Los diseñadores expertos suelen reutilizar soluciones anteriores que han funcionado en lugar de resolver cada problema desde cero. Esto los convierte en expertos. En sistemas orientados a objetos, estos patrones recurrentes de clases y objetos comunicantes ayudan a crear diseños más flexibles, elegantes y reutilizables. La idea de los "patrones de diseño", soluciones probadas que permiten a los diseñadores tomar decisiones más rápidamente y evitar errores comunes. Estos patrones no son nuevos, sino que representan soluciones exitosas aplicadas en múltiples sistemas, y el objetivo del libro es documentarlos de manera accesible.

¿Qué es un patrón de diseño?

Un patrón describe un problema recurrente en un entorno y luego ofrece una solución central que se puede aplicar repetidamente sin repetirla exactamente de la misma manera. En el contexto del diseño orientado a objetos, los patrones están relacionados con objetos e interfaces.

Un patrón tiene cuatro elementos esenciales:

- ★ **Nombre del patrón**: Sirve para describir el problema de diseño, su solución y consecuencias de manera concisa. Tener un nombre para el patrón facilita la comunicación y el razonamiento sobre el diseño.
- ★ **Problema**: Describe cuándo aplicar el patrón, explicando el problema de diseño específico y su contexto.
- ★ **Solución**: Expone los componentes de diseño, sus relaciones y colaboraciones. No se trata de un diseño concreto, sino de una plantilla abstracta que se puede aplicar en diferentes situaciones.
- ★ **Consecuencias**: Son los resultados y compensaciones de aplicar el patrón, ayudando a evaluar las alternativas y los costos/beneficios de su uso.

Este enfoque ayuda a identificar estructuras de diseño útiles y facilita la creación de diseños reutilizables.

Los siguientes son los patrones de diseño conocidos como GoF

		PROPÓSITO		
		CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
SCOPE	CLASE	Factory Method	Adapter	Interpreter Template Method
	OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Clasificamos los patrones de diseño según dos criterios. El primer criterio, llamado **propósito**, refleja lo que hace un patrón. Los patrones pueden tener un propósito creacional, estructural o conductual. Los patrones **creacionales** se ocupan del proceso de creación de objetos. Los patrones **estructurales** tratan con la composición de clases u objetos. Los patrones **conductuales** caracterizan las formas en que las clases u objetos interactúan y distribuyen responsabilidades.

El segundo criterio, llamado **ámbito (scope)**, especifica si el patrón se aplica principalmente a clases o a objetos. Los patrones de **clase** se ocupan de las relaciones entre las clases y sus subclases. Estas relaciones se establecen a través de la herencia, por lo que son estáticas y fijas en tiempo de compilación. Los patrones de **objeto** se ocupan de las relaciones entre objetos, las cuales pueden cambiar en tiempo de ejecución y son más dinámicas. Casi todos los patrones utilizan la herencia en cierta medida. Así que los únicos patrones etiquetados como "patrones de clase" son aquellos que se centran en las relaciones de clase. Cabe destacar que la mayoría de los patrones están en el ámbito de objeto.

Los **patrones creacionales de clase** delegan alguna parte de la creación de objetos a las subclases, mientras que los **patrones creacionales de objeto** delegan la creación a otro objeto. Los **patrones estructurales de clase** utilizan la herencia para componer clases, mientras que los **patrones estructurales de objeto** describen formas de ensamblar objetos. Los **patrones conductuales de clase** usan la herencia para describir algoritmos y el flujo de control, mientras que los **patrones conductuales de objeto** describen

cómo un grupo de objetos coopera para realizar una tarea que un solo objeto no puede llevar a cabo por sí mismo.

Patrones creacionales

	CREACIONAL
CLASE	Factory Method
OBJETO	Abstract Factory Builder Prototype Singleton

Son un tipo de patrón de diseño que **abstrae el proceso de creación de objetos**. Su objetivo es hacer que un sistema sea independiente de cómo se crean, componen y representan sus objetos. Estos patrones encapsulan el conocimiento sobre las clases concretas que el sistema utiliza y ocultan cómo se crean y ensamblan las instancias de esas clases. Esto proporciona flexibilidad sobre qué objetos se crean, quién los crea, cómo se crean y cuándo se crean.
Existen dos enfoques clave dentro de los patrones creacionales: **los que utilizan la herencia** para modificar la clase que se instancia y **los que delegan** la instanciación a otro objeto. Los patrones creacionales son especialmente útiles cuando un sistema depende más de la composición de objetos que de la herencia de clases.

Factory Method

El patrón **Factory Method** es un patrón creacional que **define una interfaz para la creación de objetos, pero permite que las subclases decidan qué clase concreta instanciar**. En lugar de llamar directamente al constructor de una clase, el cliente utiliza el método de fábrica, lo que permite que las subclases proporcionen diferentes implementaciones.
Componentes clave del patrón:

- ★ **Producto**: Define la interfaz de los objetos que el método de fábrica crea.
- ★ **Producto concreto**: Implementa la interfaz del producto.
- ★ **Creador**: Declara el método de fábrica, que devolverá un objeto de tipo producto.
- ★ **Creador concreto**: Sobrescribe el método de fábrica para devolver una instancia de un producto concreto.

Este patrón es útil cuando el código necesita trabajar con objetos de diferentes tipos, pero no se quiere acoplar el código con las clases concretas.
Se usa cuando no tenemos idea del tipo exacto de los objetos con los que vamos a trabajar.
Facilita la extensibilidad de la construcción del código del producto independientemente del resto de la aplicación; nos permite introducir nuevos productos sin romper el código existente.
Respetamos el principio "**open-closed**" ya que centralizamos la creación del código del producto en un lugar del programa.
El Factory Method es un patrón de clase porque utiliza herencia para delegar la responsabilidad de la creación de objetos a las subclases. En lugar de que la clase principal cree directamente instancias de objetos, el Factory Method **define un método en la clase base que puede ser sobrescrito por las subclases para determinar qué tipo de objeto específico debe ser creado**.

Abstract factory

Proporciona una interfaz para **crear familias de objetos relacionados o dependientes sin especificar sus clases concretas**. Su principal propósito es **permitir que un sistema sea independiente de cómo se crean estos objetos**, lo que facilita cambiar toda una familia de productos de manera sencilla.
Aplicabilidad: se usa cuando;

- ★ Un **sistema** debe ser **independiente** de cómo se **crean**, **componen**, y **representan**.
- ★ Un sistema debe configurarse con una de las múltiples familias de productos.
- ★ Una **familia de objetos** de productos relacionados **está diseñada para usarse en conjunto**, y es necesario hacer cumplir esta restricción.
- ★ Desea **proporcionar** una **biblioteca de productos de clase** y desea **revelar solo sus interfaces**, no sus implementaciones.

Componentes clave:

- ★ Abstract Factory: Declara una interfaz para crear cada uno de los objetos del producto.
- ★ Concrete Factory: Implementa la interfaz de la fábrica abstracta y produce objetos concretos de una familia específica.
- ★ Abstract Product: Declara una interfaz para los tipos de productos que la fábrica debe crear.
- ★ Concrete Product: Define los objetos concretos que se crearán, siguiendo la interfaz del producto abstracto.
- ★ Cliente: Utiliza únicamente las interfaces declaradas por la fábrica abstracta y los productos abstractos.

El uso de este patrón permite mantener la consistencia entre los objetos creados, dado que los productos de una misma familia están diseñados para trabajar juntos. Además, facilita el intercambio entre diferentes familias de productos sin modificar significativamente el código del cliente
Lo usamos cuando el código necesita trabajar con varias familias de productos relacionados pero no queremos depender del tipo concreto de las clases de esos productos.
Al usar este patrón, respetamos el principio "**open-closed**" y el principio de "**única responsabilidad**", ya que centralizamos la creación del código del producto en un lugar del programa.
El Abstract Factory es considerado un patrón de objeto porque **se enfoca en la creación de familias de objetos relacionados o dependientes y delega la responsabilidad de esa creación a objetos diferentes, no a la clase que lo invoca**. La creación de los objetos en el patrón Abstract Factory *no* se determina mediante herencia, sino a través de la composición de objetos, lo que permite una mayor flexibilidad en tiempo de ejecución.

Factory vs. Abstract

Factory Method	Abstract Factory
Patrón de clase; utiliza **herencia** para determinar qué clase concreta crear. Cada subclase puede sobrescribir el método de fábrica para crear su propio tipo de objeto. **Se enfoca en la creación de un solo tipo de objeto**.	Patrón de objeto; crea objetos mediante la **composición**. El cliente interactúa con una interfaz que encapsula la creación de una familia de objetos relacionados, y la implementación concreta de esa interfaz está delegada a un conjunto de fábricas concretas. **Se usa cuando se necesita crear familias de objetos relacionados**.

Builder

El Builder es un patrón creacional que **separa la construcción de un objeto complejo de su representación**, permitiendo que el mismo proceso de construcción pueda crear diferentes representaciones. Este patrón es especialmente útil cuando los objetos a crear son complejos y tienen muchas partes interdependientes.
Componentes clave del patrón Builder:

- ★ **Builder**: Especifica una interfaz abstracta para construir las partes de un producto.
- ★ **Concrete Builder**: Implementa la interfaz de Builder para construir y ensamblar las partes del producto concreto.
- ★ **Director**: Dirige el proceso de construcción utilizando el Builder.
- ★ **Product**: Representa el objeto complejo que se construye.

El Director se encarga de orquestar la construcción del objeto, mientras que el Builder establece los pasos para su construcción. Cada **Builder concreto puede crear diferentes tipos de productos complejos utilizando los mismos pasos de construcción**, lo que proporciona flexibilidad al proceso.
Cuando el tipo de un objeto comparte las mismas características (atributos), es ahí donde podemos introducir al director, que define el orden en el cual debemos llamar a los pasos de construcción para que podamos rehusar configuraciones específicas de los productos que estamos construyendo (son opcionales). Esconde los detalles de la construcción del producto.
Podemos rehusar el director pero no el builder, ya que este retorna objetos distintos, en cambio el director sigue una órden para construir el objeto.

Prototype

El patrón Prototype es un patrón creacional que se utiliza para **crear nuevos objetos copiando una instancia existente, conocida como prototipo**. En lugar de crear instancias de una clase a través de un constructor o una fábrica, **este patrón utiliza la clonación de un objeto existente para generar nuevos objetos**. Este enfoque es útil cuando la creación de un objeto es costosa o compleja y deseas evitar recrearlo desde cero.

Componentes clave del patrón Prototype:

- ★ **Prototipo**: Es una interfaz que declara el método clone() para duplicar el objeto. Este método devuelve una copia del objeto actual.
- ★ **Prototipo Concreto**: Implementa el método clone() para devolver una copia exacta del objeto. Cada clase concreta que se quiere clonar implementa este método.
- ★ **Cliente**: Crea nuevos objetos solicitando al prototipo que se clone a sí mismo en lugar de usar un constructor.

Ventajas del patrón Prototype:

Eficiencia: Evita la recreación desde cero de objetos costosos, lo cual es beneficioso cuando crear un nuevo objeto es más costoso que copiar uno existente.

Flexibilidad: Permite crear objetos dinámicamente a partir de un prototipo, lo que es útil si las clases que se deben instanciar no se conocen hasta el tiempo de ejecución.

Ejemplo de uso:

Este patrón se utiliza, por ejemplo, en sistemas que requieren crear muchos objetos con configuraciones similares o en sistemas que necesitan clones personalizados a partir de un estado inicial estándar.

Ahora que tenemos la posibilidad de clonar, también podemos acceder esos atributos declarados como privados.

Usamos a Prototype cuando nuestro código no debería depender de las clases concretas de los objetos que necesitamos copiar o duplicar.

Este patrón va a clonar objetos sin vincularlos a sus clases concretas; se va a deshacer del código de inicialización repetido.

Singleton

El patrón Singleton es un patrón creacional que **asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a esa instancia**. Este patrón es útil cuando solo debe existir un objeto de una clase en todo el sistema, como cuando se necesita un único gestor de recursos o un único manejador de configuración.

Componentes clave del patrón Singleton:

- ★ Clase Singleton:
  - Contiene una referencia estática a su única instancia.
  - Proporciona un método para acceder a esa instancia, generalmente llamado `getInstance()`.
  - El constructor de la clase es privado o protegido para evitar que se creen instancias adicionales.

Funcionamiento:

La primera vez que se llama a `getInstance()`, **la instancia de la clase se crea si no existe aún**. Las llamadas subsecuentes devolverán la misma instancia.

El constructor privado asegura que los clientes no puedan instanciar la clase directamente, lo que **impide la creación de múltiples instancias**.

Ventajas del Singleton:

Control de acceso global: Se garantiza que **solo existe una instancia del objeto**, lo cual es útil cuando se necesita un recurso centralizado.

Ahorro de memoria: Al evitar la creación de múltiples instancias, se reduce el consumo innecesario de recursos.

Patrones estructurales

Los patrones estructurales se ocupan de **cómo se componen las clases y objetos para formar estructuras más grandes**. Los patrones estructurales de clase utilizan la herencia para componer interfaces o implementaciones. Como ejemplo simple, considere cómo la **herencia múltiple** mezcla dos o más clases en una. El resultado es una clase que combina las propiedades de sus clases padre. Este patrón es particularmente útil para hacer que las bibliotecas de clases desarrolladas independientemente funcionen juntas. En lugar de componer interfaces o implementaciones, **los patrones estructurales de objeto describen formas de componer objetos para realizar nuevas funcionalidades**. La flexibilidad adicional de la composición de objetos proviene de la capacidad de cambiar la composición en tiempo de ejecución, lo cual es imposible con la composición estática de clases.

Adapter

El patrón Adapter es un patrón estructural que convierte la interfaz de una clase a otra interfaz que esperan los clientes. El adaptador permite que clases con interfaces incompatibles trabajen juntas.
El patrón Adapter es una solución de diseño que permite que clases con interfaces incompatibles trabajen juntas. Es como un **"traductor" que adapta la forma en que se comunican**.
Utiliza la **herencia y la composición** para permitir que objetos con interfaces incompatibles colaboren entre sí; para esto, crea una clase intermedia que sirve de “traductor”.
Respeta los principios de “**open-closed**” y “**única responsabilidad**”, el comportamiento de adaptación está separado, y podemos introducir nuevos adaptadores sin romper el código existente.

Aplicabilidad

Utiliza el patrón Adapter cuando:

- ★ Quieres usar una clase existente cuya interfaz no coincide con la que necesitas.
- ★ Quieres crear una clase reutilizable que colabore con clases no relacionadas o imprevistas, es decir, clases que no necesariamente tienen interfaces compatibles.
- ★ (solo adaptador de objeto) Necesitas usar varias subclases existentes, pero no es práctico adaptar su interfaz mediante la subclase de cada una. Un adaptador de objeto puede adaptar la interfaz de su clase padre.

Participantes

- ★ **Target**: Define la interfaz específica del dominio que el Cliente utiliza.
- ★ **Client**: Colabora con objetos que se ajustan a la interfaz Target.
- ★ **Adaptee**: Define una interfaz existente que necesita ser adaptada.
- ★ **Adapter**: Adapta la interfaz de Adaptee a la interfaz Target.

Adapter como patrón de diseño de clase vs. de objeto

De clase	De objeto
Utiliza **herencia** para crear un adaptador que extiende una clase existente (Adaptee) e implementa la interfaz que necesitas (Target).	Usa **composición** en lugar de herencia. Crea un adaptador que contiene una instancia del Adaptee y delega las llamadas a este.
Ambos enfoques permiten que una clase existente (Adaptee) se adapte a una nueva interfaz (Target), pero utilizan diferentes mecanismos (herencia para el adaptador de clase y composición para el adaptador de objeto). Uso según necesidad: Si se necesita **adaptar una sola clase y su comportamiento**, un adaptador de clase puede ser suficiente. Si se necesita **trabajar con múltiples instancias o subclases**, un adaptador de objeto sería la mejor opción.	

Bridge

El objetivo del patrón Bridge es **separar una abstracción de su implementación**, permitiendo que ambas evolucionen de manera independiente.
Cuando una abstracción puede tener múltiples implementaciones, una solución común es usar la herencia, pero esto puede crear problemas de rigidez. Si se hereda de una clase abstracta para implementar varias versiones, puedes acabar atado a una implementación específica, lo que dificulta extender o cambiar las implementaciones más adelante. El patrón Bridge soluciona este problema separando la abstracción de su implementación específica de plataforma. Así, **puedes cambiar la implementación sin afectar la abstracción**.
La abstracción delega el trabajo a la capa de implementación. La abstracción en el patrón Bridge puede ser representada a través de una cierta interfaz de usuario; y la implementación podría ser el código subyacente que esta interfaz de usuario utiliza en respuesta a las interacciones del usuario.
Participantes:

- ★ **Abstraction** (Window): Define la interfaz de la abstracción. Mantiene una referencia a un objeto de tipo Implementor.
- ★ **RefinedAbstraction** (IconWindow): Extiende la interfaz de Abstraction.
- ★ **Implementor** (WindowImp): Define la interfaz para las clases que implementan la funcionalidad concreta. Esta interfaz puede ser diferente a la de la abstracción.
- ★ **ConcreteImplementor** (XWindowImp, PMWindowImp): Implementa la interfaz Implementor y define la implementación concreta.

La clase Abstraction delega las solicitudes de los clientes a su objeto Implementor, que realiza la implementación concreta.
Consecuencias:
Desacoplamiento de la interfaz y la implementación: La abstracción no está vinculada permanentemente a una implementación. Es posible configurar la implementación en tiempo de ejecución e incluso cambiarla sin afectar a los clientes.
Mejora en la extensibilidad: Se puede extender tanto la abstracción como la implementación de manera independiente, lo que permite agregar nuevas funcionalidades sin crear muchas subclases.
Ocultación de detalles de implementación: Los detalles de la implementación, como el uso de múltiples objetos o mecanismos internos (como el conteo de referencias), pueden ocultarse de los clientes.
Aplicación:
El patrón Bridge es útil cuando:

- ★ Se quiere evitar una vinculación permanente entre una abstracción y su implementación.
- ★ Ambas (abstracción e implementación) deben poder extenderse por separado.
- ★ Cambios en la implementación no deberían afectar al código cliente.
- ★ Existen muchas subclases en una jerarquía debido a combinaciones de abstracciones e implementaciones.

Composite

El patrón Composite permite **componer objetos en estructuras de árbol para representar jerarquías de "parte-todo"**. Este patrón permite que los clientes traten objetos individuales y composiciones de objetos de manera uniforme. **Componer a los objetos en estructuras de árbol y luego poder trabajar con estas estructuras como si fueran objetos individuales.** Cobra sentido si podemos representar a los objetos en forma de árbol.
En aplicaciones gráficas, como editores de dibujo, los usuarios crean diagramas complejos a partir de componentes simples (como texto o líneas). Estos componentes pueden agruparse para formar componentes más grandes. Un problema de implementar esto es que el código debe tratar de manera distinta los objetos primitivos (como líneas) y los objetos contenedores (como grupos de líneas). El patrón Composite **resuelve este problema al usar la composición recursiva, permitiendo a los clientes interactuar con todos los objetos de manera uniforme, sin distinguir entre primitivos y compuestos**.

Participantes:

- ★ **Component** (Graphic):
  - Define la interfaz común para todos los objetos (primitivos y compuestos).
  - Puede declarar operaciones por defecto y manejar operaciones relacionadas con los hijos.
- ★ **Leaf** (Rectangle, Line, Text):
  - Representa los objetos "hoja" en la composición, es decir, los **elementos individuales que no tienen hijos**.
  - Implementa las operaciones necesarias para estos objetos primitivos.
- ★ **Composite** (Picture):
  - Representa un **objeto compuesto que puede contener otros objetos**.
  - Implementa operaciones tanto para gestionar los objetos hijos como para delegarles las operaciones que necesita realizar.
- ★ **Client**:
  - Manipula los objetos de la composición a través de la interfaz del componente (Graphic), sin distinguir si son objetos individuales o compuestos.

Consecuencias:

Jerarquías de clases: El patrón Composite define jerarquías de clases donde los objetos primitivos pueden ser compuestos para formar objetos más complejos. Esto permite que, en donde se espera un objeto primitivo, también pueda utilizarse un objeto compuesto.

Simplifica el código del cliente: Los clientes pueden tratar los objetos individuales y los compuestos de manera uniforme, sin necesidad de diferenciar entre ellos. Esto reduce la complejidad del código y evita el uso de condicionales.

Facilidad para agregar nuevos componentes: Es sencillo añadir nuevas clases de componentes (tanto hojas como compuestos) sin modificar el código del cliente. Las nuevas subclases de Composite o Leaf funcionarán automáticamente con las estructuras existentes.

Posible generalización excesiva: El patrón Composite hace que sea fácil agregar nuevos componentes, pero puede dificultar la restricción de qué componentes son permitidos en una composición. No se puede confiar en el sistema de tipos para imponer restricciones, por lo que a veces es necesario implementar validaciones en tiempo de ejecución.

Decorator

El patrón Decorator permite **agregar responsabilidades adicionales a un objeto de forma dinámica**. Los decoradores ofrecen una **alternativa flexible a la herencia para extender la funcionalidad.**

A veces queremos agregar funcionalidades a objetos individuales sin modificar la clase completa. Un ejemplo clásico es una interfaz gráfica de usuario, donde puedes agregar bordes o barras de desplazamiento a componentes de interfaz.

Usar la herencia para lograr esto es muy rígido, ya que se decide qué características se añaden de manera estática (en el momento de definir la clase). Por ejemplo, si decides heredar una clase para agregar un borde, todos los objetos de esa clase tendrán borde, sin posibilidad de control dinámico.

El patrón Decorator ofrece una solución más flexible: en lugar de modificar la clase, se encierra el objeto en otro que añade la funcionalidad extra. Este **"decorador" sigue la misma interfaz del objeto original, por lo que los clientes no saben si están trabajando con un objeto decorado o no**. Esto permite apilar decoradores para añadir múltiples funcionalidades.

Aplicaciones:

- ★ Agregar responsabilidades dinámicas a objetos individuales de forma transparente (sin afectar otros objetos).
- ★ Para habilitar y deshabilitar funcionalidades en objetos de manera flexible.
- ★ Cuando la herencia no es práctica porque requeriría muchas subclases para combinar todas las posibles funcionalidades.

Estructura:

- ★ **Component** (VisualComponent): Define la interfaz que los objetos decorables deben implementar.
- ★ **ConcreteComponent** (TextView): Es el objeto original al que se le pueden agregar responsabilidades adicionales.

- ★ **Decorator**: Es la clase abstracta que contiene una referencia al Component y sigue su interfaz para que los decoradores puedan usarse de manera intercambiable con el objeto original.
- ★ **ConcreteDecorator** (BorderDecorator, ScrollDecorator): Son las clases concretas que agregan responsabilidades adicionales.

Colaboraciones:
El decorador reenvía las peticiones al objeto que está decorando y puede realizar operaciones adicionales antes o después de reenviar la solicitud (por ejemplo, dibujar un borde o manejar el desplazamiento).
Consecuencias:

- ★ **Más flexibilidad que la herencia estática**: El Decorator permite agregar responsabilidades a los objetos en tiempo de ejecución sin crear nuevas subclases. Esto evita la proliferación de clases (como BorderedScrollableTextView), y permite combinar decoradores para mezclar distintas funcionalidades.
- ★ **Evita clases con demasiadas características**: En lugar de diseñar una clase con todas las posibles funcionalidades (que puede volverse muy compleja), el patrón Decorator permite definir una clase sencilla y añadirle funcionalidades incrementales según sea necesario.
- ★ No son idénticos: Desde el punto de vista de la identidad de los objetos, **un objeto decorado no es igual al objeto original**. Es decir, si confías en la identidad de los objetos, podrías tener problemas.
- ★ **Muchos objetos pequeños**: Un diseño con Decorators tiende a generar sistemas con muchos objetos pequeños que se parecen entre sí, lo que puede complicar la depuración y el aprendizaje del sistema.

Facade

El patrón Facade **proporciona una interfaz unificada a un conjunto de interfaces de un subsistema,** facilitando su uso al simplificar la interacción con este.
Dividir un sistema en subsistemas ayuda a reducir la complejidad general. Sin embargo, para que estos subsistemas sean más fáciles de manejar, se busca minimizar la comunicación y las dependencias entre ellos. El patrón Facade introduce un **objeto fachada** que **actúa como una interfaz simplificada para los clientes, ocultando los detalles complejos del subsistema.**
Por ejemplo, en un entorno de programación que da acceso a un subsistema de compilación, los componentes internos como Scanner, Parser, ProgramNode, y otros, son útiles para aplicaciones especializadas. Sin embargo, la mayoría de los usuarios simplemente quiere compilar código, sin preocuparse por los detalles internos del compilador. Aquí, *el patrón Facade crea una clase Compiler que actúa como una fachada, proporcionando una interfaz unificada y sencilla para los clientes*.
Aplicaciones:
El patrón Facade es útil cuando:

- ★ Quieres **ofrecer una interfaz simple a un subsistema complejo**, especialmente cuando este ha crecido y se ha vuelto más complicado. **Una fachada ofrece una visión predeterminada y simplificada del subsistema**, siendo suficiente para la mayoría de los clientes.
- ★ Existen muchas dependencias entre los clientes y las clases del subsistema. La **fachada reduce estas dependencias**, promoviendo la independencia y portabilidad del subsistema.
- ★ Quieres **organizar los subsistemas en capas**. Cada subsistema puede tener su propia fachada que actúa como punto de entrada para interactuar con las clases internas, simplificando las dependencias entre ellos.

Estructura:

- ★ **Facade** (Compiler): Conoce qué clases del subsistema son responsables de cada solicitud y delega las peticiones de los clientes a los objetos del subsistema correspondientes.
- ★ **Clases del subsistema** (Scanner, Parser, ProgramNode, etc.): Implementan la funcionalidad del subsistema y manejan el trabajo asignado por la fachada. Estas clases no tienen conocimiento directo de la fachada.

Colaboraciones:
Los clientes interactúan con el subsistema a través de la fachada, que envía las solicitudes a los objetos apropiados del subsistema. Aunque los objetos del subsistema realizan el trabajo, la fachada puede necesitar realizar conversiones para que las interfaces del subsistema coincidan con la interfaz que ofrece.
Los clientes que usan la fachada no necesitan interactuar directamente con los objetos del subsistema.

Consecuencias:

- ★ **Simplificación para los clientes**: El patrón Facade reduce la complejidad al ocultar los detalles internos del subsistema, facilitando el uso al manejar menos objetos.
- ★ **Acoplamiento débil**: Promueve un acoplamiento débil entre los clientes y el subsistema. Esto permite modificar los componentes internos del subsistema sin afectar a los clientes, ya que estos solo interactúan con la fachada. Esto es útil, por ejemplo, para minimizar el tiempo de recompilación en sistemas grandes cuando se hacen cambios en una parte importante del subsistema.
- ★ **No limita el acceso directo al subsistema**: El uso de la fachada no impide que las aplicaciones accedan directamente a las clases del subsistema cuando lo necesiten. De esta forma, los desarrolladores pueden optar entre facilidad de uso y flexibilidad según sus necesidades.

Es una clase que sirve como una interfaz frontal que enmascara el código estructural subyacente complejo. Mejora la legibilidad y usabilidad de una biblioteca de software al ocultar la interacción de sus componentes. Aplica el principio de responsabilidad única; define puntos de entrada a cada nivel de un subsistema, desacoplando así múltiples subsistemas y **obligándolos a comunicarse solo a través de fachadas**.

Flyweight

El patrón Flyweight es utilizado para **compartir objetos cuando se necesita manejar un gran número de instancias similares** de manera eficiente, permitiendo el uso compartido de esos objetos en lugar de crear nuevas instancias para cada uno.
Este patrón es útil cuando se necesita manejar un gran número de objetos pequeños y finamente detallados, como caracteres en un procesador de texto o iconos en una interfaz gráfica. Crear y gestionar cada objeto por separado puede resultar muy costoso en términos de memoria.
Ejemplo: se tiene un programa que necesita mostrar miles de caracteres en la pantalla. Cada carácter podría representarse como un objeto separado, pero esto consumiría una gran cantidad de memoria. **En lugar de crear un objeto por cada carácter, el patrón Flyweight permite reutilizar instancias ya existentes de objetos comunes** (como letras) y solo almacenar su estado intrínseco (como la forma de la letra) una vez, mientras que el estado extrínseco (como la posición en la pantalla) se almacena por separado.
Componentes:

- ★ **Flyweight**: Es una interfaz para los objetos que pueden ser compartidos.
- ★ **ConcreteFlyweight**: Implementa la interfaz y define los datos intrínsecos (los que pueden ser compartidos).
- ★ **UnsharedConcreteFlyweight**: A veces, algunos objetos no se pueden compartir. En esos casos, esta clase actúa como un flyweight no compartido.
- ★ **FlyweightFactory**: Es responsable de crear y gestionar los objetos flyweight. Se asegura de que los objetos compartidos sean reutilizados adecuadamente.
- ★ **Cliente**: Mantiene referencias a los objetos flyweight y define el estado extrínseco (el que cambia para cada objeto).

Aplicabilidad:
El patrón Flyweight se aplica cuando:

- ★ Se necesita manejar un gran número de objetos que comparten datos comunes.
- ★ Es necesario minimizar el uso de memoria, compartiendo objetos similares.
- ★ Los objetos pueden dividirse en estado intrínseco (que se comparte) y estado extrínseco (que varía).

Consecuencias:

- ★ Ahorro de memoria: Al compartir objetos comunes, el patrón reduce significativamente la cantidad de memoria usada.
- ★ Complejidad adicional: La gestión de estados intrínsecos y extrínsecos puede hacer que el diseño sea más complicado.

El patrón Flyweight es ideal cuando necesitas manejar un gran número de objetos similares de manera eficiente, minimizando el uso de memoria al compartir la mayor cantidad posible de información entre los objetos.

Proxy

El patrón Proxy **proporciona un sustituto o representante para otro objeto con el fin de controlar el acceso a él**.
Una razón para controlar el acceso a un objeto es **posponer su creación y ahorrar recursos hasta que realmente se necesite**. Por ejemplo, en un editor de documentos que puede incrustar objetos gráficos, algunos, como las imágenes grandes, son costosos de crear. No es necesario cargar todas las imágenes al abrir el documento, especialmente si no todas se verán al mismo tiempo.
Para manejar esto, se utiliza un proxy de imagen, que actúa como un sustituto de la imagen real y **se encarga de instanciarla solo cuando es necesaria**, es decir, cuando debe mostrarse. El proxy **funciona como una especie de envoltorio** para la imagen, almacenando su nombre de archivo y otras propiedades (como sus dimensiones), y luego, **cuando es solicitado para dibujar la imagen, instancia el objeto real.**
Este proxy evita que el editor de documentos se complique con los detalles sobre si la imagen ya fue creada o no.
Para controlar el acceso, el Proxy debe implementar la misma interfaz que el objeto original.
Aplicaciones:
El patrón Proxy es aplicable cuando:

- ★ **Proxy remoto**: Proporciona un representante local para un objeto en un espacio de direcciones diferente, como en sistemas distribuidos.
- ★ **Proxy virtual**: Crea objetos costosos bajo demanda, como en el caso de imágenes grandes.
- ★ **Proxy de protección**: Controla el acceso al objeto original, útil cuando los objetos tienen diferentes niveles de permisos de acceso.
- ★ **Referencia inteligente**: Sustituye un simple puntero y realiza acciones adicionales al acceder al objeto, como contar referencias o cargar un objeto persistente solo cuando es necesario.

Estructura:

- ★ **Proxy** (ImageProxy): Mantiene una referencia al objeto real y controla el acceso a él. Ofrece una interfaz idéntica a la del objeto real (sujeto), por lo que puede ser sustituido en cualquier lugar donde se use el sujeto.
- ★ **Sujeto** (Graphic): Define la interfaz común para el objeto real y el proxy, permitiendo que el proxy se use de manera intercambiable con el objeto real.
- ★ **Objeto real** (Image): Es el objeto que el proxy representa.

Colaboraciones:
El Proxy reenvía las solicitudes al objeto real cuando es necesario. Dependiendo del tipo de proxy, puede llevar a cabo acciones adicionales antes de hacerlo, como crear el objeto o verificar permisos.
Consecuencias:
El patrón Proxy introduce un **nivel de indirecto** en el acceso a un objeto, lo que tiene varios usos:

- ★ Un proxy remoto puede ocultar el hecho de que el objeto reside en otro espacio de direcciones, facilitando la programación distribuida.
- ★ Un proxy virtual puede realizar optimizaciones, como crear el objeto bajo demanda, lo que ahorra recursos.
- ★ Los proxies de protección y las referencias inteligentes permiten realizar tareas adicionales de administración, como gestionar permisos o cargar objetos cuando se acceden.

Optimización adicional: **Copy-on-Write (Copia al escribir)**:

- ★ Este mecanismo permite copiar un objeto solo cuando es modificado. Si el objeto no se modifica, se evita el costo de copiarlo. **El proxy puede retrasar el proceso de copia hasta que el cliente realmente modifique el objeto**, lo que reduce significativamente el costo de manejar objetos pesados en memoria.

Patrones de comportamiento

Los patrones de comportamiento están orientados a cómo los objetos interactúan y se comunican entre ellos para realizar tareas específicas de manera organizada y flexible.
Los patrones de comportamiento están **relacionados con los algoritmos y la asignación de responsabilidades entre objetos.** Estos patrones no solo describen patrones de objetos o clases, sino también los **patrones de comunicación entre ellos**. Caracterizan un flujo de control complejo que es difícil de seguir en tiempo de ejecución. Desvían tu enfoque del flujo de control para que te concentres únicamente en la forma en que los objetos están interconectados.
Los patrones de comportamiento de clase usan la herencia para distribuir el comportamiento entre clases.
Los patrones de comportamiento de objetos usan la composición de objetos en lugar de la herencia. Algunos describen cómo un grupo de objetos pares coopera para realizar una tarea que ningún objeto individual puede llevar a cabo por sí solo. Un tema importante aquí es cómo los objetos pares se conocen entre sí. Los pares podrían mantener referencias explícitas entre ellos, pero eso aumentaría su acoplamiento. En el extremo, cada objeto sabría sobre todos los demás.

Interpreter

El patrón **Interpreter** se utiliza para **definir una gramática de un lenguaje** y **proporcionar un intérprete** que utiliza esta representación para interpretar oraciones en el lenguaje. Este patrón es útil cuando se trata de problemas recurrentes que pueden expresarse como oraciones en un lenguaje simple, permitiendo construir un intérprete para resolver estos problemas mediante la interpretación de las oraciones.
Ejemplo:
Imagina que estás trabajando con expresiones regulares. En lugar de escribir un algoritmo específico para cada patrón de cadena que deseas verificar, **puedes construir un intérprete que entienda expresiones regulares y resuelva el problema interpretando estas expresiones**. Cada regla de la gramática se representa como una clase. Luego, se crea un árbol de sintaxis abstracta para cada expresión, y el método Interpret se implementa en cada clase. Este método toma como argumento el contexto que contiene la cadena de entrada y la información sobre cuánto de la cadena se ha interpretado.
Este patrón es comúnmente utilizado en compiladores y herramientas que interpretan lenguajes específicos, como los interpretadores de expresiones regulares.
El patrón Interpreter es considerado un **patrón de clase** porque su principal característica es **definir una jerarquía de clases que representan las reglas gramaticales de un lenguaje determinado.** Estas clases forman una estructura que interpreta expresiones, y la relación entre ellas se establece de manera estática, es decir, en tiempo de compilación, a través de **herencia y composición.**

Template Method

El patrón **Template Method** es un patrón de comportamiento que **define el esqueleto de un algoritmo en una operación, delegando algunos pasos a las subclases**. Esto permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura general.

Beneficios:

- ★ **Reutilización de código**: El patrón Template Method es **fundamental para la reutilización de código**, especialmente en bibliotecas de clases. Las partes del algoritmo que no varían se implementan en la clase abstracta, mientras que los pasos variables se implementan en las subclases.
- ★ **Control sobre la extensión**: Permite a las subclases intervenir en puntos específicos sin alterar el algoritmo general.

Este patrón es útil para definir partes invariantes de un proceso, mientras que las subclases tienen la flexibilidad de modificar las partes que varían, lo cual es esencial en el diseño orientado a objetos.

Ejemplo: Juego
Un ejemplo común es en la creación de juegos donde tienes diferentes tipos de juegos, pero la secuencia general de los pasos para jugar es la misma.
Supongamos que tienes una clase abstracta llamada Juego. En esta clase, defines el flujo general de un juego usando el método jugar(), que es el template method. Este método incluye los pasos clave que cualquier juego debe seguir: iniciar, jugar la partida, y finalizar el juego.
Luego, tienes diferentes juegos que implementan los detalles específicos: Fútbol, ajedrez, etc.
Explicación:
El método **jugar() es el template method** que define los pasos del algoritmo.
Las subclases (Fútbol y Ajedrez) implementan los detalles específicos de cada paso del algoritmo, como cómo iniciar el juego, cómo jugar y cómo finalizarlo.
Aunque los juegos son diferentes, la estructura del flujo general es la misma.
Este ejemplo ilustra cómo el patrón Template Method permite definir un esqueleto de proceso común, dejando a las subclases los detalles específicos que pueden variar de una implementación a otra.

Chain of Responsibility

El patrón **Chain of Responsibility** (Cadena de Responsabilidad) está diseñado para **evitar el acoplamiento directo entre el emisor de una solicitud y su receptor**. Este patrón permite que múltiples objetos tengan la oportunidad de manejar una solicitud, la cual se pasa a lo largo de una cadena de objetos hasta que uno de ellos la maneja.
Componentes del Patrón:
- ★ **Handler** (Manejador): Define la interfaz para manejar solicitudes. Además, puede implementar un enlace al sucesor.
- ★ **ConcreteHandler**: Implementa el método para manejar la solicitud. Si puede manejarla, lo hace; de lo contrario, la pasa al sucesor.
- ★ **Client** (Cliente): Inicia la solicitud a uno de los manejadores de la cadena.

Beneficios:
- ★ **Menor acoplamiento**: El patrón desacopla al emisor del receptor, permitiendo que ninguno conozca al otro explícitamente.
- ★ **Flexibilidad**: La cadena puede modificarse dinámicamente en tiempo de ejecución, lo que permite una distribución flexible de las responsabilidades.
- ★ **Posibilidad de no manejo**: Existe la posibilidad de que la solicitud no sea manejada si ningún objeto de la cadena está configurado para procesarla.

Ejemplo:
En una interfaz gráfica, un sistema de ayuda sensible al contexto puede utilizar este patrón. Un botón en un cuadro de diálogo envía una solicitud de ayuda que pasa por diferentes niveles de contexto (botón, cuadro de diálogo, aplicación) hasta que se encuentra el manejador adecuado que puede procesar la solicitud.

Command

El patrón **Command** encapsula una solicitud como un objeto, permitiendo parametrizar clientes con diferentes solicitudes, ponerlas en cola, registrarlas o soportar operaciones deshacibles (undo). Este patrón se utiliza cuando **es necesario emitir solicitudes a objetos sin conocer los detalles de la operación o quién es el receptor**.

Componentes del patrón Command:
- ★ **Command**: Declara la interfaz para ejecutar una operación.
- ★ **ConcreteCommand**: Implementa la interfaz Command, asociando un objeto receptor con una acción.
- ★ **Client**: Crea objetos ConcreteCommand y define su receptor.
- ★ **Invoker**: Llama al método Execute() del comando para llevar a cabo una solicitud.
- ★ **Receiver**: Sabe cómo realizar las operaciones solicitadas.

Ejemplo del patrón: Un buen ejemplo es el uso de comandos en interfaces de usuario como botones y menús. Cuando un usuario interactúa con un menú, un objeto MenuItem llama a un comando (por ejemplo, PasteCommand) que luego ejecuta la acción específica como pegar texto en un documento.
Consecuencias:

- ★ Desacopla el objeto que invoca la operación de aquel que sabe cómo realizarla.
- ★ Los comandos pueden tratarse como objetos de primera clase, lo que permite manipularlos y extenderlos.
- ★ Es sencillo añadir nuevos comandos sin cambiar las clases existentes.

Uso común:
Este patrón se emplea en sistemas donde es necesario soportar operaciones deshacibles o macros, como editores de texto o sistemas de diseño

Iterator

El patrón Iterator **proporciona un mecanismo para acceder a los elementos de un objeto agregado de manera secuencial sin exponer su representación interna**. Es también conocido como Cursor.
En objetos agregados como listas o colecciones, puede ser necesario acceder a los elementos de distintas formas. El patrón *Iterator* **permite separar la responsabilidad de acceder y recorrer los elementos del objeto agregado, delegando esta tarea a un objeto iterador**. Esto permite al cliente recorrer la estructura sin preocuparse por los detalles internos del objeto que contiene los datos.
Ejemplo: una lista (List), no es necesario implementar todos los métodos de recorrido dentro de la propia clase List. En lugar de eso, se puede crear un iterador (ListIterator) que se encargue del recorrido. Este iterador sabe cuál es el elemento actual y permite avanzar al siguiente o verificar si el recorrido ha finalizado.
Componentes:

- ★ Iterator: Define la interfaz para acceder y recorrer los elementos.
- ★ ConcreteIterator: Implementa la interfaz del iterador y mantiene un seguimiento de la posición actual dentro del agregado.
- ★ Aggregate: Define la interfaz para crear un iterador.
- ★ ConcreteAggregate: Implementa la interfaz de creación del iterador para devolver una instancia del ConcreteIterator.

Aplicabilidad:
Usa el patrón Iterator cuando:

- ★ Se necesita acceder a los elementos de un objeto agregado sin exponer su representación interna.
- ★ Se necesita soportar múltiples recorridos sobre un mismo objeto agregado.
- ★ Quieres proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas, apoyando la iteración polimórfica.

Consecuencias:

- ★ Variaciones en el recorrido: El patrón permite cambiar la forma en que se recorre un objeto agregado sin modificar su implementación.
- ★ Simplificación de la interfaz del agregado: Al delegar la lógica del recorrido en el iterador, **se simplifica la interfaz del objeto agregado**.
- ★ Múltiples recorridos: Un iterador mantiene su propio estado de recorrido, lo que permite realizar múltiples recorridos simultáneos sobre el mismo objeto.

Iteradores internos y externos:
Iterador externo: El cliente controla explícitamente el avance del iterador y solicita los elementos de uno en uno.
Iterador interno: El iterador controla el recorrido y el cliente le proporciona una operación a ejecutar sobre cada elemento.
El patrón Iterator permite desacoplar la lógica de recorrido de las estructuras de datos, haciendo que sea más fácil extender los recorridos o agregar nuevos tipos de agregados sin modificar el código cliente.

Mediator

El patrón **Mediator define un objeto que encapsula cómo interactúa un conjunto de objetos**. Promueve un acoplamiento débil entre los objetos al **evitar que se refieran directamente entre sí**, permitiendo variar su interacción independientemente.
En un diseño orientado a objetos, la distribución del comportamiento entre múltiples objetos puede generar una estructura con muchas conexiones entre ellos. Esto puede complicar el sistema, haciéndolo parecer un monolito donde cada objeto depende de muchos otros, lo que dificulta su mantenimiento y extensión.
Por ejemplo, en una interfaz gráfica de usuario con cuadros de diálogo, hay widgets (como botones, menús, y campos de entrada) que dependen unos de otros. Si cada widget necesita referirse a otros widgets, la interconexión entre ellos puede complicar el diseño. Aquí es donde entra en juego el patrón Mediator, que **introduce un objeto mediador que centraliza y coordina la interacción** entre los widgets.
Participantes:

- ★ Mediator (DialogDirector): Define la interfaz para la comunicación con los objetos colegas (colleagues).
- ★ ConcreteMediator (FontDialogDirector): Implementa el comportamiento cooperativo entre los objetos colegas, coordinando sus interacciones.
- ★ Colleague (ListBox, EntryField): Cada objeto conoce a su mediador y se comunica con él en lugar de hacerlo directamente con otros colegas.

Colaboraciones:
Los objetos relacionados envían y reciben solicitudes a través del mediador, que es **responsable de enrutar las solicitudes y coordinar el comportamiento cooperativo entre los objetos**.
Consecuencias:

1. **Limita la creación de subclases**: El comportamiento cooperativo se centraliza en el mediador, lo que evita la necesidad de modificar los objetos colegas.
2. **Desacopla** a los colegas: Los objetos colegas no necesitan conocer los detalles de otros, lo que promueve un acoplamiento flexible.
3. Simplifica los protocolos de los objetos: Reduce la interacción muchos-a-muchos a una interacción uno-a-muchos entre el mediador y sus colegas, lo que facilita el mantenimiento y la extensión.
4. **Centraliza el control**: La complejidad de la interacción se transfiere al mediador, lo que puede hacer que este se convierta en una entidad compleja y difícil de mantener.

El patrón Mediator es útil cuando se necesita centralizar y simplificar la interacción entre objetos que de otro modo estarían altamente acoplados entre sí.

Memento

El patrón **Memento** permite **capturar y externalizar el estado interno de un objeto, sin violar la encapsulación, para poder restaurarlo más adelante**. Es útil cuando necesitas implementar mecanismos de deshacer (undo) o puntos de control en un sistema.
En algunas aplicaciones, es necesario registrar el estado interno de un objeto para permitir que el sistema restaure ese estado más adelante. Esto es típico en editores gráficos o de texto, donde se implementan funciones de "deshacer". Sin embargo, exponer el estado interno de un objeto directamente violaría la encapsulación, lo que podría afectar la confiabilidad y extensibilidad de la aplicación.
Por ejemplo, en un editor gráfico que mantiene conectividad entre objetos (como líneas que conectan rectángulos), es difícil revertir los movimientos de forma precisa solo almacenando la distancia que se movieron. El Memento permite al sistema almacenar el estado interno del objeto sin exponer sus detalles.
Participantes:

- ★ **Memento** (SolverState): **Almacena el estado interno del objeto originador**. Solo el objeto originador puede acceder al contenido del Memento.
- ★ **Originador** (ConstraintSolver): **Crea el Memento** y lo usa para restaurar su estado.
- ★ **Cuidador** (Caretaker): Se encarga de guardar el Memento y nunca lo modifica directamente.

Colaboraciones:
El Cuidador solicita un Memento del Originador y lo guarda.
Si es necesario restaurar el estado anterior, el Cuidador devuelve el Memento al Originador, quien usa la información almacenada para volver a su estado anterior.

Consecuencias:

1. Preservación de la encapsulación: El patrón Memento **evita exponer detalles internos del objeto mientras permite guardar su estado**.
2. Simplificación del Originador: El **manejo del almacenamiento** del estado se **delega al Cuidador**, lo que simplifica la lógica del Originador.
3. Costos: El uso de Mementos puede ser costoso si el estado a almacenar es grande o si se crean demasiados Mementos.

Este patrón es ideal cuando se requiere implementar funcionalidades de deshacer sin violar la encapsulación. Delega la creación de las instantáneas del estado al propietario real de ese estado. Por lo tanto, la clase original puede hacer las instantáneas, ya que tiene acceso completo a su propio estado.
El término "instantáneas" hace referencia a capturar un estado específico de un objeto en un momento dado, de manera que más adelante puedas restaurar ese estado si es necesario.
Es como tomar una "foto" o un "registro" del estado interno de un objeto en un punto particular del tiempo. Esa "instantánea" se guarda en un objeto Memento, y luego se puede utilizar para volver a ese estado sin exponer los detalles internos del objeto.

Observer

El patrón Observer define una **dependencia uno-a-muchos** entre objetos de manera que **cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente**.
En un sistema donde varios objetos cooperan, es común que se necesite mantener la consistencia entre ellos. Sin embargo, no se desea lograr esta consistencia mediante un acoplamiento fuerte, ya que esto reduciría la reutilización de los objetos.
Por ejemplo, en un entorno de interfaz gráfica de usuario (GUI), las clases que representan los datos de la aplicación y las clases que muestran esos datos deben mantenerse consistentes. Si los datos cambian, la vista debe reflejar ese cambio de inmediato. Sin embargo, se quiere evitar un acoplamiento fuerte entre ambas clases. Aquí es donde entra en juego el patrón Observer: permite que múltiples vistas se sincronicen con un único modelo de datos sin que estas dependan fuertemente entre sí.
Este tipo de **interacción** también se conoce como **publish-subscribe**: el sujeto es el editor de notificaciones, y los observadores son los suscriptores que reciben las notificaciones cuando ocurre un cambio en el sujeto.
Participantes:

- ★ Subject (Sujeto):
  - Mantiene una **lista de observadores**.
  - Proporciona métodos para agregar, eliminar y notificar a los observadores.
- ★ Observer (Observador):
  - Define una **interfaz para recibir actualizaciones** desde el sujeto.
- ★ ConcreteSubject (Sujeto Concreto):
  - Almacena el estado de interés para los observadores y notifica los cambios a estos.
- ★ ConcreteObserver (Observador Concreto):
  - Mantiene una **referencia** al sujeto concreto.
  - Implementa la actualización, manteniéndose **sincronizado** con el **estado** del **sujeto**.

Colaboraciones:
El sujeto notifica a sus observadores cada vez que cambia su estado, generalmente llamando al método update() de cada observador.
Los observadores pueden consultar el estado del sujeto después de ser notificados para realizar las acciones correspondientes.
Consecuencias:

1. Acoplamiento flexible: Los observadores y el sujeto están acoplados débilmente, lo que permite que ambos evolucionen de manera independiente.
2. **Soporte para comunicación "uno a muchos"**: Un sujeto puede tener varios observadores, permitiendo que un cambio en el sujeto afecte a múltiples objetos.
3. Actualizaciones automáticas: Los observadores se actualizan automáticamente cuando cambia el estado del sujeto.

4. Posibles problemas de rendimiento: Si hay demasiados observadores, las notificaciones pueden volverse costosas en términos de rendimiento.

Este patrón es ideal para situaciones donde varios objetos dependen del estado de uno solo, como en interfaces gráficas, sistemas de eventos o modelos de datos, asegurando que todos los objetos se mantengan sincronizados sin necesidad de depender directamente unos de otros

State

El patrón **State** permite que un **objeto altere su comportamiento cuando su estado interno cambia**. El objeto parecerá cambiar de clase, ya que se comportará de manera diferente según su estado.

Participantes:

- ★ **Contexto** (TCPConnection):
  - Define la interfaz de interés para los clientes.
  - Mantiene una instancia de una subclase de ConcreteState que define el estado actual.
- ★ **State** (TCPState):
  - Define una interfaz para encapsular el comportamiento asociado a un estado particular del contexto.
- ★ **Subclases** de ConcreteState (TCPEstablished, TCPListen, TCPClosed):
  - Cada subclase implementa el comportamiento asociado a un estado del contexto.

Colaboraciones:

- ★ El Contexto delega las solicitudes relacionadas con el estado al objeto ConcreteState actual.
- ★ El Contexto puede pasarse a sí mismo como argumento al objeto State, lo que permite que el objeto de estado acceda al contexto si es necesario.
- ★ Los clientes configuran el contexto con objetos de estado, pero no necesitan interactuar directamente con ellos.
- ★ Tanto el Contexto como las subclases de ConcreteState pueden decidir qué estado sigue a otro y bajo qué circunstancias.

Consecuencias:

1. **Localización del comportamiento específico del estado**: El patrón State organiza todo el comportamiento relacionado con un estado particular en una clase. Esto **facilita agregar nuevos estados y transiciones** simplemente definiendo nuevas subclases. El patrón evita tener grandes sentencias condicionales dispersas en la implementación del contexto, lo que simplifica el mantenimiento.
2. Transiciones explícitas de estado: Definir el estado actual mediante objetos de estado hace que las transiciones de estado sean más explícitas. En lugar de manejar cambios de estado con variables internas, **el patrón cambia de estado al reemplazar el objeto de estado en el contexto, lo que evita inconsistencias.**
3. Posibilidad de compartir objetos de estado: Si los objetos de estado no tienen variables de instancia, pueden compartirse entre varios contextos. En este caso, los objetos de estado se comportan como flyweights, ya que solo representan el comportamiento, no un estado intrínseco.

El patrón State permite encapsular estas transiciones y comportamientos, haciéndolos más fáciles de gestionar y extender.

State vs. Strategy

State puede ser considerado una extensión de Strategy ya que los dos patrones están basados en la composición, los dos cambian el comportamiento del contexto delegando el trabajo a objetos que ayuden.

State	Strategy
Pueden ser dependientes ya que podes ir de un estado a otro, o necesitar de otro estado. los estados se conocen. Se trata de hacer cosas diferentes basadas en el estado en el que se encuentran, entonces varía.	Son independientes y no conocen al otro objeto. Se trata de tener **diferentes implementaciones que llegan a lo mismo**.

Strategy

El patrón **Strategy** define una **familia de algoritmos**, **los encapsula por separado y los hace intercambiables**. Esto permite variar el algoritmo de manera independiente del cliente que lo usa.
Supongamos un sistema que necesita dividir un texto en líneas. Existen varias formas de hacerlo, pero no es práctico tener todos los algoritmos incrustados en una clase, ya que:

1. Complejidad del cliente: Los clientes que usan la lógica de división se volverían más complicados si incluyeran el código de todos los algoritmos, lo que dificultaría el mantenimiento.
2. Algoritmos apropiados en diferentes momentos: No siempre se necesitarán todos los algoritmos, y tenerlos todos disponibles al mismo tiempo sería ineficiente.
3. Dificultad para agregar o modificar algoritmos: Si los algoritmos de división están integrados en el cliente, es difícil agregar nuevos o modificar los existentes sin alterar el código base.

El patrón Strategy resuelve estos problemas encapsulando los algoritmos en clases separadas, llamadas estrategias.
Aplicabilidad:
El patrón Strategy se usa cuando:

- ★ Muchas clases relacionadas difieren solo en su comportamiento. **Las estrategias permiten configurar una clase con uno de varios comportamientos**.
- ★ Se necesitan diferentes variantes de un algoritmo. Por ejemplo, se puede definir diferentes algoritmos según las compensaciones entre espacio y tiempo.
- ★ Un algoritmo usa datos que los clientes no deben conocer. El patrón Strategy **oculta estructuras de datos complejas específicas del algoritmo**.
- ★ Una clase tiene **múltiples comportamientos que se gestionan mediante sentencias condicionales**. El patrón Strategy permite mover cada rama condicional a su propia clase de estrategia.

Estructura:

1. Strategy: Declara una **interfaz común para todos los algoritmos**. El contexto la usa para invocar el algoritmo.
2. ConcreteStrategy: Implementa el algoritmo utilizando la interfaz de Strategy.
3. Context: Está configurado con un objeto ConcreteStrategy y mantiene una referencia a este.

Colaboraciones:

- ★ Context y Strategy interactúan para implementar el algoritmo seleccionado. El contexto puede pasar datos necesarios al algoritmo o incluso pasarse a sí mismo como argumento para que la estrategia pueda acceder a él.
- ★ Los clientes configuran el contexto con la estrategia adecuada y luego interactúan sólo con el contexto, sin necesidad de interactuar directamente con las estrategias.

Consecuencias:

- ★ Familias de algoritmos relacionados: Las jerarquías de clases Strategy definen una familia de algoritmos o comportamientos que pueden reutilizarse.
- ★ Alternativa a la subclase: En lugar de usar herencia para variar comportamientos, encapsular los algoritmos en clases Strategy permite variarlos de manera independiente del contexto.
- ★ Eliminación de sentencias condicionales: Las estrategias eliminan la necesidad de usar if o switch para seleccionar un comportamiento, lo que simplifica el código.
- ★ Variedad de implementaciones: Se pueden ofrecer estrategias con diferentes compromisos entre tiempo y espacio.
- ★ Los clientes deben conocer las estrategias: Un inconveniente es que los **clientes** deben **entender** las diferencias entre las **estrategias** para seleccionar la correcta.
- ★ Aumento de la comunicación entre Strategy y Context: Puede haber sobrecarga si se pasa más información de la necesaria entre el contexto y la estrategia, especialmente si algunas estrategias no usan todos los datos proporcionados.
- ★ Aumento del número de objetos: El uso de estrategias aumenta el número de objetos en el sistema, aunque esto puede mitigarse si las estrategias no mantienen estado y se comparten entre contextos.

Visitor

El patrón **Visitor** permite representar una operación a ser realizada sobre los elementos de una estructura de objetos. **Este patrón permite definir nuevas operaciones sin modificar las clases de los elementos sobre los que opera**.
El patrón Visitor resuelve este problema al **encapsular estas operaciones en un objeto separado**, llamado **visitante** (visitor). Cuando un elemento del árbol de sintaxis acepta al visitante, le envía una solicitud que contiene su clase específica. Así, el visitante puede realizar la operación adecuada para ese tipo de nodo sin necesidad de modificar las clases de los nodos.
Participantes:

- ★ Visitor (NodeVisitor):
  - Declara una operación Visit para cada clase de elementos concretos en la estructura.
- ★ ConcreteVisitor (TypeCheckingVisitor):
  - Implementa las operaciones definidas en Visitor. Cada operación ejecuta un fragmento del algoritmo específico para cada tipo de elemento.
- ★ Element (Node):
  - Declara una operación Accept que toma un visitante como argumento.
- ★ ConcreteElement (AssignmentNode, VariableRefNode):
  - Implementa la operación Accept, que toma un visitante y permite que este llame a su método específico.
- ★ ObjectStructure (Program):
  - Puede enumerar sus elementos y proporcionar una interfaz de alto nivel para que el visitante los recorra.

Colaboraciones:

- ★ El cliente crea un objeto Visitor concreto y luego recorre la estructura de objetos, permitiendo que el visitante realice las operaciones en cada elemento.
- ★ Cuando un elemento es visitado, llama a la operación correspondiente del Visitor y le proporciona su propio estado como argumento para que este pueda realizar la operación deseada.

Consecuencias:

1. Facilidad para agregar nuevas operaciones: El patrón **Visitor permite agregar nuevas operaciones que dependen de los elementos de una estructura de objetos sin modificar las clases de esos elementos**. Esto se logra simplemente añadiendo un nuevo visitante.
2. Agrupación de operaciones relacionadas: El patrón agrupa operaciones relacionadas en una clase visitante en lugar de distribuirlas por las clases de la estructura de objetos, lo que simplifica el mantenimiento y evita que las clases se llenen de operaciones innecesarias.
3. Dificultad para agregar nuevas clases de elementos: Aunque es fácil agregar nuevas operaciones, añadir nuevas clases de elementos (por ejemplo, nuevos tipos de nodos en un árbol de sintaxis) puede ser difícil, ya que requeriría modificar la clase Visitor y todas sus subclases para soportar el nuevo elemento.
4. Visitar diferentes jerarquías de clases: A diferencia del patrón Iterator, que solo puede recorrer elementos de una misma clase base, Visitor puede trabajar con estructuras de objetos que no compartan una clase padre común. Esto permite que un visitante visite diferentes tipos de objetos.
5. Acumulación de estado: Los visitantes pueden acumular estado a medida que visitan cada elemento de la estructura, lo que simplifica la implementación de algunas operaciones.
6. Posible ruptura de encapsulamiento: El patrón Visitor puede requerir que las clases expongan parte de su estado interno para que el visitante pueda operar sobre él, lo que puede comprometer el encapsulamiento.

El patrón Visitor es útil **cuando tienes una estructura de objetos y necesitas realizar muchas operaciones diferentes sobre esos objetos**, sin querer modificar sus clases cada vez que surge una nueva operación. Es especialmente útil en sistemas complejos como compiladores, donde los nodos de un árbol de sintaxis necesitan soportar múltiples operaciones. Sin embargo, puede ser más difícil de mantener si frecuentemente se agregan nuevos tipos de elementos a la estructura.