mediador.
4. ConcreteColleague (Colega Concreto): Son clases que implementan la interfaz Colleague. Son los objetos que necesitan comunicarse entre sí, pero en lugar de comunicarse directamente, utilizan el Mediador



Memento
• Se utiliza para capturar el estado interno de un objeto en un punto en el tiempo, de modo que el objeto pueda ser restaurado a ese estado más tarde. Esto es útil en características como los sistemas de deshacer/rehacer en editores de texto, o para tomar instantáneas del estado de un sistema.

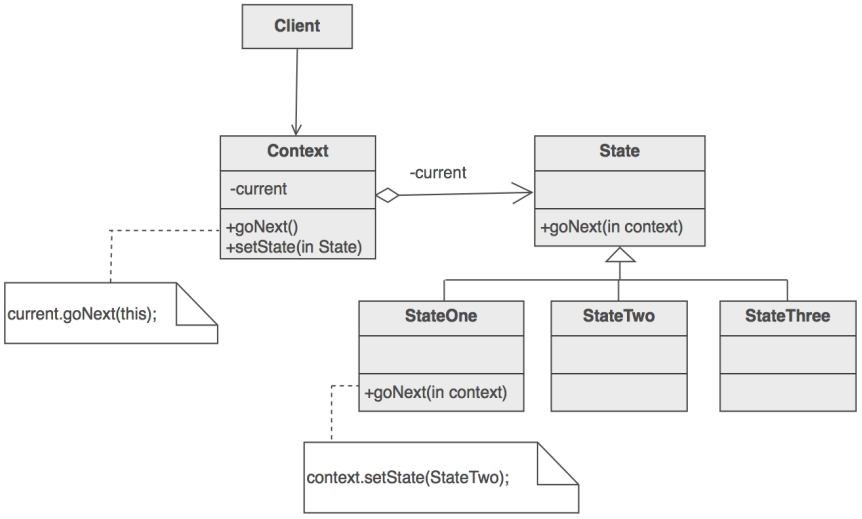
. Originator (Creador): Es el objeto cuyo estado queremos guardar y restaurar. Tiene un método que crea un memento conteniendo una instantánea de su estado actual, y otro método que restaura su estado a partir de un memento.
2. Memento (Recuerdo): Es un objeto que almacena el estado del Originator. Esencialmente, es una representación del estado interno del Originator, pero no debe permitir que ningún otro objeto modifique su contenido.
3. Caretaker (Custodio): Es responsable de mantener un registro de los diferentes estados del Originator mediante mementos. El Caretaker solo debe almacenar y recuperar mementos; no debe alterarlos ni examinar su contenido



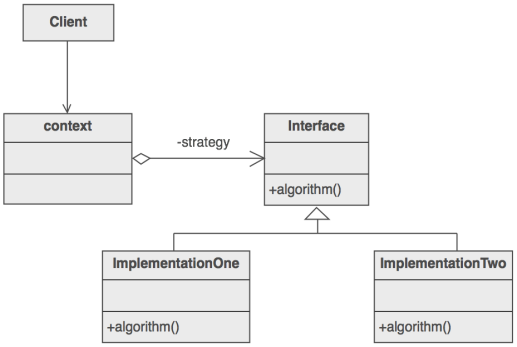
State
permite a un objeto cambiar su comportamiento cuando su estado interno cambia.
• Esto puede ser útil en situaciones donde un objeto debe cambiar su comportamiento de manera dinámica en tiempo de ejecución, en función de ciertas condiciones.
• En otras palabras, el patrón State sugiere que se cree una nueva clase para cada estado posible de un objeto, y que se extraiga el comportamiento específico de ese estado a esa clase.

Componentes del patrón State:

1. Contexto: Es la clase que tiene un estado. Contiene una referencia a una instancia de uno de los estados concretos y delega a ella el comportamiento que depende del estado.
2. State: Es una interfaz que define una interfaz común para todos los estados concretos. Define los métodos que deben implementar las clases de estado concreto.
3. Concrete States: Son las clases que implementan la interfaz State y definen el comportamiento asociado con un estado particular del Contexto.



Strategy •
Permite seleccionar un algoritmo o estrategia en tiempo de ejecución. En lugar de implementar un único algoritmo directamente dentro de una clase, el patrón Strategy utiliza interfaces para hacer que un conjunto de algoritmos sean intercambiables. • Componentes del patrón Strategy:
1. Strategy: Es una interfaz común a todos los algoritmos soportados. Declara un método que se utiliza para ejecutar un algoritmo.
2. Concrete Strategies: Son clases que implementan la interfaz Strategy. Cada una de ellas encapsula un algoritmo específico.
3. Context: Es la clase que contiene una referencia a una estrategia. Cambia la estrategia según sea necesario y delega la ejecución a la estrategia asociada.

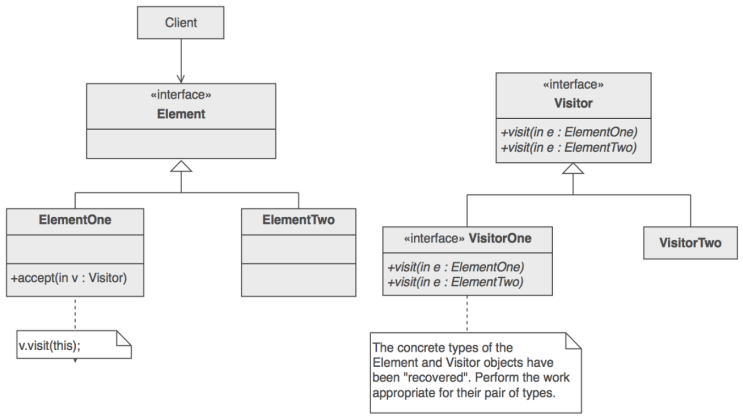


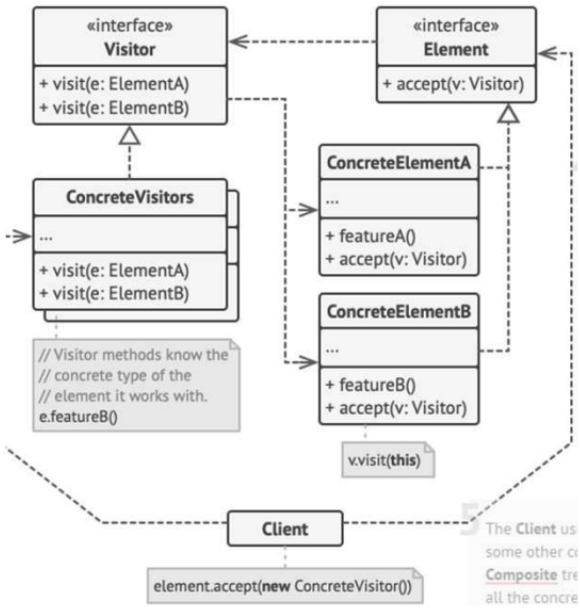
Visitor

Permite separar algoritmos de los objetos sobre los que operan. Esto puede ser útil cuando necesitan realizar operaciones sobre estos objetos sin alterar sus clases. •

Los componentes principales del patrón Visitor son:

1. Visitor: Es una interfaz que declara un conjunto de métodos de visita, uno para cada tipo concreto de elemento en la estructura de objetos. Cada método de visita acepta un único argumento, que es uno de los tipos de elementos de la estructura.
2. ConcreteVisitor: Estas son las clases que implementan la interfaz Visitor. Implementan cada uno de los métodos de visita definidos en la interfaz Visitor.
3. Element: Es una interfaz que declara un método accept que acepta un objeto de tipo Visitor como argumento.
4. ConcreteElement: Son las clases que implementan la interfaz Element. Implementan el método accept y, por lo general, tienen lógica adicional relacionada con el elemento.
5. Object Structure: Es una estructura de objetos que puede contener varios elementos ConcreteElement. Permite que los visitantes visiten los elementos.





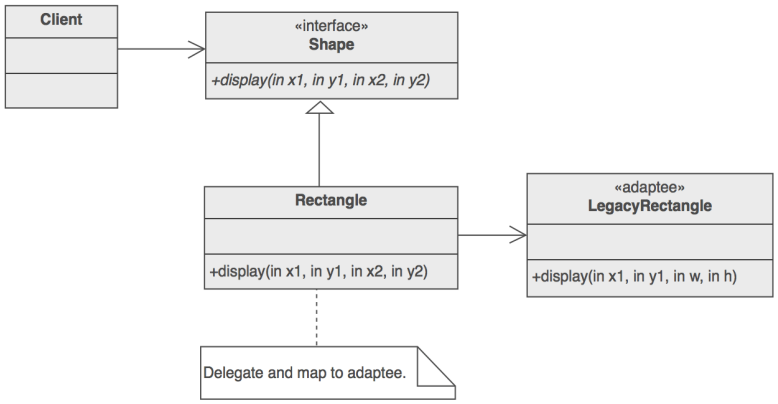
State, Strategy y Visitor

• Intención – State: Cambiar el comportamiento de un objeto basado en su estado interno. – Strategy: Permitir que un objeto tenga varios algoritmos o estrategias intercambiables. – Visitor: Agregar nuevas operaciones a clases sin modificarlas.

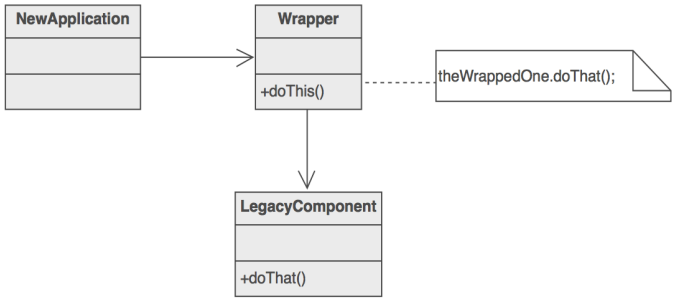
• Estructura – State: Contexto que mantiene referencia a un estado concreto.
• Interfaz de estado. Estados concretos que implementan la interfaz de estado. – Strategy: Contexto que mantiene referencia a una estrategia concreta.

• Interfaz de estrategia. Estrategias concretas que implementan la interfaz de estrategia.. – Visitor: Elementos que tienen un método accept para recibir visitantes.
• Interfaz de visitante. Visitantes concretos que implementan la interfaz de visitante.

Adapter • Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.
• A grandes rasgos, se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.



The Adapter could also be thought of as a "wrapper".

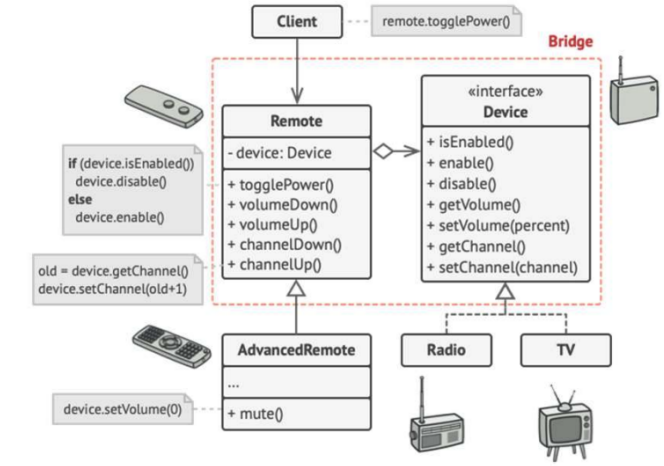


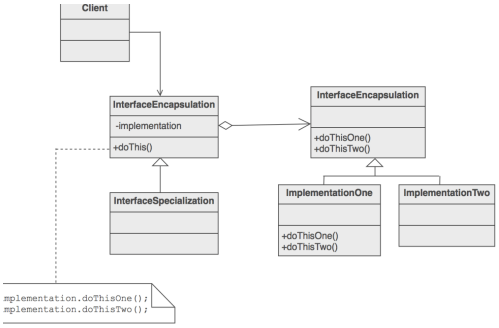
• Utiliza este patrón cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con otra parte del código.

Bridge

• Es un patrón de diseño estructural que permite dividir una clase, o un grupo de clases relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

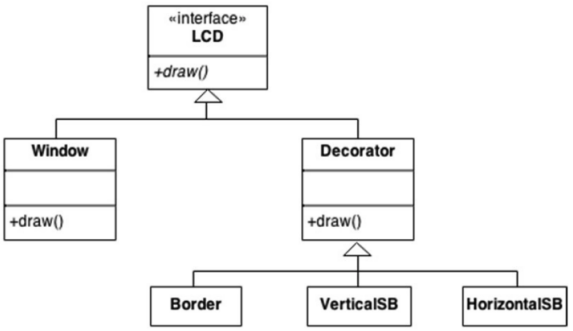
Puedes cambiar las clases de la GUI sin tocar las clases relacionadas con la API. Además, añadir soporte para otro sistema operativo sólo requiere crear una subclase en la jerarquía de implementación.
Utiliza el patrón Bridge cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).

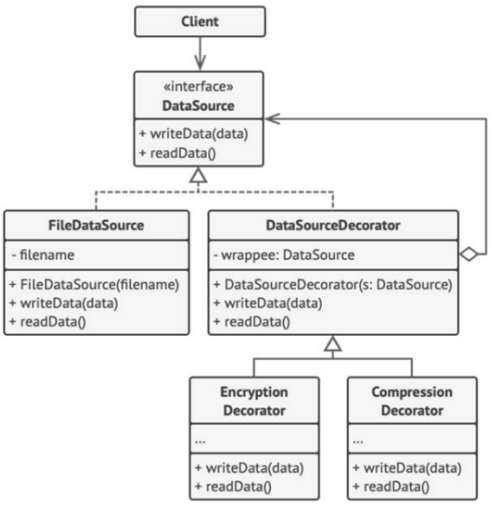




Decorator
• Es un patrón de diseño estructural que te permite añadir funcionalidades a objetos. Los decoradores proporcionan una alternativa flexible a la subclasificación para ampliar la funcionalidad
La solución consiste en encapsular el objeto original dentro de una interfaz contenedora abstracta. Tanto los objetos decoradores como el objeto central heredan de esta interfaz abstracta. La interfaz utiliza una composición recursiva para permitir que se agregue un

número ilimitado de "capas" de decoradores a cada objeto principal





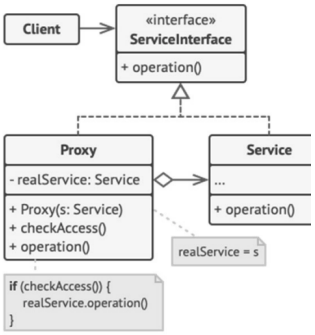
Facade

Proporcionar una interfaz unificada a un conjunto de interfaces en un subsistema. Facade define una interfaz de nivel superior que facilita el uso del subsistema. • Envuelve un subsistema complicado con una interfaz más simple.

Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.

Proxy

• Proporcione un sustituto o marcador de posición para otro objeto para controlar el acceso a él.
• Agregue un contenedor y una delegación para proteger el componente real de una complejidad indebida.
• Es utilizado para optimizar recursos. • El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el



trabajo.