Análisis y diseño de aplicaciones I

UT5 – Patrones de diseño

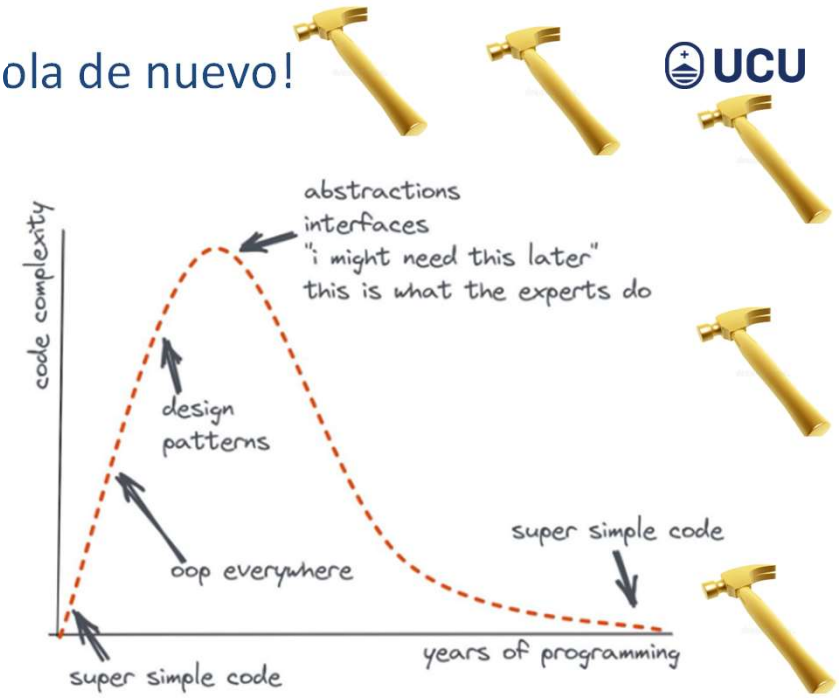
1

Agenda

- Chain of responsability
- Command
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

2

¡Hola de nuevo!



3

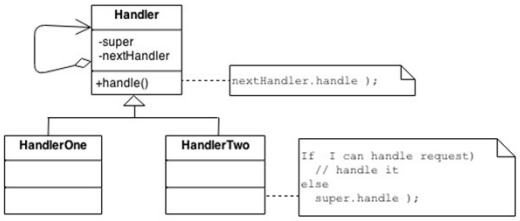
Chain of responsability

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

4

Chain of responsability

- Es importante garantizar que haya una "***red de seguridad***" que capture cualquier solicitud que no sea manejada por ninguno de los objetos en la cadena.
- El último eslabón de la cadena debe ser cuidadoso de no delegar a un "siguiente nulo".



- Las clases derivadas saben cómo satisfacer las solicitudes de los clientes. Si el **objeto "actual" no está disponible** o no es suficiente, entonces delega al objeto base, que a su vez **delega al "siguiente"** objeto, y así sucesivamente.
- Es posible que varios manejadores contribuyan al manejo de una sola solicitud, y la solicitud puede ser pasada por toda la longitud de la cadena.

5

Command

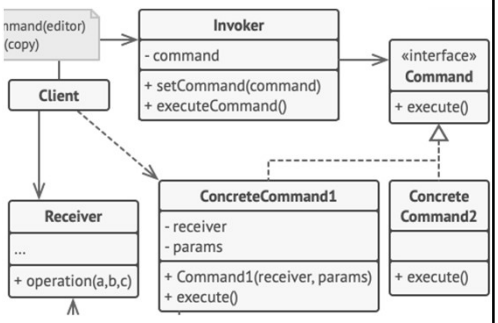
Transforma una solicitud en un objeto independiente con toda la información sobre la solicitud.

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

6

Command

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



- Las ventajas del Patrón Command incluyen un mayor desacoplamiento entre clases, la posibilidad de controlar diversas solicitudes fácilmente y una mayor flexibilidad en la ejecución de operaciones.
- Facilita la adición de nuevas operaciones sin modificar el código existente.
- Desventajas; puede llevar a un mayor número de clases y objetos, lo que aumenta la complejidad del código.
- El mantenimiento de un historial de solicitudes puede consumir bastante memoria, lo cual podría ser una desventaja en sistemas con recursos limitados.

7

Command

```
using System;

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

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

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

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

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

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

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

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

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

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

8

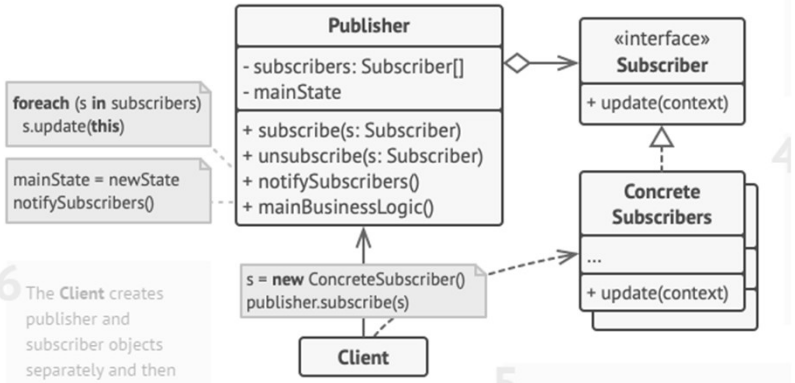
Command

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

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

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

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

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

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

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


Observer

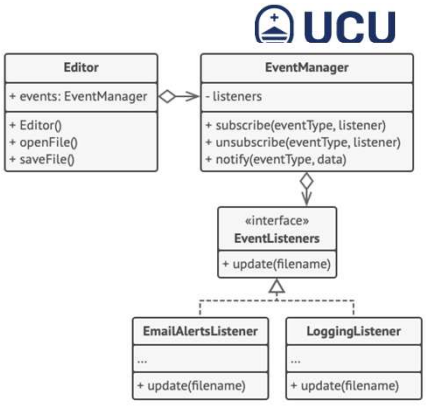
- También conocido como Publisher-Subscriber, define **una dependencia uno a muchos entre objetos**.
- Cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Esto es útil en situaciones donde un cambio en un objeto requiere cambios en otros objetos, sin saber cuántos objetos necesitan ser cambiados.
- El patrón Observer se compone de dos tipos de actores principales:
  1. **Sujeto (Subject)**: Es el objeto que **tiene la información o el estado que queremos observar**. Puede haber uno o varios observadores interesados en el estado del sujeto. El sujeto permite que los observadores se suscriban o se den de baja y es responsable de enviar notificaciones a los observadores cuando su estado cambie.
  2. **Observadores (Observers):** Son los objetos que necesitan **mantenerse informados** acerca de los cambios en el estado del sujeto. Implementan una interfaz que define el método de notificación que será invocado por el sujeto.

Observer



11

Observer



- **EventManager**: Esta es la clase base que maneja la lógica de subscripción. Tiene un hashmap listeners que mantiene una lista de suscriptores asociados con tipos de eventos. Además, tiene métodos para suscribirse (subscribe), cancelar la suscripción (unsubscribe) y notificar (notify) a los oyentes de ciertos eventos.
- **Editor**: Esta clase representa un editor de archivos, que actúa como el publicador en este patrón Observer. Contiene una instancia de EventManager para manejar la lógica de subscripción. Además, tiene métodos openFile y saveFile que realizan operaciones en un archivo y luego notifican a los suscriptores de los cambios.
- **EventListener**: Esta es una interfaz que define el contrato que deben seguir los observadores. En este caso, solo tienen que implementar un método update.
- ***LoggingListener* y *EmailAlertsListener***: Estas son clases concretas que implementan la interfaz EventListener. LoggingListener escribe en un archivo de log cuando se le notifica, mientras que EmailAlertsListener envía un correo electrónico.
- Cuando el **método openFile o saveFile** de la clase Editor es llamado, esto desencadena una notificación a todos los observadores que están suscritos a ese evento específico. Los observadores (en este caso, LoggingListener y EmailAlertsListener) entonces realizan sus acciones correspondientes (escribir en un archivo de log o enviar un correo electrónico).

12

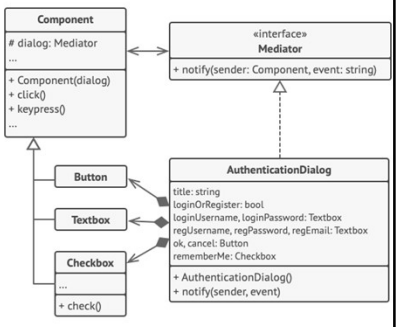
Mediator

- Se utiliza para reducir la comunicación compleja entre objetos estrechamente relacionados. En lugar de que los objetos se comuniquen directamente entre sí, estos objetos interactúan a través de un objeto mediador central.
- Esto es útil cuando tienes un conjunto de objetos que están estrechamente relacionados y que necesitan comunicarse entre sí de maneras complejas. En lugar de tener un número creciente de conexiones entre cada par de objetos, solo necesitas conectar cada objeto con el objeto mediador.
- Los componentes principales del patrón Mediator son:
  1. **Mediator** (Mediador): Es una interfaz que define cómo los objetos pueden comunicarse con el mediador.
  2. **ConcreteMediator** (Mediador Concreto): Es una clase que implementa la interfaz Mediator y coordina la comunicación entre objetos colegas. Mantiene referencias a los objetos colegas y puede tener lógica compleja para coordinar estos objetos.
  3. **Colleague** (Colega): Es una interfaz (o clase base) que define cómo los objetos colegas pueden comunicarse con el Mediador. A menudo, los objetos colegas tienen una referencia al mediador.
  4. **ConcreteColleague** (Colega Concreto): Son clases que implementan la interfaz Colleague. Son los objetos que necesitan comunicarse entre sí, pero en lugar de comunicarse directamente, utilizan el Mediador.

13

Mediator

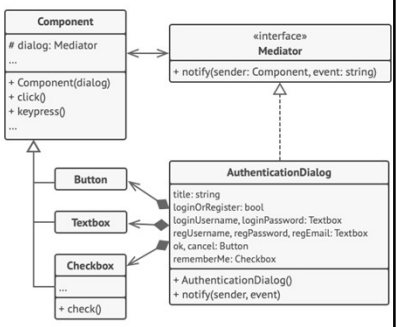
- **Mediator**: Es una interfaz que declara un método notify, que es usado por los componentes para notificar al mediador acerca de varios eventos.
- **AuthenticationDialog**: Esta es la clase concreta que implementa la interfaz Mediator. Es el Mediador en sí. Gestiona la lógica entre varios componentes como checkboxes, botones y cuadros de texto. Contiene referencias a estos componentes y define cómo deben interactuar entre sí.



14

Mediator

- El método ***notify*** es el más importante. Es llamado por los componentes cuando quieren notificar de un evento. El método contiene la lógica para manejar estos eventos. Por ejemplo, si el ***checkbox loginOrRegisterChkBx*** es marcado, el título cambia y muestra los componentes correspondientes para el formulario de inicio de sesión o registro.
- **Component**: Es una clase base para los componentes que interactúan con el Mediador. Tiene una referencia al Mediador y métodos como click y keypress que notifican al Mediador cuando son llamados.
- **Button, Textbox, Checkbox**: Son clases concretas que heredan de la clase Component. Representan componentes de la interfaz de usuario. Estos componentes no se comunican directamente entre sí, sino que comunican sus acciones al Mediador.



En resumen, Es utilizado para encapsular y simplificar la comunicación entre varios componentes en una interfaz de usuario.

Los componentes individuales no necesitan saber cómo interactúan entre sí. En su lugar, simplemente notifican al Mediador de sus acciones, y el Mediador contiene la lógica para gestionar cómo deben interactuar estos componentes.

15

Observer Vs Mediator

- Ambos tratan con la comunicación entre objetos, pero lo hacen de maneras distintas. Aquí hay una comparación entre ellos:
- **Propósito**:
  - Observer: Se utiliza para crear un sistema de notificación que permita a los objetos estar informados acerca de los cambios que ocurren en otros objetos. En esencia, permite que un objeto (el observador) reaccione a los cambios que ocurren en otro objeto (el sujeto).
  - Mediator: Facilita la comunicación entre un conjunto de objetos, actuando como un intermediario, de modo que los objetos no se comuniquen directamente entre sí. Esto ayuda a reducir las dependencias entre los objetos, lo que puede hacer que el sistema sea más fácil de mantener y entender.
- **Relación**:
  - Observer: Define una relación de "uno a muchos" entre objetos, de modo que cuando un objeto cambia de estado, todos los objetos dependientes son notificados y actualizados automáticamente.
  - Mediator: Define relaciones entre pares de objetos, permitiendo que se comuniquen entre sí sin tener referencias directas. El mediador centraliza la comunicación.

16

Observer Vs Mediator

- **Acoplamiento**:
  - Observer: Reduce el acoplamiento entre el sujeto y los observadores, porque el sujeto no necesita saber nada acerca de quiénes son los observadores o qué hacen. Solo mantiene una lista de observadores que deben ser notificados cuando cambia su estado.
  - Mediator: Reduce el acoplamiento entre un conjunto de objetos al evitar que se comuniquen directamente entre sí. En lugar de eso, los objetos solo se comunican a través del mediador.
- **Complejidad y responsabilidad:**
  - Observer: Tiende a ser menos complejo en términos de la lógica de interacción, ya que se trata principalmente de notificar a los observadores sobre cambios en el sujeto.
  - Mediator: Puede volverse más complejo, especialmente si la lógica de interacción entre los objetos es complicada, ya que el mediador centraliza esta lógica de interacción.

17

Observer Vs Mediator

- **Uso común:**
  - Observer: Es comúnmente usado en escenarios donde un cambio en el estado de un objeto debe reflejarse automáticamente en otros objetos sin que el objeto cambiado conozca qué y cómo se afectarán los otros objetos (por ejemplo, modelos de datos y vistas en una aplicación GUI).
  - Mediator: Es útil cuando un conjunto de objetos necesita interactuar de maneras complejas, pero quieres evitar un esquema de comunicación altamente acoplado y complicado entre ellos (por ejemplo, en un chat donde varios usuarios interactúan entre sí).
- En **resumen**, el patrón Observer es más adecuado cuando **solo necesitas mantener informados a varios objetos** sobre los cambios en un objeto, mientras que el patrón Mediator es mejor cuando ***necesitas facilitar una comunicación más compleja*** entre un conjunto de objetos.

18

Memento

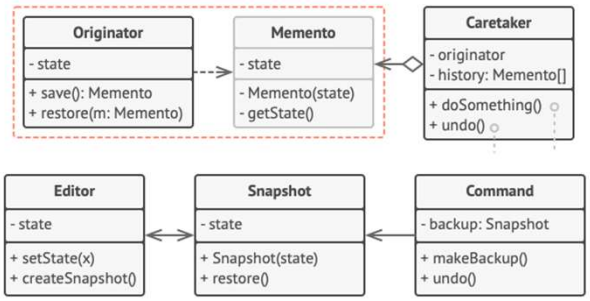
- Se utiliza para capturar el estado interno de un objeto en un punto en el tiempo, de modo que el objeto pueda ser restaurado a ese estado más tarde. Esto es útil en características como los sistemas de deshacer/rehacer en editores de texto, o para tomar instantáneas del estado de un sistema.

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

19

Memento

- **Editor**: Esta es la clase Originator en el patrón Memento. Contiene el estado que queremos ser capaces de deshacer.
- ***createSnapshot***, crea un objeto Memento (Snapshot) que contiene una copia del estado actual del editor.
- **Snapshot**: Esta es la clase Memento en el patrón. Contiene una copia del estado del editor en un momento dado.
- Los atributos en Snapshot son ***privados***, lo que significa que su estado no puede ser alterado una vez que se crea el objeto.
- El constructor toma el estado del editor y lo almacena en los campos de la clase.



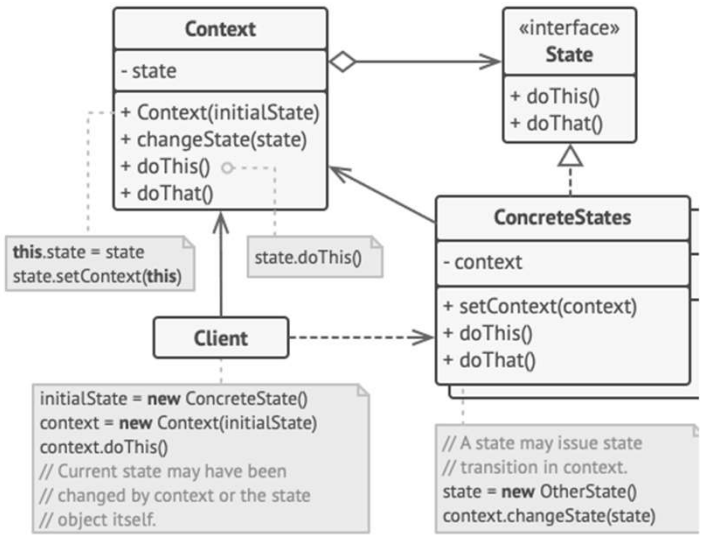
- ***restore***, se utiliza para restaurar el estado del editor al estado almacenado en el Memento.
- **Command**: Actúa como el Caretaker en el patrón Memento.
- ***makeBackup*** crea un objeto Memento del estado actual del editor antes de realizar cambios en su estado.
- **undo** utiliza el Memento almacenado para restaurar el estado anterior del editor.
- En un escenario de uso típico, cada vez que se va a realizar un cambio en el editor, primero se crearía un Memento del estado actual utilizando makeBackup. Si luego el usuario decide deshacer la última acción, se podría llamar al método undo para restaurar el estado del editor al estado almacenado en el último Memento.

20

State

- Permite a un objeto cambiar su comportamiento cuando su estado interno cambia.
- Esto puede ser útil en situaciones donde un objeto debe cambiar su comportamiento de manera dinámica en tiempo de ejecución, en función de ciertas condiciones.
- En otras palabras, el patrón State sugiere que se cree una ***nueva clase para cada estado*** posible de un objeto, y que se extraiga el comportamiento específico de ese estado a esa clase.
- Componentes del patrón State:
  1. **Contexto**: Es la clase que tiene un estado. Contiene una referencia a una instancia de uno de los estados concretos y delega a ella el comportamiento que depende del estado.
  2. **State**: Es una interfaz que define una interfaz común para todos los estados concretos. Define los métodos que deben implementar las clases de estado concreto.
  3. **Concrete States**: Son las clases que implementan la interfaz State y definen el comportamiento asociado con un estado particular del Contexto.

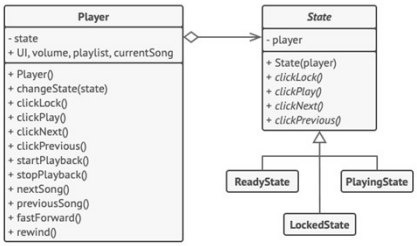
21



22

State

- **Player** es la clase Contexto en el patrón State. Contiene el estado actual del reproductor de audio.
- El método changeState que permite cambiar el estado actual del reproductor.
- Los métodos clickLock, clickPlay, clickNext, y clickPrevious son delegados al estado actual.
- **State** el estado general del reproductor de audio.
- Contiene una referencia protegida a Player, lo que permite que las subclases de State interactúen con el contexto.



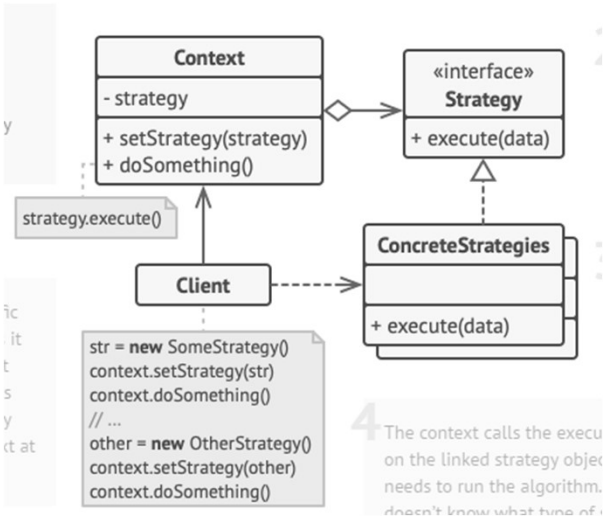
- ***Concrete States*** (LockedState, ReadyState, PlayingState classes): Estas clases representan estados específicos del reproductor de audio.
- En resumen, este código **representa un reproductor de audio que tiene diferentes comportamientos dependiendo de su estado actual**. Los comportamientos están encapsulados en clases de estado concreto, y el reproductor de audio delega acciones a la clase de estado actual.

23

Strategy

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

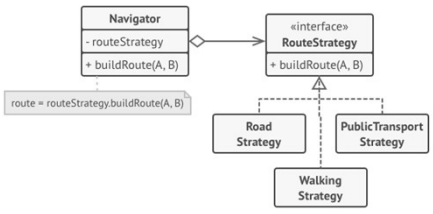
24



25

Strategy

- **RoutingStrategy** es la interfaz que declara el método BuildRoute que todas las estrategias concretas deben implementar.
- **Navigator** es la clase contexto. Mantiene una referencia a una estrategia (_routingStrategy) y permite que los clientes establezcan la estrategia en tiempo de ejecución.



- **RoadStrategy, PublicTransportStrategy y WalkingStrategy** son clases que implementan RoutingStrategy. Cada una de estas clases encapsula la lógica de cómo construir una ruta basada en diferentes medios de transporte.
- En suma:
  - Permite que el navegador GPS calcule rutas utilizando diferentes medios de transporte (carretera, transporte público, a pie) sin tener que modificar la clase del navegador GPS en sí.
  - Facilita la adición de nuevas estrategias de enrutamiento en el futuro sin cambiar el código existente.
  - Proporciona una forma de seleccionar dinámicamente un algoritmo de enrutamiento en tiempo de ejecución.
  - En términos más generales, el patrón Strategy se utiliza para desacoplar el comportamiento específico de la lógica de una clase, haciendo que el código sea más modular, mantenible y adaptable a cambios.

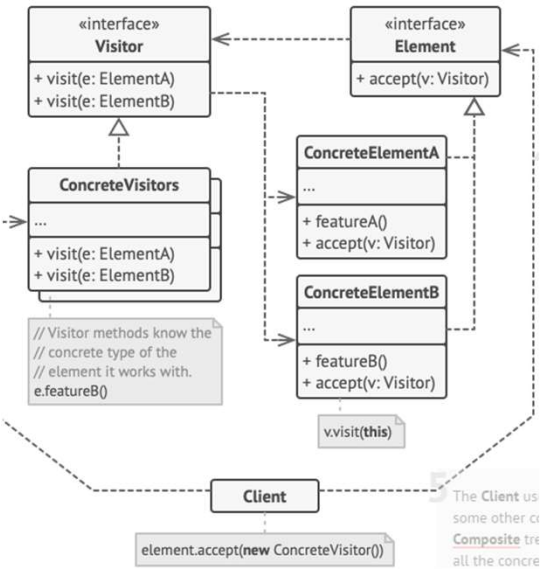
26

Visitor

- Permite separar algoritmos de los objetos sobre los que operan. Esto puede ser útil cuando necesitan realizar operaciones sobre estos objetos sin alterar sus clases.
- Los componentes principales del patrón Visitor son:
  1. **Visitor**: Es una interfaz que declara un conjunto de métodos de visita, uno para cada tipo concreto de elemento en la estructura de objetos. Cada método de visita acepta un único argumento, que es uno de los tipos de elementos de la estructura.
  2. **ConcreteVisitor**: Estas son las clases que implementan la interfaz Visitor. Implementan cada uno de los métodos de visita definidos en la interfaz Visitor.
  3. **Element**: Es una interfaz que declara un método accept que acepta un objeto de tipo Visitor como argumento.
  4. **ConcreteElement**: Son las clases que implementan la interfaz Element. Implementan el método accept y, por lo general, tienen lógica adicional relacionada con el elemento.
  5. **Object Structure**: Es una estructura de objetos que puede contener varios elementos ConcreteElement. Permite que los visitantes visiten los elementos.

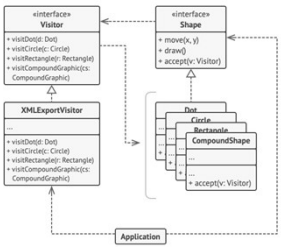
27

Visitor



28

Visitor



- **Shape** (Elemento Interface): declara los métodos que deben ser implementados por los elementos concretos.
- **Dot, Circle, Rectangle, CompoundShape** (Elementos Concretos): Cada una de estas clases debe implementar el método accept de tal manera que llame al método del Visitor que corresponde a su clase. Esto se logra pasando this como argumento al método del visitor (por ejemplo, v.visitDot(this) en la clase Dot).
- **Visitor** (Interfaz de Visitante): Declara un conjunto de métodos de visita, uno para cada tipo concreto de elemento.
- **XMLExportVisitor** (Visitante Concreto): Define la lógica para exportar cada tipo de forma a un formato XML. Por ejemplo, para un círculo, podría exportar el ID, las coordenadas del centro y el radio.
- **Application** (Cliente): Es la parte del código que quiere realizar operaciones sobre los objetos de las formas sin conocer sus clases concretas. En este ejemplo, la clase Application quiere exportar todas las formas a XML***. Crea un XMLExportVisitor*** y recorre todas las formas llamando al método accept en cada forma con el visitante XML como argumento. *Esto redirige la llamada al método apropiado en el objeto visitante.*
- **En resumen**, este patrón permite que las clases de formas geométricas sean extendidas con nuevas operaciones (en este caso, exportación a XML) sin modificar las clases en sí. En lugar de eso***, se encapsula la lógica de exportación*** en una clase de visitante separada ***y se utiliza el método accept para delegar la operación al objeto visitante apropiado***. Esto hace que el código sea más flexible y facilita la adición de nuevas operaciones sin cambiar las clases de los elementos.

29

State, Strategy y Visitor

- ***Intención***
  - **State**: Cambiar el comportamiento de un objeto basado en su estado interno.
  - **Strategy**: Permitir que un objeto tenga varios algoritmos o estrategias intercambiables.
  - **Visitor**: Agregar nuevas operaciones a clases sin modificarlas.
- ***Estructura***
  - **State**: Contexto que mantiene referencia a un estado concreto.
    - Interfaz de estado. Estados concretos que implementan la interfaz de estado.
  - **Strategy**: Contexto que mantiene referencia a una estrategia concreta.
    - Interfaz de estrategia. Estrategias concretas que implementan la interfaz de estrategia..
  - **Visitor**: Elementos que tienen un método accept para recibir visitantes.
    - Interfaz de visitante. Visitantes concretos que implementan la interfaz de visitante.

30

State, Strategy y Visitor

- ***Uso común***
  - State: Cuando el comportamiento de un objeto debe cambiar dinámicamente en función de su estado.
  - Strategy: Cuando se necesita elegir entre varias implementaciones de un algoritmo o comportamiento en tiempo de ejecución.
  - Visitor: Para realizar operaciones sobre una estructura de objetos heterogéneos sin tener que modificar sus clases. Útil para mantener la lógica de operación separada de la estructura de objetos.

31

Tarea de Aplicación 5
PATRONES DE COMPORTAMIENTO.

32

Bibliografía

- https://refactoring.guru/design-patterns
- https://sourcemaking.com/design_patterns
- Design Patterns (1994) Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside

33

"Cuando algo es lo suficientemente importante, lo haces incluso si las probabilidades de que salga bien no te acompañan - Elon Musk"

34