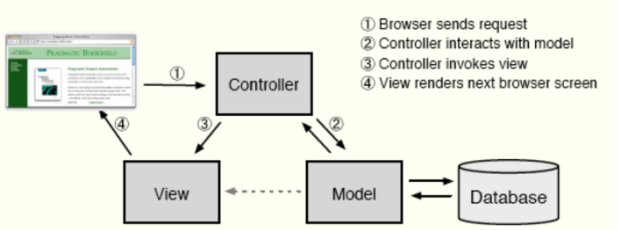
Ignacio Méndez Pérez
2023

Resumen Ingeniería de Software



- HTTP es un protocolo *stateless* (sin estado). Es decir, cada solicitud se considera como una entidad independiente, sin ninguna relación con las solicitudes anteriores.

- Arquitectura Model – View – Controller (**MVC**).

    - Modelo: Representa los datos y la lógica de negocio de la aplicación. Es responsable de acceder y manipular los datos, y provee métodos para realizar operaciones y responder consultas relacionadas con la lógica de dominio.

    - Vista: Es la interfaz de usuario que muestra la información al usuario y recibe sus interacciones. La vista se encarga de presentar los datos del modelo de una manera visualmente atractiva y comprensible.

    - Controlador: Actúa como intermediario entre el modelo y la vista. Recibe las interacciones del usuario a través de la vista y coordina las acciones necesarias para responder a esas interacciones. El controlador actualiza el modelo según las acciones del usuario y decide qué vista mostrar en función de los cambios en el modelo.

- **Asociaciones** **N-N**:

    - Cuando tenemos una relación de muchos a muchos en una base de datos, no podemos simplemente poner una llave foránea en alguna de las tablas, dado que una fila debería poder tener "múltiples" llaves foráneas y esto no es posible con una simple tabla. Es por esto por lo que se debe crear una tabla intermedia que contiene un listado con las relaciones.

- **Cookies**:

    - Sabemos que la web es *stateless*, sin embargo, de vez en cuando es útil pasar información entre requests sucesivos de una misma sesión sin pasar por la BD.

    - Las cookies son pequeños archivos de texto que se almacenan en el navegador web de un usuario cuando visita un sitio web. Estos archivos se utilizan para almacenar información específica sobre la sesión y la interacción del usuario con el sitio. Cuando el usuario vuelve a visitar el mismo sitio, el navegador envía las cookies asociadas con ese sitio de vuelta al servidor, lo que permite al servidor recordar información y adaptar la experiencia del usuario.

- Las cookies son particularmente útiles para retener información que cumpla las siguientes características:

  1. Liviana: Idealmente de no más de un par de KB.
  2. Temporal: Solo es necesario retener la información brevemente.
  3. No crítica: Se acepta que las cookies pueden borrarse en cualquier momento.

- **Extremme Programming (XP)**:

  - Cliente siempre como parte del equipo de desarrollo.
  - Casos de uso muy simples.
  - Cliente especifica casos de prueba.
  - Software lo más simple posible.
  - Programación en pares.

- **Scrum**:

  - Modelo de proceso iterativo incremental.
  - Software se elabora en iteraciones llamadas “sprints”.
  - Qué va a ir en el sprint depende principalmente de la negociación con el representante del cliente (*product owner*).
  - Al final de cada iteración hay dos reuniones antes de comenzar la siguiente:
    - Review: Qué es lo que se logró y que es lo que falta.
    - Retroespectiva: Cómo funciona el proceso

  - 3 roles:

    - Product owner:
      - Responsable de representar a los interesados y garantizar que se construya un producto o sistema de alta calidad que satisfaga las necesidades y requisitos del cliente.
      - Define y prioriza el backlog del producto.

    - Scrum Master:
      - El Scrum Master es un facilitador y líder de servicio al equipo Scrum. Su objetivo principal es garantizar que el equipo pueda trabajar de manera eficiente y efectiva, siguiendo los principios y prácticas de Scrum.
      - Resuelve problemas.
      - NO es el jefe del proyecto.

    - Team:
      - Diverso, de 5 a 8 personas.

- Los sprints duran entre 2 y 3 semanas. Son todos de igual duración (*timeboxed*).

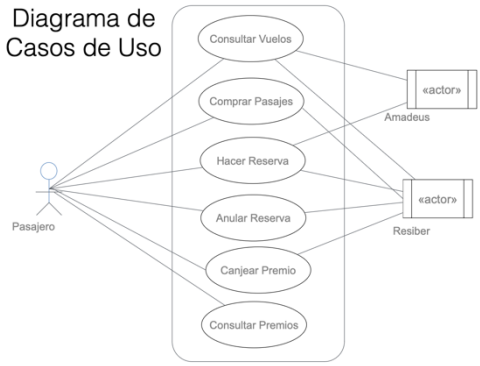
- *Dailys*: Miembros indican que hice desde ayer, qué pretendo hacer hoy y posibles obstáculos visibles. NO es momento para resolver problemas.

- **Kanban**:

  - Visualizar flujo de trabajo.
  - Dividir el trabajo en trozos pequeños y escribirlos en tarjetas.
  - Usar columnas para saber en qué parte del *workflow* está cada tarea.

- **Casos de uso**:

  - Usuario tiene metas, queremos que el software les ayude a lograrlas.
  - Una vez que se hayan capturado las necesidades de todos los actores hemos completado los requisitos funcionales.
  - Un actor representa a un grupo de usuarios similares.
  - El caso de uso es un relato de como un actor usa el sistema.
  - Existe el *happy path* (se usa todo como corresponde) y el *unhappy path* (flujos alternativos).



- Problemas de sobreestimar:

  - Ley de Parkinson: El trabajo se expandirá hasta usar todo el tiempo disponible.
  - Ley de Goldratt: Si hay demasiado tiempo se malgasta el comienzo del proyecto.

- Problemas de subestimar:

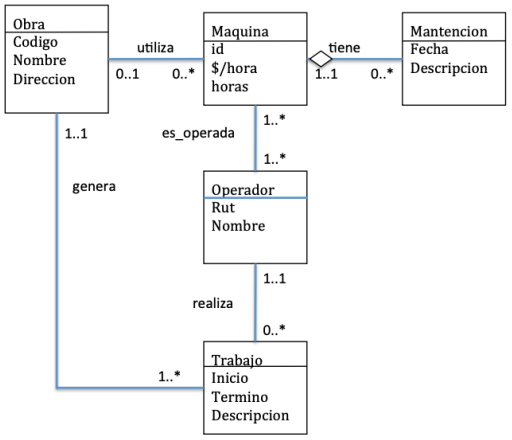
  - Errores de planeación, errores de coordinación.
  - Reducción de tiempo en análisis y diseño puede generar mucho más retraso.
  - Dinámica del proyecto atrasado, problemas con el cliente, arreglos parche, etc.

- Modelo de cascada:
  - Se basa en un proceso secuencial en el que el desarrollo del software se divide en etapas distintas y bien definidas, y cada etapa se lleva a cabo de manera secuencial, siguiendo un flujo de arriba hacia abajo, similar a una cascada.

Ignacio Méndez Pérez
2023

- Modelo de dominio

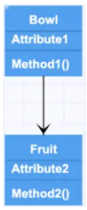
  - Representación estructurada de los conceptos, entidades y reglas que están involucrados en un determinado dominio o área de conocimiento. En el contexto del desarrollo de software, el modelo de dominio se utiliza para comprender y definir los conceptos clave relacionados con el problema que se va a resolver con el software.



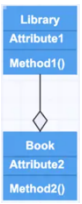
- **Flechas de diagrama de clases**:

  - **Asociación** significa cualquier tipo de relación o conexión entre clases. Por ejemplo, mostramos un vínculo directo entre un autobús urbano y sus pasajeros utilizando una línea de asociación. Mostramos una asociación simple con una línea recta.

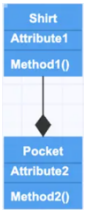
  - La **asociación dirigida** muestra una relación fuerte entre clases. Las clases deben comunicarse. Representamos una asociación directa con una flecha que apunta a nuestra clase de objeto. Por ejemplo, un recipiente puede contener frutas. El recipiente actúa como una clase contenedora para la clase de frutas.



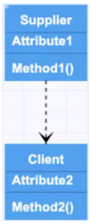
  - Utilizamos flechas de **agregación** cuando queremos transmitir que dos clases están asociadas, pero no tan cercanamente como en una asociación directa. La clase hija puede existir de manera independiente al elemento padre. Por ejemplo, un libro aún existe si alguien lo saca de la biblioteca.



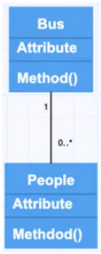
  - Las flechas de **composición** aparecen en los diagramas de clases de UML cuando queremos mostrar una asociación similar a la agregación, pero con una diferencia clave. Las asociaciones de composición muestran relaciones donde el sub-objeto existe solo mientras exista la clase contenedora. Las clases tienen un ciclo de vida común. Por ejemplo, un bolsillo en la parte delantera de una camisa no puede existir si destruimos la camisa.



  - Las flechas de **dependencia** nos muestran dónde dos elementos dependen entre sí, pero en una relación menos fuerte que una asociación básica. Los cambios en la clase padre también afectarán a la clase hija. La dependencia muestra una relación del tipo proveedor-cliente.

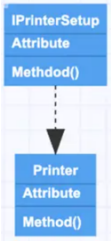


- Las flechas de **multiplicidad** o **cardinalidad** muestran un lugar en nuestro diagrama UML donde una clase puede contener muchos (¡o ninguno!) elementos. Por ejemplo, un autobús urbano puede tener cualquier cantidad de pasajeros en un momento dado. Las personas suben y bajan constantemente a medida que el autobús se desplaza por las calles. Mostramos esto en nuestro diagrama con la notación $0..*$ que significa que nuestra clase puede contener desde cero hasta muchos objetos.

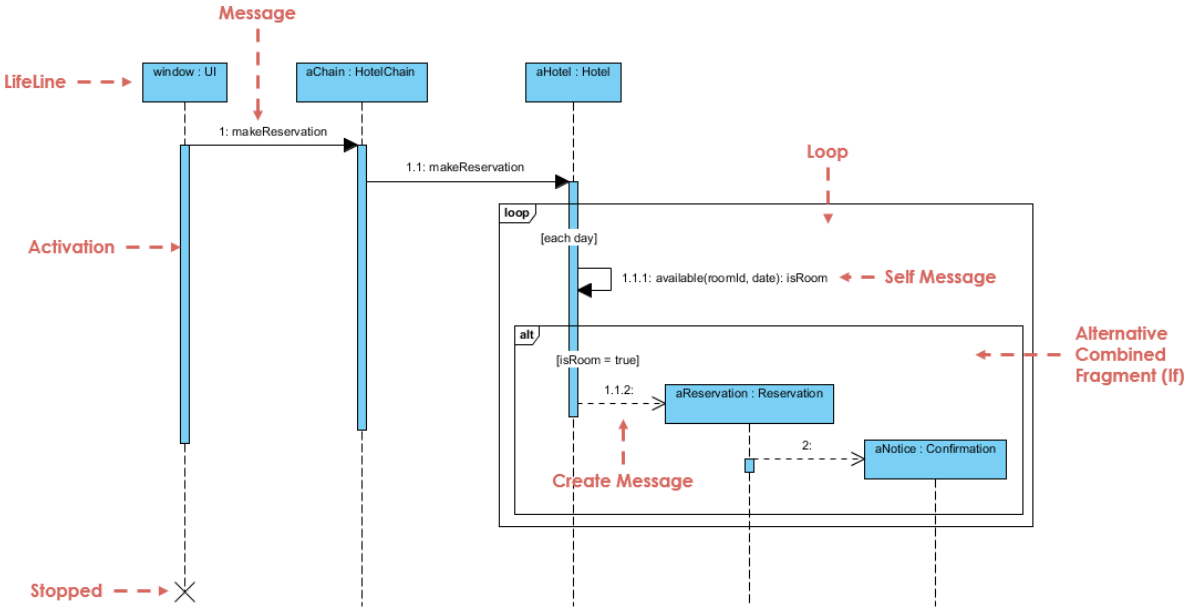


- Utilizamos flechas de **herencia** para mostrar que una clase hija hereda la funcionalidad de la clase padre. Por ejemplo, un aguacate es un tipo de fruta. Fruta es la superclase. Aguacate es la subclase. El aguacate hereda su condición de fruta de su clase padre, fruta.

- Utilizamos flechas de **realización** o **implementación** para indicar un lugar donde una clase implementa la función definida en otra clase. Por ejemplo, la interfaz de configuración de la impresora establece las preferencias de impresión que están siendo implementadas por la impresora. La disposición muestra una asociación de realización.



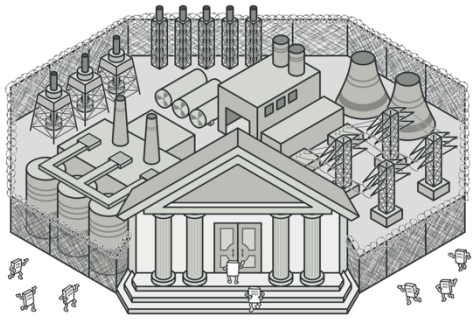
- **Diagrama de secuencia**:



Ignacio Méndez Pérez
2023

- **Patrones de diseño**:

  - **Fachada (*Facade*)**:



El patrón de diseño de fachada, también conocido como "*facade*", es un patrón de diseño estructural que proporciona una interfaz simplificada y unificada para un conjunto de interfaces más complejas de un subsistema. Actúa como una capa de abstracción sobre el subsistema, ocultando su complejidad y proporcionando una interfaz más fácil de usar.

La fachada se encarga de encapsular las interacciones complejas entre los componentes del subsistema y proporciona una interfaz simple y coherente para que los clientes interactúen con él. Esto promueve el modularidad, el acoplamiento bajo y la facilidad de uso.

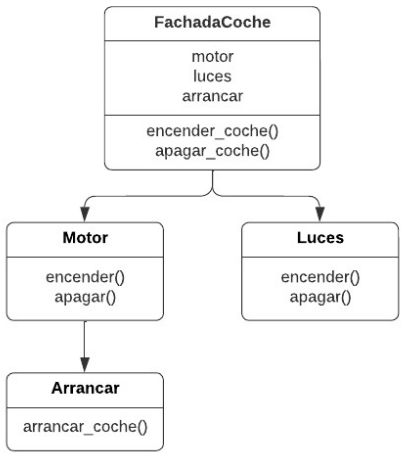
Ejemplo:

1. Sistema de procesamiento de pagos en línea: En un sistema de procesamiento de pagos en línea, puede haber varios componentes involucrados, como el procesamiento de tarjetas de crédito, la validación de direcciones, la verificación de fondos, etc. En lugar de exponer todos estos componentes complejos al cliente, se puede utilizar una fachada que proporcione una interfaz única y sencilla para realizar un pago. La fachada se encarga de coordinar todas las interacciones con los componentes internos y presenta al cliente una interfaz simple para realizar el pago.

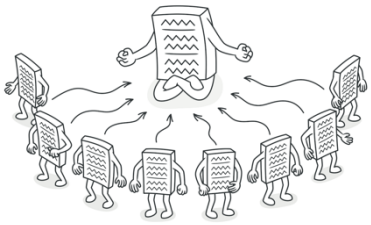
```python
# Subsistema 1
class Motor:
    def encender(self):
        print("Motor encendido")

    def apagar(self):
        print("Motor apagado")

# Subsistema 2
class Luces:
    def encender_luces(self):
        print("Luces encendidas")

    def apagar_luces(self):
        print("Luces apagadas")

# Subsistema 3
class Arrancar:
    def arrancar_coche(self):
        print("Coche arrancado")
```

```python
# Fachada
class FachadaCoche:
    def __init__(self):
        self.motor = Motor()
        self.luces = Luces()
        self.arrancar = Arrancar()

    def encender_coche(self):
        self.motor.encender()
        self.luces.encender_luces()
        self.arrancar.arrancar_coche()
        print("Coche listo para conducir")

    def apagar_coche(self):
        self.motor.apagar()
        self.luces.apagar_luces()
        print("Coche apagado")
```

```python
# Uso del patrón Fachada
coche_fachada = FachadaCoche()
coche_fachada.encender_coche()

print("\nRealizando un viaje ... \n")

coche_fachada.apagar_coche()
```



- ***Singleton*:**



El patrón de diseño *Singleton* es un patrón creacional que asegura que solo haya una instancia de una clase determinada en todo el sistema y proporciona un punto de acceso global a esa instancia.

La idea detrás del patrón *Singleton* es restringir la creación de objetos de una clase a una sola instancia y garantizar que dicha instancia sea accesible desde cualquier parte del sistema. Esto puede ser útil en situaciones en las que solo se permite una instancia de una clase específica, como una clase de configuración global o un objeto de registro

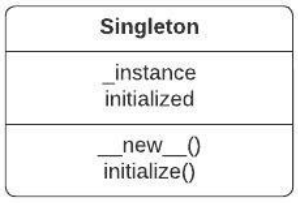
Ejemplo:

1. Gestor de conexiones de base de datos: En una aplicación que requiere conexiones frecuentes a una base de datos, se puede utilizar el patrón Singleton para crear un gestor de conexiones de base de datos. El Singleton asegurará que solo exista una instancia del gestor de conexiones y proporcionará un punto de acceso global para obtener esa instancia. Esto garantiza que todas las partes de la aplicación utilicen la misma conexión de base de datos, evitando problemas de inconsistencia o conflictos de acceso.

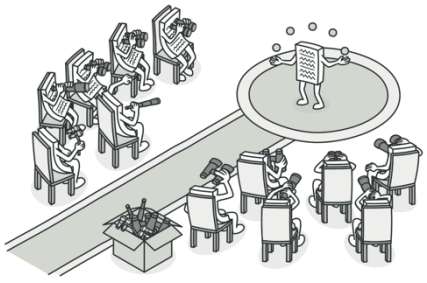
```python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
            cls._instance.initialized = False
        return cls._instance

    def initialize(self):
        if not self.initialized:
            print("Inicializando la instancia Singleton")
            self.initialized = True
```

```python
# Uso del patrón Singleton
s1 = Singleton()
s1.initialize()

s2 = Singleton()
s2.initialize()

print("s1 es igual a s2:", s1 is s2)
```

```
True
```



- **Observador (*Observer*)**:



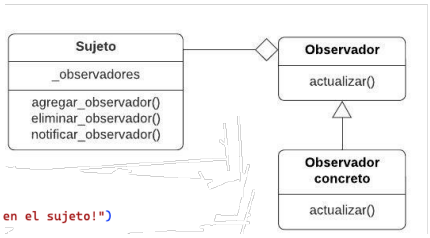
El patrón de diseño Observador, también conocido como patrón de eventos o pub/sub (publicación/suscripción), es un patrón de comportamiento que permite la comunicación entre objetos de manera flexible y desacoplada. En este patrón, hay un objeto llamado “sujeto” o “observable” que mantiene una lista de objetos “observadores”. Cuando ocurre un cambio en el sujeto, notifica automáticamente a todos los observadores registrados para que puedan actualizar su estado o realizar acciones correspondientes.

Ejemplo:

1. Actualización de la interfaz de usuario en tiempo real: Supongamos que se está desarrollando una aplicación en la que los datos cambian constantemente y se desea que la interfaz de usuario refleje estos cambios en tiempo real. En lugar de que la interfaz de usuario consulte continuamente los datos para verificar si ha habido cambios, se puede implementar el patrón Observador. El sujeto sería el objeto que mantiene los datos, y los observadores serían los componentes de la interfaz de usuario interesados en estos datos. Cuando se produzca un cambio en los datos, el sujeto notificará automáticamente a los observadores, quienes actualizarán su interfaz de usuario en consecuencia.

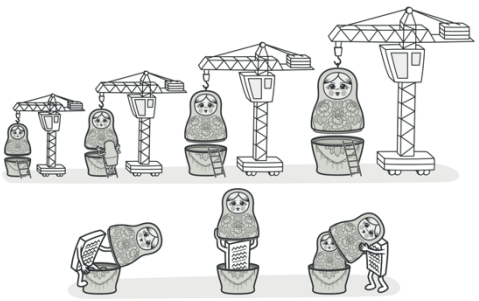
```python
# Sujeto (Subject)
class Sujeto:
    def __init__(self):
        self._observadores = []

    def agregar_observador(self, observador):
        self._observadores.append(observador)

    def eliminar_observador(self, observador):
        self._observadores.remove(observador)

    def notificar_observadores(self, mensaje):
        for observador in self._observadores:
            observador.actualizar(mensaje)

# Observador (Observer)
class Observador:
    def actualizar(self, mensaje):
        raise NotImplementedError

# Observador concreto (Concrete Observer)
class ObservadorConcreto(Observador):
    def actualizar(self, mensaje):
        print(f"Observador recibió el mensaje: {mensaje}")
```

```python
# Uso del patrón Observer
sujeto = Sujeto()
observador1 = ObservadorConcreto()
observador2 = ObservadorConcreto()

sujeto.agregar_observador(observador1)
sujeto.agregar_observador(observador2)

sujeto.notificar_observadores("¡Se ha producido un cambio en el sujeto!")

sujeto.eliminar_observador(observador1)

sujeto.notificar_observadores("Otro cambio en el sujeto")
```



- **Decorador (*Decorator*)**:

El patrón de diseño Decorador, también conocido como *Decorator*, es un patrón estructural que permite agregar funcionalidad adicional a un objeto de manera dinámica. El Decorador actúa como una capa envolvente alrededor del objeto original, agregando o modificando su comportamiento sin alterar su estructura interna.



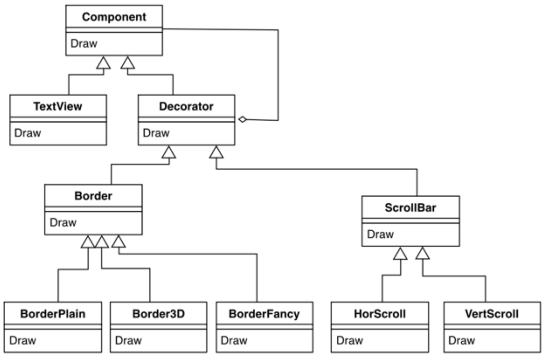
Ejemplo:

1. Decoración de objetos gráficos: Suponga que se dispone de una componente gráfica `TextView` (clase) capaz de desplegar un texto en una ventana gráfica. El problema es que necesitamos ventanas con 3 distintos tipos de bordes (`Plain`, 3D, `Fancy`) y también poder agregar *scrollbars* tanto verticales como horizontales (una, otra, o ambas).

Ignacio Méndez Pérez
2023

```python
class Component:
    def draw(self):
        pass

class TextView(Component):
    def draw(self):
        print("Drawing text view")

class Decorator(Component):
    def __init__(self, component):
        self._component = component

    def draw(self):
        self._component.draw()
```

```python
class BorderDecorator(Decorator):
    def __init__(self, component, border_type):
        super().__init__(component)
        self._border_type = border_type

    def draw(self):
        super().draw()
        print(f"Adding {self._border_type} border")

class ScrollbarDecorator(Decorator):
    def __init__(self, component, scrollbar_type):
        super().__init__(component)
        self._scrollbar_type = scrollbar_type

    def draw(self):
        super().draw()
        print(f"Adding {self._scrollbar_type} scrollbar")
```



Método Draw de los decoradores simplemente invocan primero el draw de ls componente que contienen y luego dibujan lo específico del decorador según sea un borde o una barra de scroll

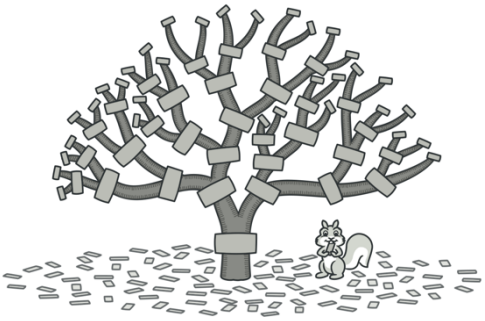
```python
# Uso de ejemplo
plain_text_view = TextView()
decorated_plain_text_view = ScrollbarDecorator(BorderDecorator(plain_text_view, "Plain"), "Vertical")
decorated_plain_text_view.draw()

fancy_text_view = TextView()
decorated_fancy_text_view = ScrollbarDecorator(fancy_text_view, "Vertical")
decorated_fancy_text_view.draw()
```

Component

Component

- **Composite**:

El patrón de diseño Composite es un patrón estructural que permite tratar a los objetos individuales y a las composiciones de objetos de manera uniforme. El patrón Composite permite construir estructuras jerárquicas de objetos en forma de árbol, donde los nodos del árbol pueden ser tanto objetos individuales como composiciones de objetos.



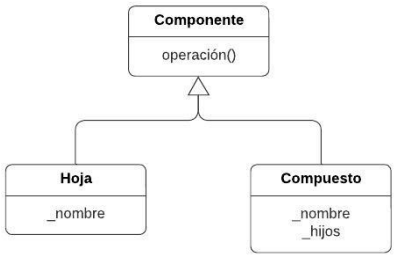
Ejemplo:

1. <u>Representación de una estructura de árbol</u>: Supongamos que se está desarrollando un editor de diagramas que permite crear diagramas compuestos por formas básicas como círculos, rectángulos y líneas. El patrón Composite se puede utilizar para modelar las formas individuales como hojas del árbol y las composiciones de formas (por ejemplo, grupos de formas) como nodos internos del árbol. De esta manera, se puede tratar tanto las formas individuales como los grupos de formas de manera uniforme al realizar operaciones como dibujar, redimensionar o mover.

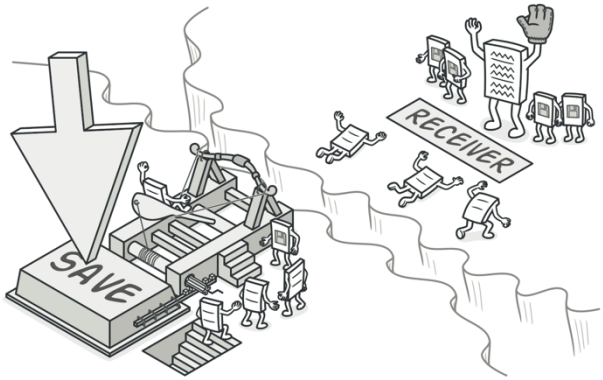
```python
# Componente (Component)
class Componente(ABC):
    @abstractmethod
    def operacion(self):
        pass

# Hoja (Leaf)
class Hoja(Componente):
    def __init__(self, nombre):
        self._nombre = nombre

    def operacion(self):
        return f"Hoja: {self._nombre}"
```

```python
# Compuesto (Composite)
class Compuesto(Componente):
    def __init__(self, nombre):
        self._nombre = nombre
        self._hijos = []

    def agregar(self, componente):
        self._hijos.append(componente)

    def operacion(self):
        resultado = f"Compuesto: {self._nombre}\n"
        for hijo in self._hijos:
            resultado += hijo.operacion() + "\n"
        return resultado
```

```python
# Uso del patrón Composite
hoja1 = Hoja("Hoja 1")
hoja2 = Hoja("Hoja 2")

compuesto1 = Compuesto("Compuesto 1")
compuesto1.agregar(hoja1)
compuesto1.agregar(hoja2)

hoja3 = Hoja("Hoja 3")

compuesto2 = Compuesto("Compuesto 2")
compuesto2.agregar(compuesto1)
compuesto2.agregar(hoja3)

print(compuesto2.operacion())
```



- ***Command***:



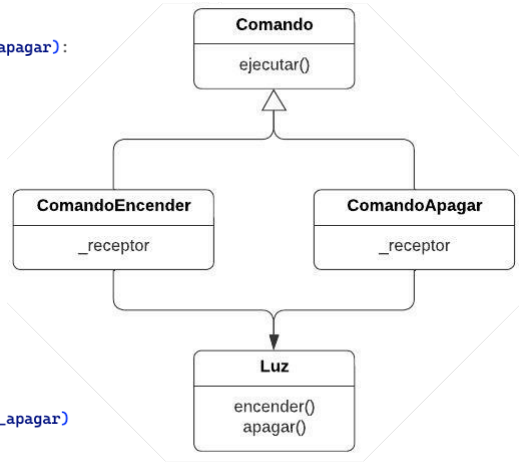
El patrón de diseño Command, también conocido como Orden o Comando, es un patrón de comportamiento que encapsula una solicitud como un objeto, lo que permite parametrizar clientes con diferentes solicitudes, encolar o registrar solicitudes y soportar operaciones reversibles.

Ejemplo:

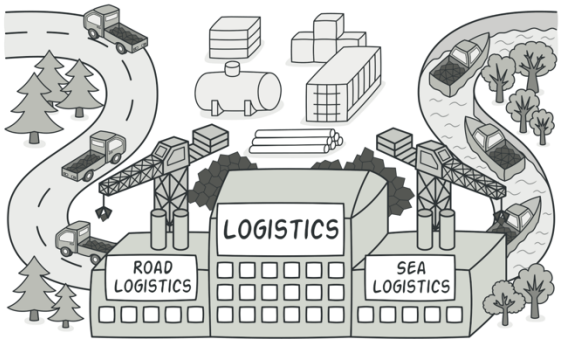
1. Sistema de gestión de órdenes en un restaurante: En un sistema de gestión de órdenes de un restaurante, se puede utilizar el patrón Command para representar las diferentes órdenes que los clientes realizan. Cada tipo de orden (como una orden de comida, una orden de bebida, una orden de postre, etc.) se puede encapsular en un objeto Command. El cliente (mesero o sistema automatizado) puede enviar los comandos a una cola de órdenes o registrarlos para su posterior procesamiento. Luego, los comandos se pueden ejecutar cuando sea necesario, y también se pueden deshacer o rehacer si es necesario.

```python
# Comando (Command)
class Comando(ABC):
    @abstractmethod
    def ejecutar(self):
        pass

# Receptor
class Luz:
    def encender(self):
        print("Luz encendida")

    def apagar(self):
        print("Luz apagada")

# Comandos concretos (Concrete Commands)
class ComandoEncender(Comando):
    def __init__(self, receptor):
        self._receptor = receptor

    def ejecutar(self):
        self._receptor.encender()

class ComandoApagar(Comando):
    def __init__(self, receptor):
        self._receptor = receptor

    def ejecutar(self):
        self._receptor.apagar()
```

```python
# Invocador
class ControlRemoto:
    def __init__(self):
        self._comando_encender = None
        self._comando_apagar = None

    def set_comandos(self, comando_encender, comando_apagar):
        self._comando_encender = comando_encender
        self._comando_apagar = comando_apagar

    def encender_luz(self):
        self._comando_encender.ejecutar()

    def apagar_luz(self):
        self._comando_apagar.ejecutar()

# Uso del patrón Command
luz = Luz()
comando_encender = ComandoEncender(luz)
comando_apagar = ComandoApagar(luz)

control_remoto = ControlRemoto()
control_remoto.set_comandos(comando_encender, comando_apagar)

control_remoto.encender_luz()
control_remoto.apagar_luz()
```



- **Factory method**:



El patrón de diseño Factory Method, también conocido como Método de Fábrica, es un patrón creacional que proporciona una interfaz para crear objetos, pero permite a las subclases decidir qué clase concreta instanciar. El Factory Method define un método abstracto en una clase base, que las subclases implementan para crear objetos de tipos diferentes pero relacionados.

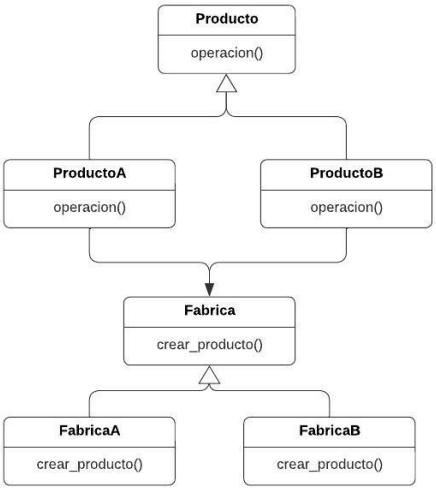
Ejemplo:

1. <u>Creación de objetos de diferentes tipos en un juego</u>: Supongamos que estás desarrollando un videojuego en el que los jugadores pueden seleccionar diferentes personajes. Cada personaje tiene sus propias habilidades y características. Puedes utilizar el patrón Factory Method para crear objetos de los diferentes tipos de personajes. La clase base tendría un método abstracto llamado “crearPersonaje()”, y cada subclase (por ejemplo, “Cazador”, “Mago” o “Guerrero”) implementaría ese método para crear una instancia del personaje correspondiente.

```python
from abc import ABC, abstractmethod

# Producto
class Producto(ABC):
    @abstractmethod
    def operacion(self):
        pass

# Productos concretos
class ProductoA(Producto):
    def operacion(self):
        return "Operación de ProductoA"

class ProductoB(Producto):
    def operacion(self):
        return "Operación de ProductoB"
```

```python
# Fábrica (Creator)
class Fabrica(ABC):
    @abstractmethod
    def crear_producto(self):
        pass

# Fábricas concretas
class FabricaProductoA(Fabrica):
    def crear_producto(self):
        return ProductoA()

class FabricaProductoB(Fabrica):
    def crear_producto(self):
        return ProductoB()
```

```python
# Uso del patrón Factory Method
fabrica_a = FabricaProductoA()
producto_a = fabrica_a.crear_producto()
print(producto_a.operacion())

fabrica_b = FabricaProductoB()
producto_b = fabrica_b.crear_producto()
print(producto_b.operacion())
```



- **Atributos de calidad / requisitos no funcionales**:

  - Performance (tiempo en respuesta, latencia, etc.).
  - Escalabilidad.
  - Disponibilidad.
  - Seguridad.
  - Continuidad operacional (recuperación).
  - Accesibilidad.
  - Monitoreo.

- **Modelo 4C:** Context > Containers > Components > Code

- **Patrones arquitectónicos**

  - **Client-Server**:

    - El cliente es la interfaz de usuario o la aplicación que se ejecuta en el dispositivo del usuario, como una computadora, un teléfono móvil o una tableta. El cliente envía solicitudes al servidor y recibe las respuestas correspondientes. Puede ser una aplicación de escritorio, una aplicación web o una aplicación móvil.

    - El servidor es el componente que procesa las solicitudes del cliente y envía las respuestas correspondientes. Puede ser una computadora o un grupo de computadoras dedicadas a ejecutar aplicaciones y servicios para múltiples clientes. El servidor gestiona los recursos y la lógica de negocio, y proporciona la funcionalidad requerida por los clientes.

  - **Arquitectura de Capas**:

    - Modelo de diseño de software en el que se divide una aplicación en capas lógicas o niveles, cada una con una responsabilidad específica. Cada capa se comunica con las capas adyacentes utilizando interfaces bien definidas, lo que permite la modularidad, el mantenimiento y la escalabilidad del sistema.

- **Verificación vs Validación**:

  - Verificación: Se centra en ¿estamos construyendo correctamente el software? y se realiza a través de revisiones, inspecciones, pruebas estáticas y análisis del código. Es decir, se verifica que el software se haya desarrollado según los estándares y especificaciones previamente definidos.

    - Análisis estático: Análisis de código (sin correrlo) mediante herramientas que permiten detectar elementos sospechosos.

- Inspección formal de código: Código revisado por equipo de pares con el objetivo de detectar la mayor cantidad de defectos.
- Testing: Proceso de ejecutar un programa con el objetivo de encontrar un error.
  - Tests no funcionales: Se prueba performance, carga, confiabilidad, etc.
  - Tests de usabilidad: Observación de personas reales usando el software.
  - A/B Testing: Se ponen a prueba 2 versiones, se ve cual es mejor.

- Validación: Se centra en ¿estamos construyendo el software correcto? y se lleva a cabo mediante pruebas dinámicas, simulaciones, prototipos y revisiones de requisitos. Se valida que el software cumpla con los requisitos y expectativas del usuario final y sea adecuado para su uso previsto.