Universidad del Valle de Guatemala
Facultad de Ingeniería
Ingeniería de Software 1
Sección 30
Profesor: Erick Francisco Marroquín Rodríguez

“Tarea investigativa: Patrones de diseño”

Andy Fuentes 22944
Jorge Lopez 221038
Diederich Solis 22952
Davis Roldan 22672
Erick Barrera 22924

6 de marzo del 2024, Guatemala de la Asunción

A. Introducción

En el vasto panorama del desarrollo de software, la aplicación de patrones de diseño se presenta como un enfoque fundamental para optimizar la creación de sistemas complejos y resolver desafíos recurrentes de manera eficiente. Este trabajo investigativo se sumerge en la comprensión y aplicación de tres patrones esenciales: Facade, Builder y State. A medida que exploramos estas estructuras, descubriremos cómo contribuyen significativamente a la calidad del software al simplificar la complejidad, fomentar la modularidad y mejorar la mantenibilidad del código.

B. Resumen

C. Patrón Facade

Características:

1. **Interfaz simplificada:** El Facade ofrece una interfaz única y simple para interactuar con sistemas complejos, haciendo que estos sean más accesibles y fáciles de usar.

2. **Reducción de complejidad:** Al proporcionar un punto de acceso simplificado, los clientes pueden evitar lidiar directamente con la complejidad interna de los subsistemas.

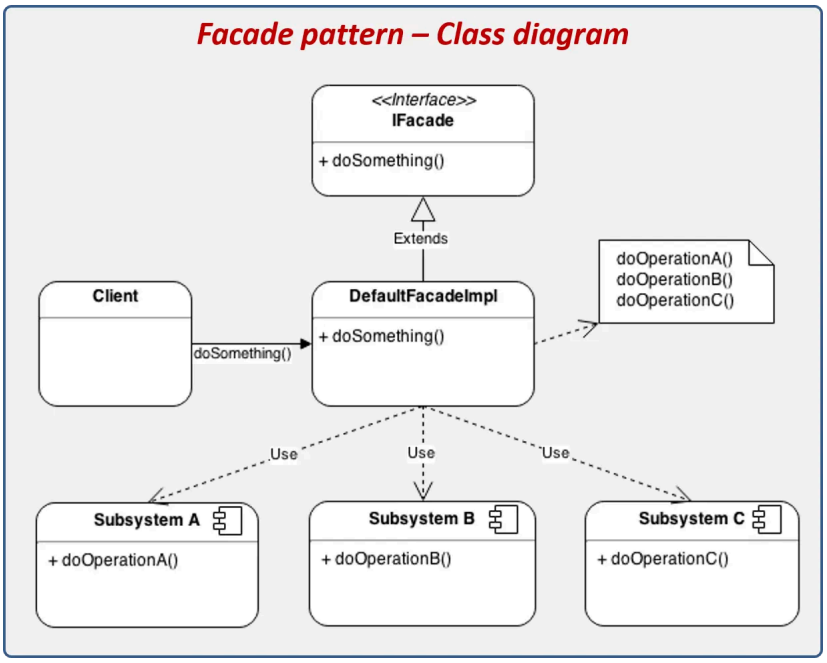
3. **Facilita la comunicación:** Simplifica la manera en que el código cliente se comunica con las partes complejas del sistema.

4. **Desacoplamiento:** Mejora la independencia entre sistemas, permitiendo cambios internos sin afectar a los usuarios del Facade.

5. **Organización del código:** Agrupa interfaces complejas, mejorando la legibilidad y mantenibilidad del código.

Ejemplos:

Diagrama de ejemplo:



1. **Sistema de Reservas de Viaje:**

   Funcionalidad Completa: Puede haber sistemas complejos que gestionan reservas de vuelos, hoteles y alquiler de coches.

   Fachada: Un servicio de reservas en línea que ofrece una interfaz única para buscar, seleccionar y confirmar vuelos, alojamientos y transporte.

2. **Biblioteca Digital:**

   Funcionalidad Completa: Una biblioteca digital puede tener sistemas de gestión de contenido, búsqueda avanzada, control de acceso, etc.

   Fachada: Una interfaz web simple que permite a los usuarios buscar, ver y descargar libros electrónicos sin necesidad de entender la complejidad del sistema subyacente.

3. **Sistema de Gestión de Proyectos:**

   Funcionalidad Completa: Herramientas que gestionan tareas, asignaciones de recursos, seguimiento de tiempo, etc.

   Fachada: Una interfaz unificada que permite a los usuarios crear, asignar y hacer un seguimiento de las tareas sin lidiar con la complejidad total del sistema.

4. **Plataforma de Compras en Línea:**

   Funcionalidad Completa: Manejo de inventario, procesamiento de pagos, seguimiento de envíos, etc.

Fachada: Una interfaz de usuario amigable que permite a los compradores buscar productos, agregar al carrito y realizar pagos sin preocuparse por los detalles internos.

5. **Sistema Bancario en Línea:**

   Funcionalidad Completa: Transferencias de fondos, inversiones, préstamos, gestión de cuentas, etc.

   Fachada: Una aplicación web que ofrece a los usuarios una interfaz fácil para revisar saldos, hacer transferencias y pagar facturas sin necesidad de entender todos los aspectos del sistema bancario.

D. Patrón Builder Patterns

Características:

1. **Separación de construcción y representación:** El patrón Builder permite construir objetos paso a paso, ocultando los detalles de la construcción y la representación final del objeto. Esto permite que un objeto sea construido en varias etapas y en diferentes formas finales.

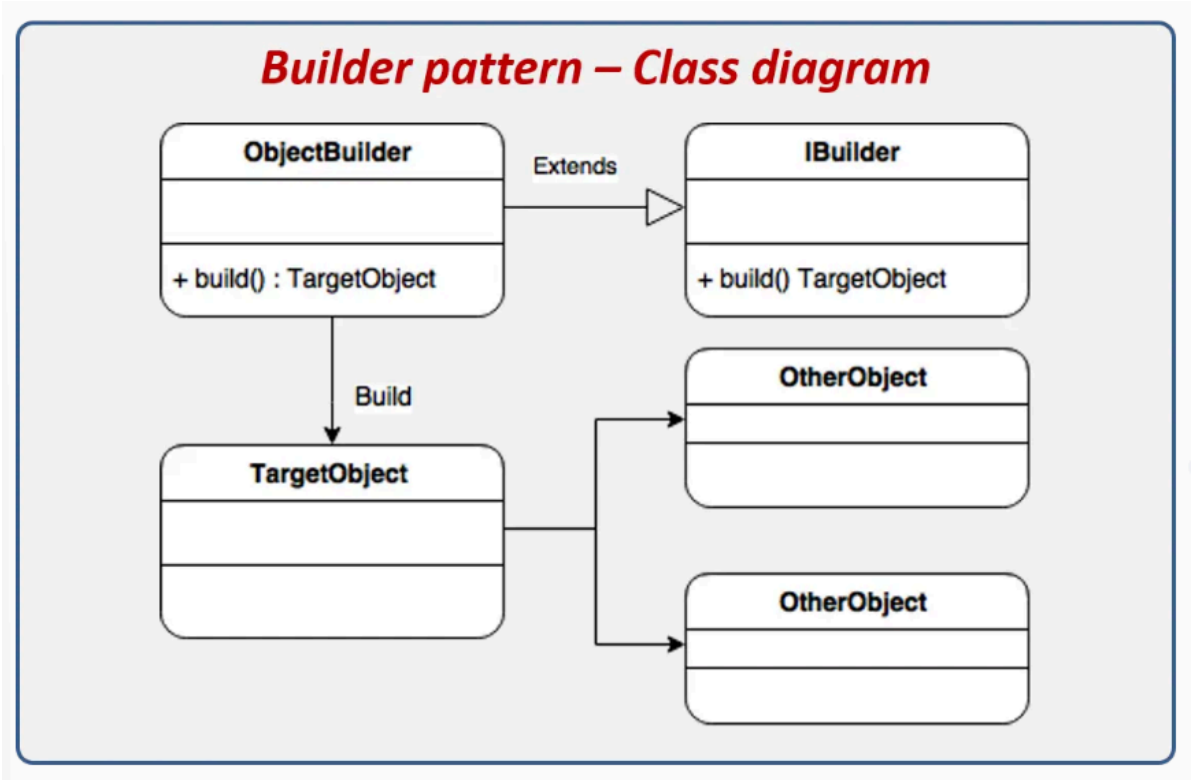
2. **Encapsulamiento**: Al utilizar el patrón Builder, los detalles de cómo un objeto complejo es ensamblado están encapsulados detrás de un objeto constructor. Esto permite modificar el proceso de construcción sin afectar el código cliente.

3. **Proceso de construcción**: El patrón proporciona un control completo sobre el proceso de construcción del objeto. Esto es útil cuando el proceso de construcción debe variar para crear diferentes representaciones del objeto.

4. **Uso del director**: Existe un objeto director que se utiliza para orquestar la construcción del objeto usando el builder específico. El director conoce los pasos necesarios para construir el objeto y los realiza en el orden correcto.

5. **Construcción de objetos complejos:** Es ideal para construir objetos complejos que tienen múltiples componentes o para cuando la construcción de un objeto requiere pasos específicos, posiblemente en un orden específico.

Ejemplos:



IBuilder: Este componente no es obligatorio en todos los casos, sin embargo, es buena práctica especificar una interface común que tendrán todos los Builder que definiremos en nuestra aplicación, puede ser una interface que defina únicamente el método build.

ObjectBuilder: Esta es la clase que utilizaremos para crear los TarjetObjet, esta clase debe de heredar de IBuilder e implementar el método build, el cual será utilizado para crear al TarjetObject. Como regla general todos los métodos de esta clase retornan a si mismo con la finalidad de agilizar la creación, esta clase por lo general es creada como una clase interna del TargetObject.

TarjetObjet: Representa el objeto que deseamos crear mediante el ObjectBuilder, ésta puede ser una clase simple o puede ser una clase muy compleja que tenga dentro más objetos.

OtherObjets: Representa los posibles objetos que deberán ser creados cuando el TarjetObject sea construido por el ObjectBuilder.

E. Patrón State

Características:

- Encapsulación del Estado: El patrón State encapsula los estados específicos en clases separadas. Cada estado es representado por una subclase concreta dentro de una jerarquía de clase que implementa una interfaz común de estado. Esto facilita la adición de nuevos estados sin modificar el código existente.

- Separación de Concerns: Separa el comportamiento relacionado con un estado particular en una clase dedicada a ese estado. Esto ayuda a mantener el código organizado, facilita la mantenibilidad y promueve la cohesión alta dentro de las clases de estado.

- Transiciones de Estado: Las transiciones de un estado a otro se manejan de manera más fluida, permitiendo que el objeto cambie su comportamiento dinámicamente al cambiar su estado interno. Las propias clases de estado pueden controlar las transiciones a otros estados, basadas en la lógica de la aplicación o en las acciones del usuario.

- Reduce condiciones: Al utilizar el patrón State, se reduce la necesidad de sentencias condicionales (if/else o switch/case) para controlar el comportamiento basado en el estado del objeto. En su lugar, el comportamiento específico del estado se encapsula dentro de las clases de estado.

- Sustitución de la herencia por Composición: El patrón favorece la composición sobre la herencia. En lugar de tener un objeto que hereda diferentes comportamientos y luego usar condicionales para decidir cuál usar, el objeto se compone con el estado correcto que representa su comportamiento actual.

- Fácil de Extender: Agregar nuevos estados o modificar el comportamiento de los estados existentes se vuelve más fácil, ya que cada estado está encapsulado en su propia clase. Esto cumple con el principio de abierto/cerrado, permitiendo que el sistema sea más fácil de extender sin modificar el código existente.

- Contexto: El patrón utiliza una clase de contexto que mantiene una referencia a un objeto de estado que refleja el estado actual del sistema. El contexto delega las solicitudes de los clientes al objeto de estado actual, el cual puede ser reemplazado por otro objeto de estado si es necesario cambiar el comportamiento del contexto.

Ejemplos:

Diagrama de ejemplo:

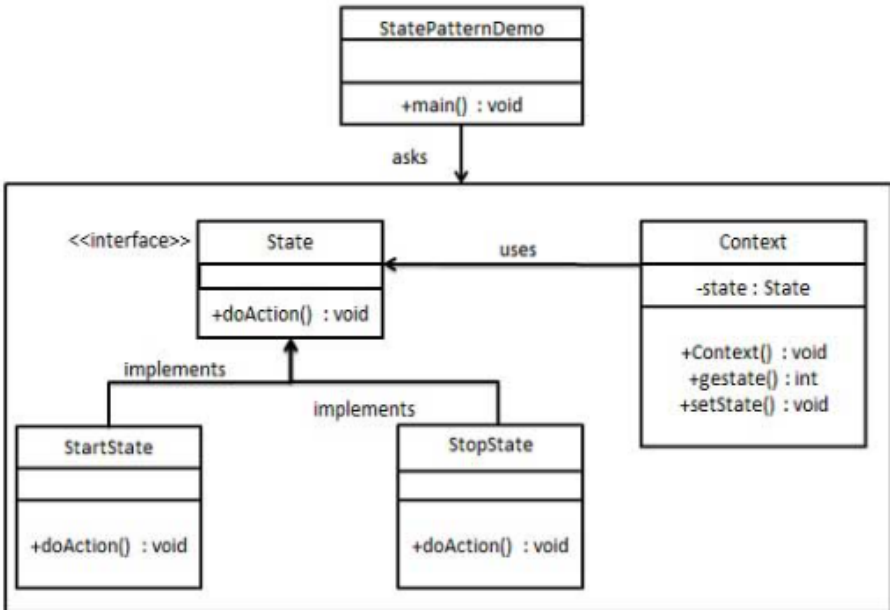
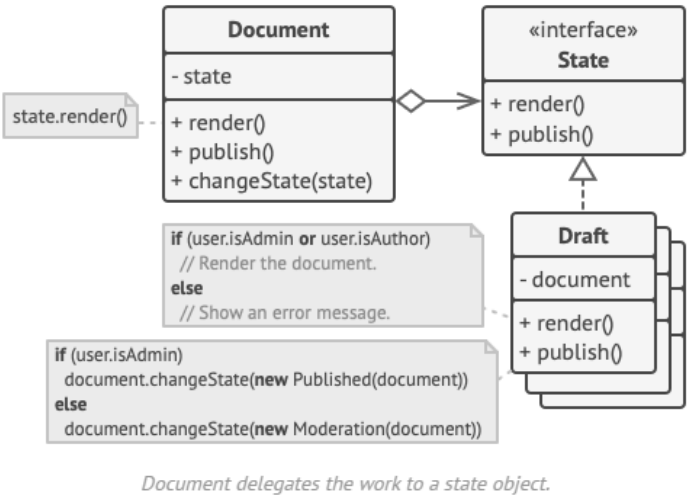


Diagrama a un ejemplo aplicado:



Document delegates the work to a state object.

1. **Sistemas de gestión de pedidos:** Para manejar los diferentes estados de un pedido (por ejemplo, nuevo, en proceso, enviado, entregado).

2. **Juegos:** Para gestionar los estados de los personajes o del juego mismo (por ejemplo, normal, potenciado, dañado).

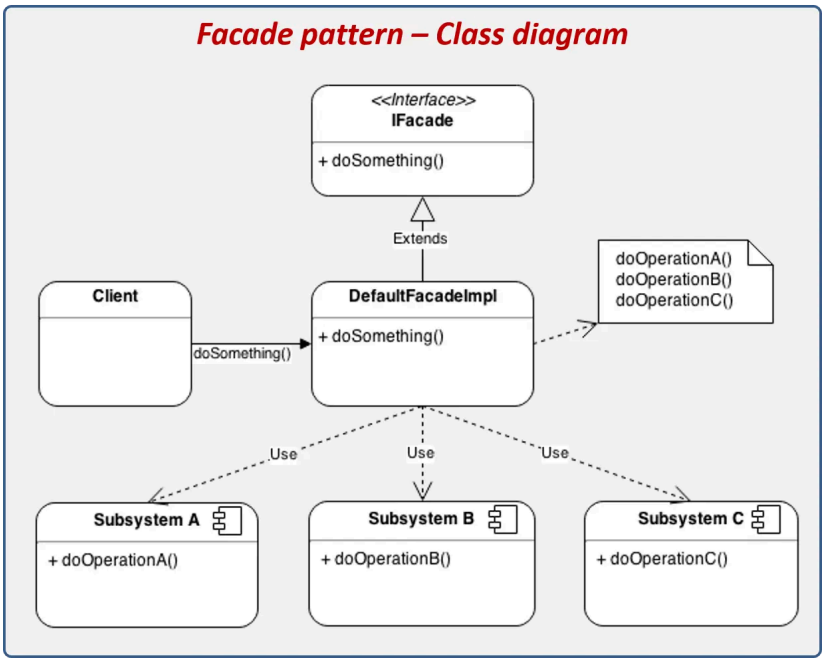
3. **Aplicaciones de flujo de trabajo:** Para controlar los estados de tareas o documentos en un proceso (por ejemplo, borrador, revisión, aprobado) como se puede revisar en el diagrama de ejemplo.

4. **Interfaces de usuario:** Para cambiar el comportamiento de los elementos de la interfaz según el estado (por ejemplo, activo, inactivo, deshabilitado).

5. **Sistemas de autenticación:** Para manejar los estados de una sesión de usuario (por ejemplo, no autenticado, autenticado, sesión caducada).

F. Tablas Patrones de diseño:

Facade:

Patrón Facade
INTENCIÓN
Simplificar el acceso a una serie de interfaces de un subsistema complejo, proporcionando una interfaz única simplificada.
CONOCIDO COMO
Facade
Motivo
Reducir la complejidad y las dependencias entre sistemas
APLICACIONES
Se utiliza cuando se quiere proporcionar una interfaz simple a un conjunto complejo de interfaces o cuando hay muchas dependencias entre clientes y las clases de implementación.
ESTRUCTURA

PARTICIPANTES
Facade: interfaz simplificada para interactuar con el subsistema. Subsistemas: conjuntos de clases que realizan la funcionalidad compleja.

COLABORACIONES
Los clientes comunican con el subsistema a través de la fachada, sin necesidad de manejar múltiples clases del subsistema.
CONSECUENCIAS
Mejora la usabilidad y reduce la complejidad del cliente, pero puede llevar a una fachada "monolítica" con demasiada funcionalidad.
IMPLEMENTACIÓN
Crear una clase fachada que agrega las clases del subsistema y proporciona métodos simples para sus operaciones.
CÓDIGO DE EJEMPLO

```
# Subsistema
class CPU:
    def encender(self):
        print("Encendiendo CPU")

    def apagar(self):
        print("Apagando CPU")

class Memoria:
    def cargar(self):
        print("Cargando memoria")

    def liberar(self):
        print("Liberando memoria")

class Disco:
    def leer(self):
        print("Leyendo disco")

    def escribir(self):
        print("Escribiendo en disco")

# Facade
class ComputadoraFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memoria = Memoria()
        self.disco = Disco()

    def encender(self):
        self.cpu.encender()
        self.memoria.cargar()
        self.disco.leer()

    def apagar(self):
        self.disco.escribir()
        self.memoria.liberar()
```

```
        self.cpu.apagar()

# Cliente
if __name__ == "__main__":
    computadora = ComputadoraFacade()
    computadora.encender()
    computadora.apagar()
```

USOS CONOCIDOS

Frameworks y bibliotecas que simplifican el uso de API complejas.

P. RELACIONADOS

Adapter, Mediator.

State

Patrón State

INTENCIÓN

permitir a un objeto modificar su comportamiento cuando su estado interno cambia, haciendo parecer que su clase ha cambiado

CONOCIDO COMO

State

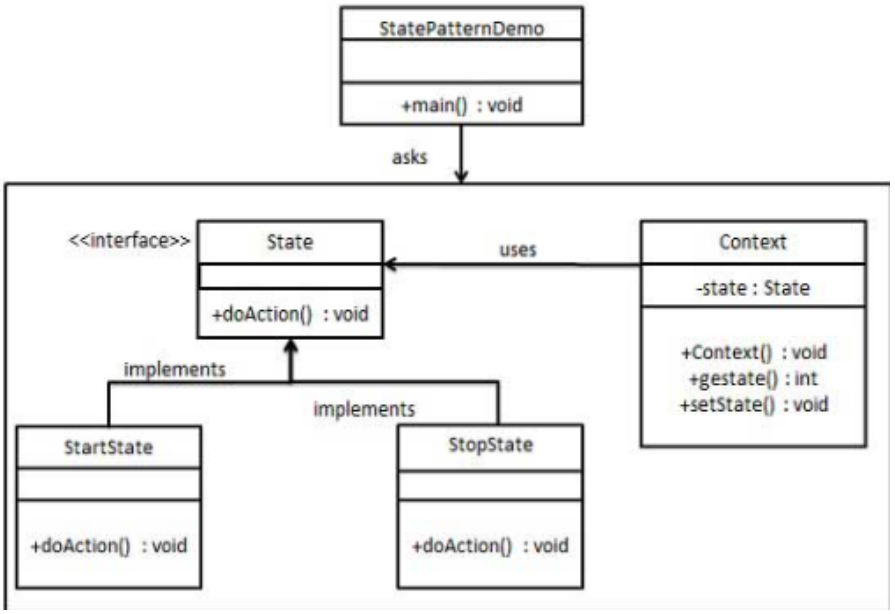
Motivo

manejar la complejidad en sistemas donde los objetos pueden encontrarse en diferentes estados con comportamientos específicos para cada estado.

APLICACIONES

El patrón State se aplica en situaciones donde el comportamiento de un objeto depende de su estado interno, y necesita cambiar en tiempo de ejecución. Es útil en sistemas de gestión de estados complejos, como flujos de trabajo, juegos con múltiples estados de personajes etc.

ESTRUCTURA



PARTICIPANTES

Contexto: Mantiene una referencia al estado actual y puede definir una interfaz para delegar comportamientos al estado.
State: Define una interfaz común para todos los estados concretos, encapsulando los comportamientos asociados con los estados del objeto.
Concrete States: Implementan la interfaz State, proporcionando los comportamientos específicos asociados a los estados del contexto.

COLABORACIONES

Las colaboraciones ocurren principalmente entre el Contexto, que mantiene una instancia de uno de los Concrete States que representa el estado actual del sistema, y los diferentes Concrete States. El Contexto delega las solicitudes al estado actual, y este maneja la solicitud basada en su implementación específica. Los estados pueden solicitar cambios en el Contexto, lo que puede llevar a que el Contexto cambie su estado actual por otro, reflejando así una transición de estados dentro del sistema.

CONSECUENCIAS

Localización de Comportamientos de Estado: Encapsula los comportamientos asociados a los estados en clases específicas, facilitando la modificación y adición de nuevos estados.
Hace Transiciones de Estado Explícitas: Las transiciones entre estados se vuelven más claras y explícitas al estar definidas en el código.
Estado y Transiciones Encapsulados: Permite cambiar el comportamiento de un objeto durante la ejecución mediante el cambio de su estado interno, sin necesidad de grandes condicionales.
Puede incrementar el Número de Clases: La implementación de este patrón puede resultar en un aumento del número de clases, ya que cada estado se representa como una clase independiente.

IMPLEMENTACIÓN

Para implementar el patrón State, se crea una clase "Contexto" que tiene un estado actual. Este estado está representado por una interfaz "Estado" y múltiples clases concretas que implementan esta interfaz, cada una correspondiendo a un posible estado del Contexto. El Contexto delega las operaciones a la instancia de "Estado" actual, permitiendo cambiar su comportamiento al cambiar

el estado actual por otro, modificando así la clase de estado concreto a la que se delegan estas operaciones.

CÓDIGO DE EJEMPLO

```
from __future__ import annotations
from abc import ABC, abstractmethod


class Context:
    """
    The Context defines the interface of interest to clients. It also maintains
    a reference to an instance of a State subclass, which represents the current
    state of the Context.
    """

    _state = None
    """
    A reference to the current state of the Context.
    """

    def __init__(self, state: State) -> None:
        self.transition_to(state)

    def transition_to(self, state: State):
        """
        The Context allows changing the State object at runtime.
        """

        print(f"Context: Transition to {type(state).__name__}")
        self._state = state
        self._state.context = self

    """
    The Context delegates part of its behavior to the current State object.
    """

    def request1(self):
        self._state.handle1()

    def request2(self):
        self._state.handle2()


class State(ABC):
    """
    The base State class declares methods that all Concrete State should
    implement and also provides a backreference to the Context object,
    associated with the State. This backreference can be used by States to
    transition the Context to another State.
    """

    @property
    def context(self) -> Context:
```

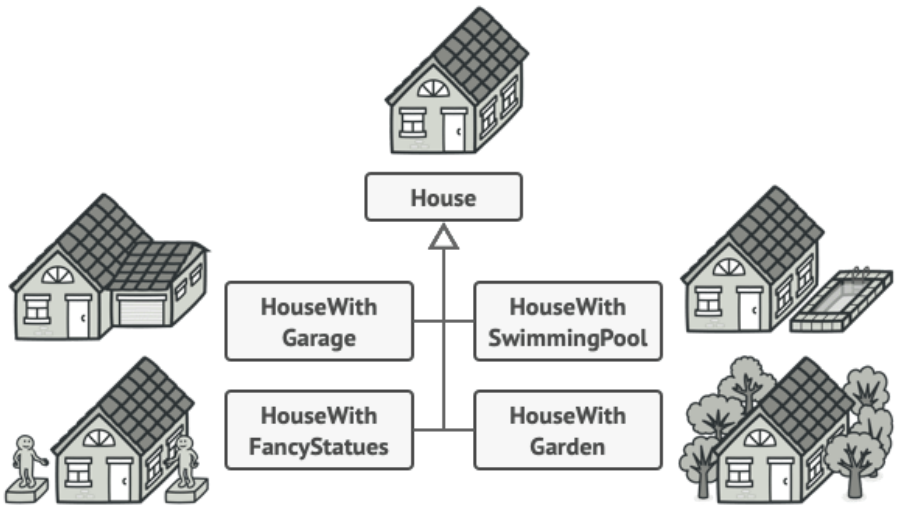
```
        return self._context

    @context.setter
    def context(self, context: Context) -> None:
        self._context = context

    @abstractmethod
    def handle1(self) -> None:
        pass

    @abstractmethod
    def handle2(self) -> None:
        pass


"""
Concrete States implement various behaviors, associated with a state of the
Context.
"""


class ConcreteStateA(State):
    def handle1(self) -> None:
        print("ConcreteStateA handles request1.")
        print("ConcreteStateA wants to change the state of the context.")
        self.context.transition_to(ConcreteStateB())

    def handle2(self) -> None:
        print("ConcreteStateA handles request2.")


class ConcreteStateB(State):
    def handle1(self) -> None:
        print("ConcreteStateB handles request1.")

    def handle2(self) -> None:
        print("ConcreteStateB handles request2.")
        print("ConcreteStateB wants to change the state of the context.")
        self.context.transition_to(ConcreteStateA())


if __name__ == "__main__":
    # The client code.

    context = Context(ConcreteStateA())
    context.request1()
    context.request2()
```

USOS CONOCIDOS

se usa en interfaces de usuario para controlar el estado de elementos gráficos, en juegos para manejar estados de personajes o niveles, en aplicaciones de gestión de pedidos para seguir el progreso de los pedidos, en sistemas de control de acceso para gestionar diferentes estados de autenticación, y en aplicaciones de flujo de trabajo para manejar el estado de tareas o documentos

a lo largo de diferentes etapas de procesamiento. Estos usos permiten cambios dinámicos en el comportamiento de objetos basados en su estado interno.
P. RELACIONADOS
Strategy, Command, Observer

Patterns

Patrón Builder Patterns
INTENCIÓN
tiene la intención de separar la construcción de un objeto complejo de su representación, de tal manera que el mismo proceso de construcción pueda crear diferentes representaciones.
CONOCIDO COMO
Builder
Motivo
Es conocido también por resolver el problema de los "Telescoping Constructor Anti-pattern" o el anti-patrón de constructores telescópicos. Este antipatrón ocurre cuando un objeto requiere ser instanciado con una gran cantidad de parámetros
APLICACIONES
El patrón de diseño Builder es ampliamente utilizado en software de aplicación cuando se necesita flexibilidad y claridad en la creación de objetos complejos. Su aplicación mejora la legibilidad del código, facilita el mantenimiento y promueve la eficiencia en el proceso de desarrollo de software.
ESTRUCTURA

PARTICIPANTES

Builder: Esta es la interfaz que define todos los pasos necesarios para construir un producto. Los pasos pueden incluir la construcción de diferentes partes del producto.
Concrete Builder: Implementa la interfaz Builder y proporciona una implementación concreta de los pasos de construcción.
Director: Es opcional y se encarga de orquestar la construcción del objeto usando el Builder. Define el orden en el que se deben llamar los pasos de construcción para producir determinados objetos.

COLABORACIONES

El cliente inicia el proceso, eligiendo un constructor específico para el tipo de objeto deseado. Si se utiliza un director, este guía al constructor a través de los pasos necesarios. El constructor trabaja en la construcción del objeto, y al final, el cliente recibe el producto terminado.

CONSECUENCIAS

el patrón Builder ofrece una solución robusta para construir objetos complejos, aumentando la flexibilidad y la claridad del código, pero a costa de aumentar la complejidad y la cantidad de código necesario.

IMPLEMENTACIÓN

Define la interfaz Builder: Crea una interfaz Builder que declare métodos para construir las diferentes partes del objeto complejo.

Crea ConcreteBuilders: Implementa la interfaz Builder con clases concretas que definan la construcción de las partes específicas del objeto complejo. Cada ConcreteBuilder construye una versión diferente del objeto.

Opcional - Usa un Director: Si necesitas, crea una clase Director que tome una instancia de Builder y ejecute los métodos necesarios en el orden correcto para construir el objeto. El director sabe cómo construir el objeto utilizando el builder, pero no los detalles específicos de cada parte.

Define el Producto: Los ConcreteBuilders construirán el objeto final, paso a paso. El producto es lo que se construye usando el patrón Builder.

Cliente construye el objeto: El cliente crea un objeto ConcreteBuilder, opcionalmente lo pasa a un Director, y luego utiliza el builder para construir el objeto paso a paso (directamente o a través del director). Al final, el cliente puede recuperar el objeto construido del builder.

CÓDIGO DE EJEMPLO

```
from abc import ABC, abstractmethod

class CarBuilder(ABC):
    @abstractmethod
    def reset(self):
        pass

    @abstractmethod
    def set_seats(self, number):
        pass

    @abstractmethod
```

```
    def set_engine(self, engine):
        pass

    @abstractmethod
    def set_trip_computer(self, has_trip_computer):
        pass

    @abstractmethod
    def set_gps(self, has_gps):
        pass



class SedanCarBuilder(CarBuilder):
    def __init__(self):
        self.car = {}

    def reset(self):
        self.car = {}

    def set_seats(self, number):
        self.car['seats'] = number

    def set_engine(self, engine):
        self.car['engine'] = engine

    def set_trip_computer(self, has_trip_computer):
        self.car['trip_computer'] = has_trip_computer

    def set_gps(self, has_gps):
        self.car['gps'] = has_gps

    def get_result(self):
        return self.car




class CarDirector:
    def __init__(self, builder):
        self._builder = builder

    def construct_sports_car(self):
        self._builder.reset()
        self._builder.set_seats(2)
        self._builder.set_engine('SportsEngine')
        self._builder.set_trip_computer(True)
        self._builder.set_gps(True)

    def construct_family_car(self):
        self._builder.reset()
        self._builder.set_seats(5)
        self._builder.set_engine('FamilyEngine')
        self._builder.set_trip_computer(True)
```

```
        self._builder.set_gps(True)


# Construir un coche deportivo
builder = SedanCarBuilder()
director = CarDirector(builder)
director.construct_sports_car()
sports_car = builder.get_result()
print(sports_car)

# Construir directamente un coche familiar sin usar el director
builder.reset()
builder.set_seats(4)
builder.set_engine('FamilyEngine')
builder.set_trip_computer(True)
builder.set_gps(True)
family_car = builder.get_result()
print(family_car)
```

USOS CONOCIDOS
diseño Builder se utiliza ampliamente en la ingeniería de software para resolver varios problemas relacionados con la creación de objetos complejos. Algunos de los usos más conocidos como Construcción de Objetos Complejos, Inmutabilidad de Objetos, Creación de Documentos: entre otros
P. RELACIONADOS
Patrón Factory Method

G. Resumen

Se detallan tres patrones de diseño fundamentales: Facade, Builder y State. El patrón Facade simplifica la interacción con sistemas complejos, ofreciendo una interfaz sencilla y reduciendo la complejidad para los desarrolladores. El patrón Builder facilita la construcción de objetos complejos paso a paso, permitiendo variaciones en el proceso sin cambiar el código subyacente. Por último, el patrón State permite que un objeto modifique su comportamiento al cambiar su estado interno, lo que resulta en un código más limpio y fácil de mantener. Estos patrones mejoran la modularidad, mantenibilidad y flexibilidad del software, siendo esenciales para desarrollar aplicaciones eficientes y fáciles de gestionar.

H. Conclusión

Llegado a este punto, podemos concluir que los patrones de diseño Facade, Builder y State resultan ser soluciones extremadamente útiles a la hora de desarrollar software de calidad. Cada uno resuelve problemas comunes de una forma que simplifica el código y lo hace más modular, flexible y mantenible.

El patrón Facade nos permite ocultar la complejidad de subsistemas detrás de una interfaz más simple e intuitiva, lo cual hace que nuestros programas sean más fáciles de usar para el cliente final. A su vez, mejora el desacoplamiento entre los distintos componentes, facilitando cambios sin afectar la funcionalidad.

Por otra parte, Builder nos brinda una forma muy elegante de construir objetos de manera flexible y configurable, separando está tarea de la lógica y representación del objeto en sí. Esto nos permite construir familias completas de objetos relacionados de forma programática. Asimismo, elimina el problema de los telescoping constructors al no sobrecargar con parámetros las clases.

Finalmente, State encapsula el comportamiento relacionado a cada posible estado de un objeto en clases específicas. De esta forma, simplificamos el código al eliminar complejas condicionales if/else y switch/case, permitiendo que los objetos cambien su comportamiento dinámicamente mediante cambios de estado. A su vez, los nuevos estados son fáciles de agregar extendiendo la jerarquía existente.

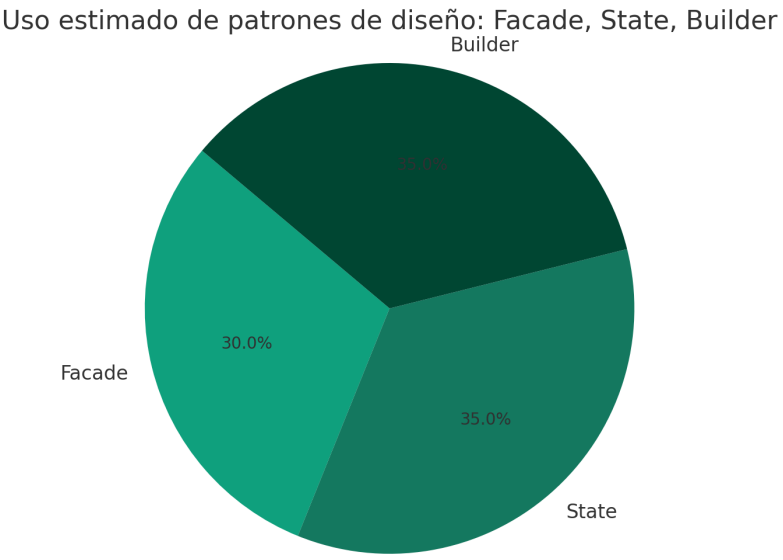
Link Presentación:

https://www.canva.com/design/DAF-xxyKH7Y/yxCIG1WYcQmmo2_EWJT9wg/edit?utm_content=DAF-xxyKH7Y&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

I. Anexos

1. **patrones de creación:**
   PATRONES DE DISEÑO EN JAVA. En esta serie vamos a hacer un repaso… | by Carlos Arturo Gonzalez | Medium

2. **Diagrama:**



J. Bibliografía

- Facade. (2024, January 1). Refactoring.Guru. https://refactoring.guru/design-patterns/facade

- State. (s.f). Refactoring.Guru. https://refactoring.guru/design-patterns/state