

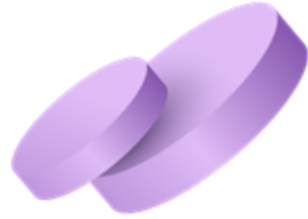


# Patrones de Comportamiento

State, Strategy, Template Method & Visitor



# Patrones de comportamiento



Dentro del area de ingenieria de software, un patron de comportamiento se refiere a una solución general y reutilizable para problemas recurrentes en el diseño de software.

Estos patrones tienen el objetivo de capturar las mejores practicas y experiencias previas de los desarrolladores y proporcionar un enfoque estructurado para la resolucion de problemas comunes.

# Patrones de comportamiento

01

State

02

Strategy

03

Template Method

04

Visitor



The background features a light gray central rectangle. Surrounding it are various colorful geometric shapes: a yellow 3D bar and a pink gear in the top-left; a pink square with the number '01' in the top-center; a pink gear on a purple cylinder in the bottom-right; and several other pink, orange, and teal shapes at the corners.

01

State

# Patrón State

## Diseño



Se caracteriza por cambiar el comportamiento de la aplicación según su estado. Para lograr esto, necesitas crear una serie de clases que representen los diferentes estados por los que puede pasar la aplicación; es decir, cada estado en el que puede navegar una aplicación requiere una clase.

## Idea



Un programa solo puede tener una cantidad limitada de espacio en un momento dado. En cada estado único, el programa se comporta de manera diferente y puede cambiar instantáneamente de un estado a otro. Estas reglas de cambio, llamadas transiciones, también son definitivas y predeterminadas.

# Aplicabilidad



Se puede utilizar el patrón State cuando se tenga un objeto que se comporte de manera diferente a su estado actual, la cantidad de estados sea enorme y el código específico del estado cambie con frecuencia.



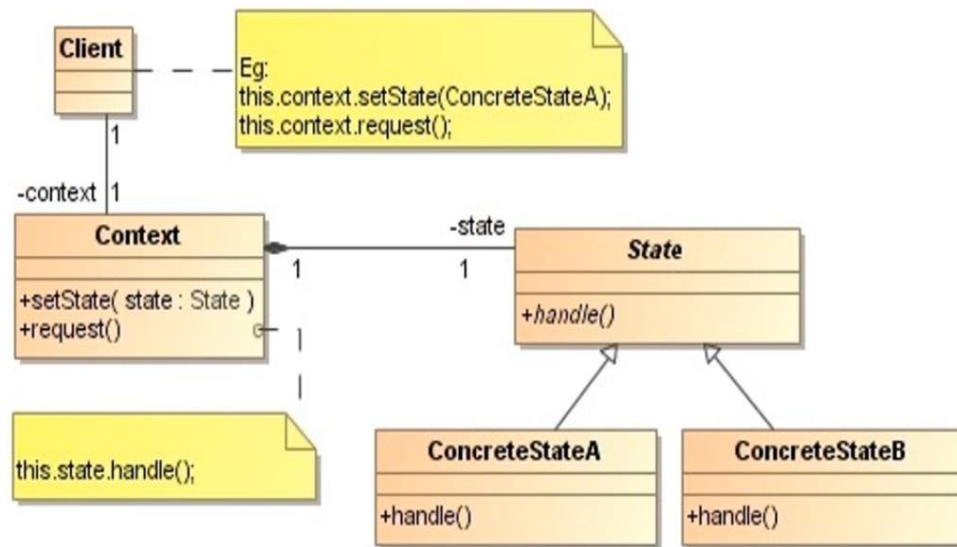
Cuando se tenga una clase contaminada con condicionales enormes que cambian el comportamiento de la clase en función de los valores actuales de los campos de clase.



# Estructura



1. La clase Contexto almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado.
2. La interfaz State declara métodos específicos del estado.
3. Los Estados concretos proporcionan implementaciones de sus propios métodos específicos del estado.



# Ejemplo de Estados de una clase documento



## Borrador

Se encarga de mover el documento al estado de moderación.



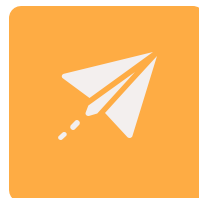
## Moderación

Hace publico el documento, con la condicion de que el usuario actual sea un administrador.



## Publicado

No hace nada en absoluto.





The background is a light gray rectangle with rounded corners, tilted slightly to the right. It is surrounded by various colorful geometric shapes: a yellow 3D bar and a pink gear in the top left; a pink gear on a purple cylinder in the bottom right; and several pink, orange, and teal shapes at the corners.

02

# Strategy

# Estados



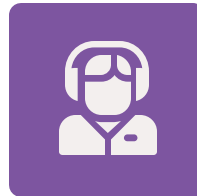
## Desarrollo ágil

El desarrollo iterativo e incremental se enfatiza mediante enfoques ágiles como Scrum y Kanban.



## Colaboración y comunicación

Para los equipos de ingeniería de software, la colaboración y la comunicación efectivas son esenciales.

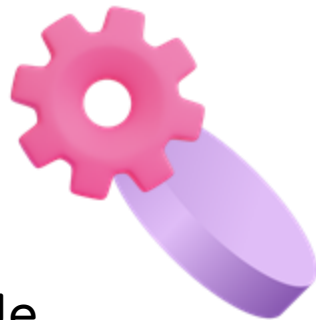


## Desarrollo por pruebas

TDD es una metodología en la que las pruebas se crean antes de la implementación del código.

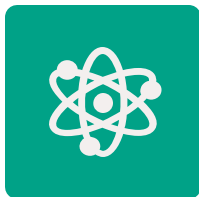


# Estados



## Patrones de diseño

Los patrones de diseño proporcionan soluciones reutilizables a problemas típicos de diseño de software.



## Integración e implementación

Los procedimientos de CI/CD implican la fusión constante de actualizaciones de código, la realización de pruebas automatizadas y la entrega continua de software.



## Control de versiones

Los equipos pueden rastrear y administrar con éxito los cambios de código gracias a las herramientas de control de versiones como Git.



# Documentación



- La documentación clara es esencial para comprender el sistema de software y mantenerlo en el tiempo.
- Esta documentación debe incluir comentarios de código, manuales de usuario y especificaciones técnicas.
- La mejora de la capacidad de mantenimiento del programa y la transferencia de conocimientos son el resultado de una documentación clara.
- El aprendizaje continuo es crucial porque la ingeniería de software es un campo en continuo crecimiento.

The background is a light gray rectangle with rounded corners, tilted slightly to the right. It is surrounded by various colorful geometric shapes: a yellow 3D bar and a pink gear in the top left; a pink gear on a purple cylinder in the bottom right; and several pink, orange, and teal shapes at the corners.

03

# Template Method

# Template Method

## Técnica



El patrón de diseño Template Method es una técnica que permite definir la estructura de un algoritmo en una clase base, mientras permite que las subclasses implementen pasos específicos sin alterar dicha estructura.

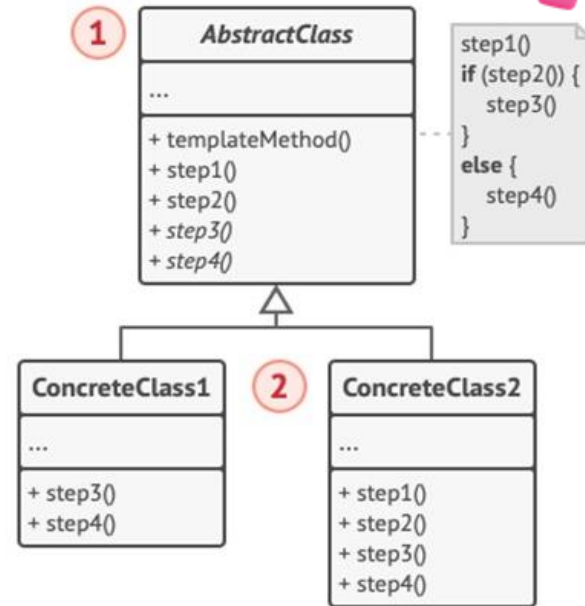
## Funcionamiento



El funcionamiento del patrón Template Method es el siguiente: se crea una clase base abstracta que contiene un método plantilla, el cual define la secuencia de pasos del algoritmo.

# Estructura

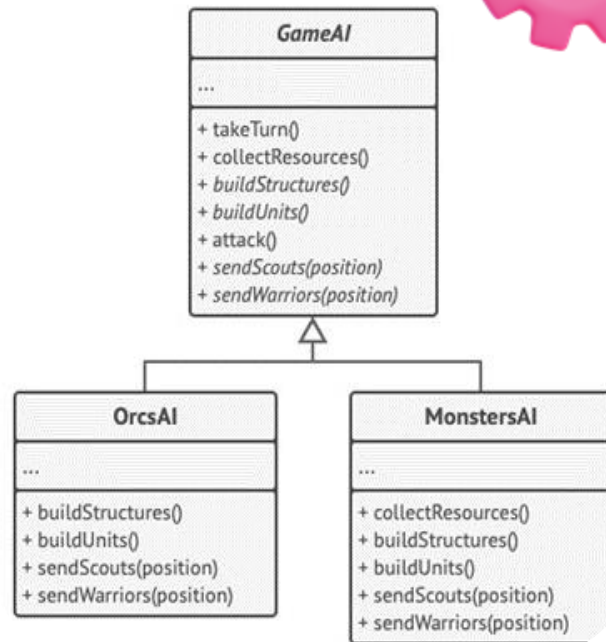
1. La Clase Abstracta declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse abstractos o contar con una implementación por defecto.
2. Las Clases Concretas pueden sobrescribir todos los pasos, pero no el propio método plantilla.





# Implementación

Un ejemplo de implementación del patrón Template Method es el siguiente: supongamos que tenemos un juego de estrategia en el que diferentes razas (orcos, humanos, monstruos) tienen su propia inteligencia artificial (IA) para tomar decisiones. Cada raza tiene una estructura de unidades y edificios similares, pero con comportamientos específicos.



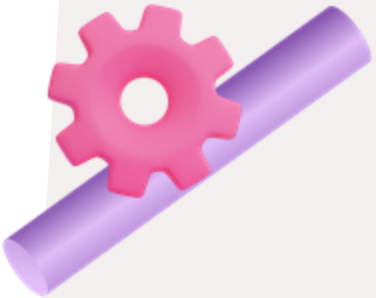


The background is a light gray rectangle with rounded corners, tilted slightly to the right. It is surrounded by various colorful geometric shapes: a yellow 3D bar and a pink gear in the top-left; a pink gear on a purple cylinder in the bottom-right; and several pink, orange, and teal polygons in the corners.

04


Visitor

# Visitor



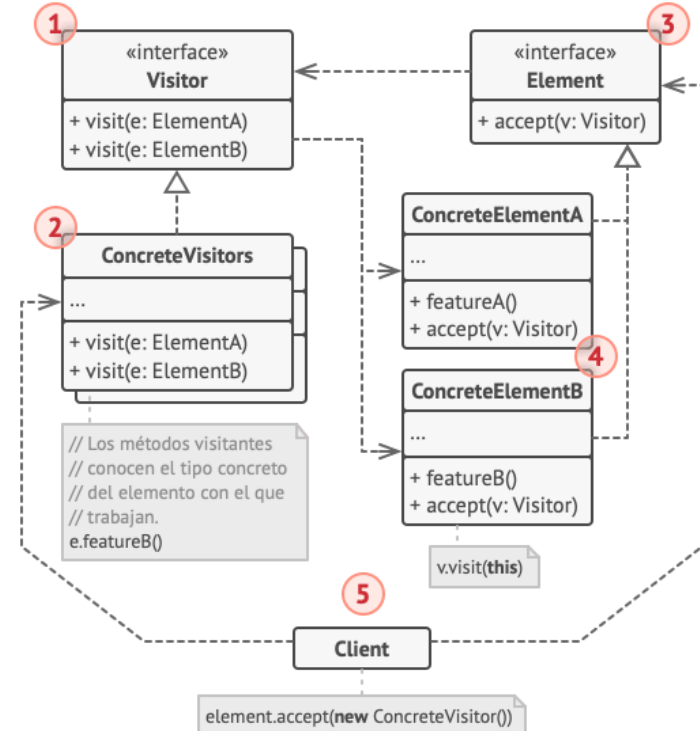
El patrón Visitor es un patrón de diseño de software que se utiliza para separar la estructura de un objeto de sus operaciones o algoritmos. Proporciona una forma de agregar operaciones adicionales a una clase sin modificar su estructura.

El propósito principal del patrón Visitor es separar la lógica de la operación de los objetos sobre los que opera. Esto significa que se pueden agregar nuevas operaciones sin modificar las clases existentes. El patrón Visitor utiliza el polimorfismo para lograr esto.



# Estructura

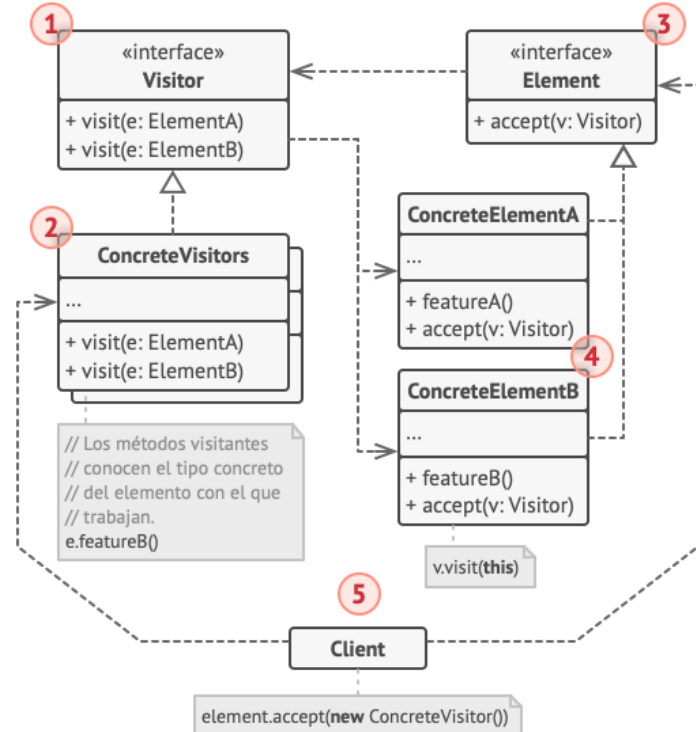
1. La **interfaz Visitante** declara un grupo de métodos visitantes que pueden tomar elementos concretos de una estructura de objetos como argumentos.
2. Cada **Visitante Concreto** implementa varias versiones de los mismos comportamientos, personalizadas para las distintas clases de elemento concreto.
3. La **interfaz Elemento** declara un método para “aceptar” visitantes. Este método deberá contar con un parámetro declarado con el tipo de la interfaz visitante.



# Estructura

4. Cada **Elemento Concreto** debe implementar el método de aceptación.

5. El **Cliente** representa normalmente una colección o algún otro objeto complejo (por ejemplo, un árbol **Composite**).



# Aplicabilidad

- Utilizar el patrón Visitor cuando se necesite realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).
- Utilizar el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.
- Utilizar el patrón cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.



## PROS

- Principio de abierto/cerrado.
- Principio de responsabilidad única.
- Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos.



## CONTRAS

- Debes actualizar todos los visitantes cada vez que una clase se añade o elimina de la jerarquía de elementos.
- Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar.

# Relaciones con Otros Patrones

- Se puede tratar al patrón Visitor como una versión más potente del patrón **Command**. Sus objetos pueden ejecutar operaciones sobre varios objetos de distintas clases.
- Se puede utilizar el patrón Visitor para ejecutar una operación sobre un árbol **Composite** entero.
- Se puede utilizar Visitor en conjunto con **Iterator** para recorrer una estructura de datos compleja y ejecutar alguna operación sobre sus elementos, incluso aunque todos tengan clases distintas.



# Implementación

Se definen las clases **Visitante** y **Elemento** como interfaces utilizando clases base sin métodos implementados. Luego, se implementan las clases **ElementoA** y **ElementoB**, donde se define el método **aceptar** y las operaciones específicas de cada clase.

```
Visitor.py > ...
1  # Paso 1: Declara la interfaz Visitante con métodos para cada clase de elemento concreto
2  # existente en el programa.
3  class Visitante:
4      def visitarElementoA(self, elementoA):
5          pass
6
7      def visitarElementoB(self, elementoB):
8          pass
9
10 # Paso 2: Declara la interfaz de Elemento.
11 class Elemento:
12     def aceptar(self, visitante):
13         pass
14
15 # Paso 3: Implementa los métodos de aceptación en las clases de elemento concreto.
16 class ElementoA(Elemento):
17     def aceptar(self, visitante):
18         visitante.visitarElementoA(self)
19
20     def operacionA(self):
21         # Lógica específica de ElementoA
22         print("Operación A en ElementoA")
23
24 class ElementoB(Elemento):
25     def aceptar(self, visitante):
26         visitante.visitarElementoB(self)
27
28     def operacionB(self):
29         # Lógica específica de ElementoB
30         print("Operación B en ElementoB")
```

# Implementación

Después, se crea la clase **VisitanteConcreto** que hereda de **Visitante** e implementa los métodos visitantes correspondientes.

Por último, en el bloque `if __name__ == "__main__":`, se crea una instancia de **ElementoA**, una instancia de **ElementoB**, una instancia de **VisitanteConcreto** y se llama al método **aceptar** en cada elemento pasando el visitante correspondiente.

```
42 # Paso 4: Implementa los métodos visitantes en una clase concreta visitante.
43 class VisitanteConcreto(Visitante):
44     def visitarElementoA(self, elementoA):
45         # Implementación del método visitante para ElementoA
46         elementoA.operacionA()
47
48     def visitarElementoB(self, elementoB):
49         # Implementación del método visitante para ElementoB
50         elementoB.operacionB()
```

```
42 # Paso 5: Cliente que crea objetos visitantes y los pasa a través de métodos de aceptación.
43 if __name__ == "__main__":
44     elementoA = ElementoA()
45     elementoB = ElementoB()
46
47     visitante = VisitanteConcreto()
48
49     elementoA.aceptar(visitante)
50     elementoB.aceptar(visitante)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Microsoft Windows [Version 10.0.22621.1848]  
(c) Microsoft Corporation. All rights reserved.

C:\Users\S\_G\_C\_M\Desktop\U T P>C:/Users/S\_G\_C\_M/AppData/Local/Microsoft/WindowsApps/python3.10.exe  
Operación A en ElementoA  
Operación B en ElementoB



# Conclusión

El patrón de Visitor nos proporciona una forma de separar algoritmos de la estructura de los objetos visitados, lo que nos permite agregar nuevas operaciones sin modificar.

Template Method nos ayuda a definir el esqueleto del algoritmo de una clase base, permitiendo que las subclases proporcionen implementaciones específicas de algunos pasos del algoritmo.

El patron State permite que un objeto altere su comportamiento cuando su estado interno cambia, lo que lleva a una apariencia diferente de su clase.

El patrón de Strategy permite definir una familia de algoritmos intercambiables y encapsula cada uno de ellos, lo que permite que los algoritmos varíen independientemente de los clientes que los utilizan.

# Gracias!

