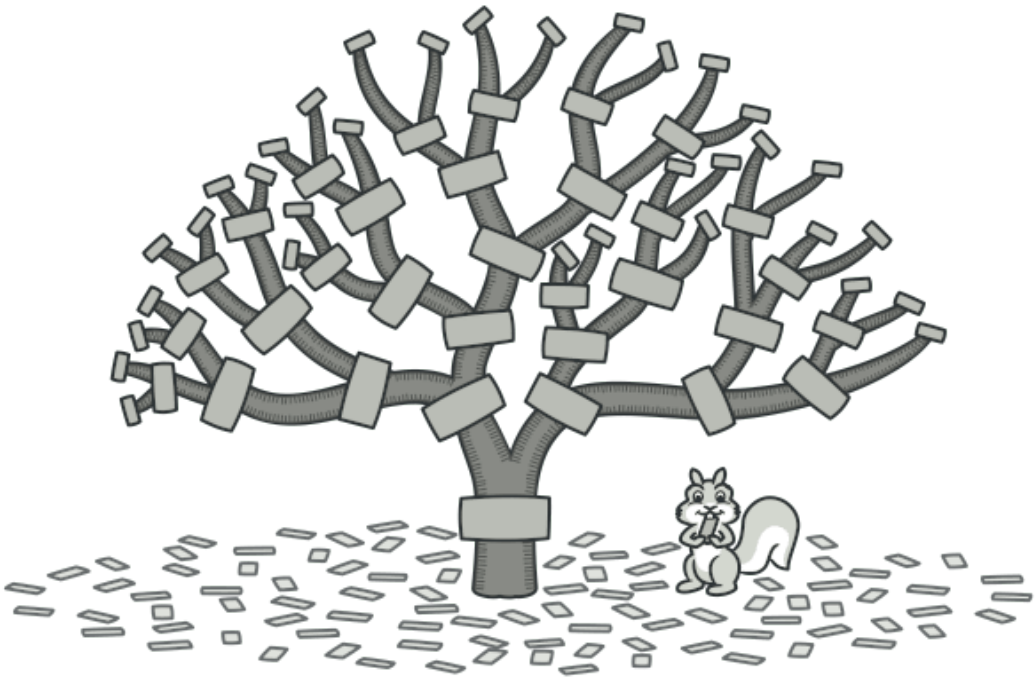
Composite

También llamado: Objeto compuesto, Object Tree

Propósito

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

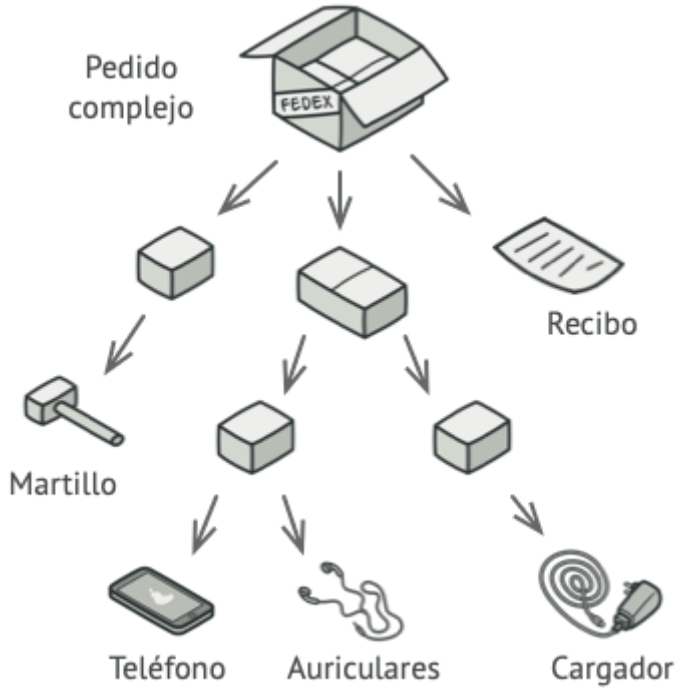


Problema

El uso del patrón Composite sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.

Por ejemplo, imagina que tienes dos tipos de objetos: `Productos` y `Cajas`.
Una `Caja` puede contener varios `Productos` así como cierto número de `Cajas` más pequeñas. Estas `Cajas` pequeñas también pueden contener algunos `Productos` o incluso `Cajas` más pequeñas, y así sucesivamente.

Digamos que decides crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos... y otras cajas. ¿Cómo determinarás el precio total de ese pedido?



Un pedido puede incluir varios productos empaquetados en cajas, que a su vez están empaquetados en cajas más grandes y así sucesivamente. La estructura se asemeja a un árbol boca abajo.

Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de `Productos` y `Cajas` a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

Solución

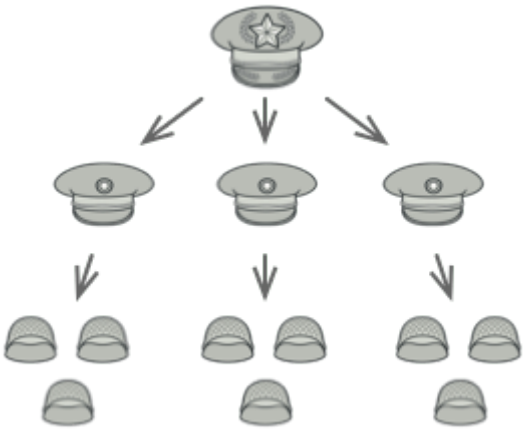
El patrón Composite sugiere que trabajes con `Productos` y `Cajas` a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

El patrón Composite te permite ejecutar un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos.

La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol. No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol.

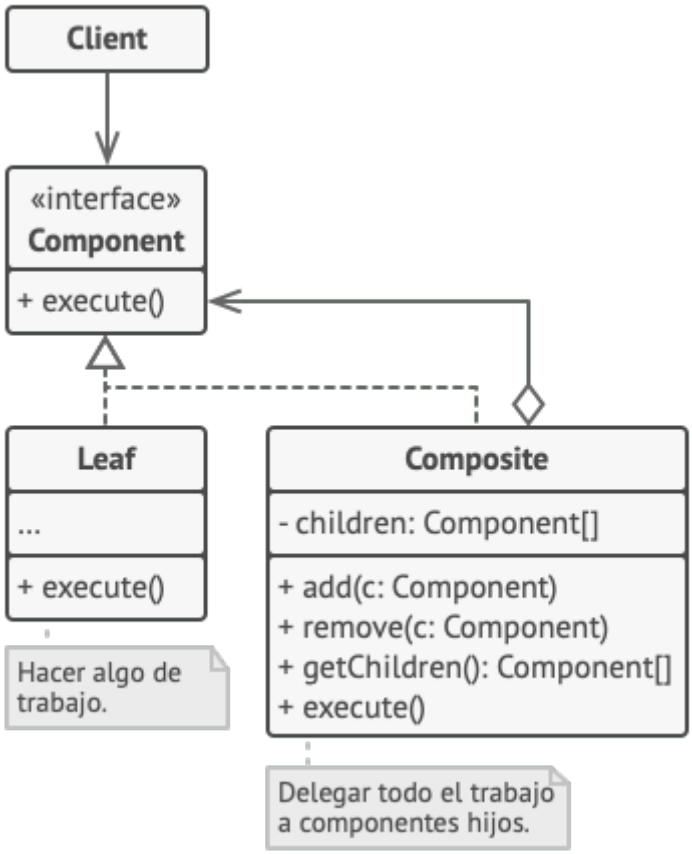
Analogía en el mundo real



Un ejemplo de estructura militar.

Los ejércitos de la mayoría de países se estructuran como jerarquías. Un ejército está formado por varias divisiones; una división es un grupo de brigadas y una brigada está formada por pelotones, que pueden dividirse en escuadrones. Por último, un escuadrón es un pequeño grupo de soldados reales. Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo por cada nivel hasta que todos los soldados saben lo que hay que hacer.

Estructura



1. La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.
2. La **Hoja** es un elemento básico de un árbol que no tiene subelementos.

   Normalmente, los componentes de la hoja acaban realizando la mayoría del trabajo real, ya que no tienen a nadie a quien delegarle el trabajo.

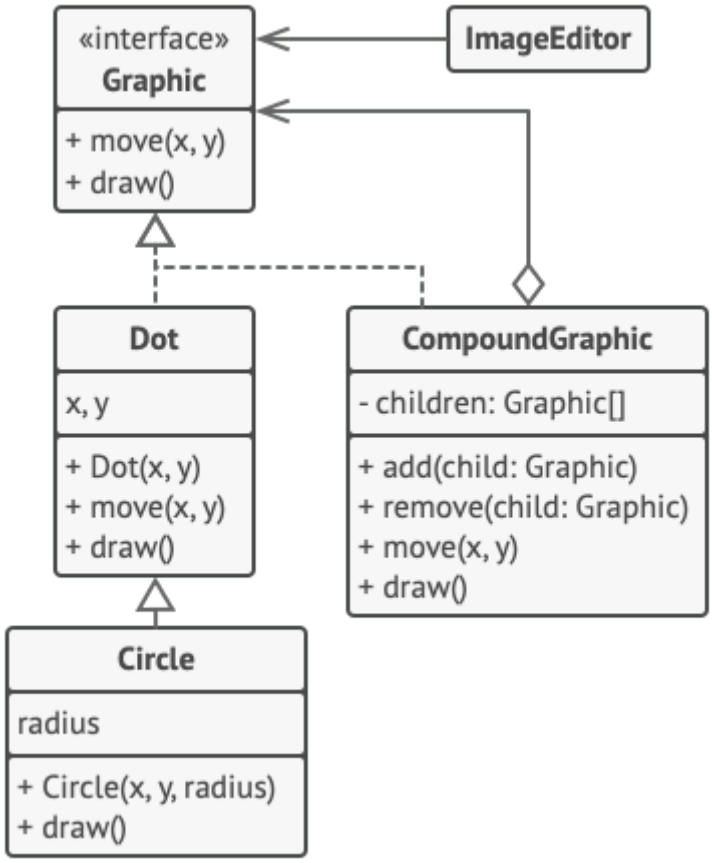
3. El **Contenedor** (también llamado *compuesto*) es un elemento que tiene subelementos: hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente.

   Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

4. El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.

Pseudocódigo

En este ejemplo, el patrón **Composite** te permite implementar el apilamiento (*stacking*) de formas geométricas en un editor gráfico.



Ejemplo del editor de formas geométricas.

La clase `GráficoCompuesto` es un contenedor que puede incluir cualquier cantidad de subformas, incluyendo otras formas compuestas. Una forma compuesta tiene los mismos métodos que una forma simple. Sin embargo, en lugar de hacer algo por su cuenta, una forma compuesta pasa la solicitud de forma recursiva a todos sus hijos y “suma” el resultado.

El código cliente trabaja con todas las formas a través de la interfaz común a todas las clases de forma. De este modo, el cliente no sabe si está trabajando con una forma simple o una compuesta. El cliente puede trabajar con estructuras de objetos muy complejas sin acoplarse a las clases concretas que forman esa estructura.

```
// La interfaz componente declara operaciones comunes para
// objetos simples y complejos de una composición.
interface Graphic is
    method move(x, y)
    method draw()

// La clase hoja representa objetos finales de una composición.
// Un objeto hoja no puede tener ningún subobjeto. Normalmente,
// son los objetos hoja los que hacen el trabajo real, mientras
// que los objetos compuestos se limitan a delegar a sus
// subcomponentes.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }
```

```
    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Dibuja un punto en X e Y.

// Todas las clases de componente pueden extender otros
// componentes.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Dibuja un círculo en X y Y con radio R.

// La clase compuesta representa componentes complejos que
// pueden tener hijos. Normalmente los objetos compuestos
// delegan el trabajo real a sus hijos y después "recapitulan"
// el resultado.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // Un objeto compuesto puede añadir o eliminar otros
    // componentes (tanto simples como complejos) a o desde su
    // lista hija.
    method add(child: Graphic) is
        // Añade un hijo a la matriz de hijos.

    method remove(child: Graphic) is
        // Elimina un hijo de la matriz de hijos.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // Un compuesto ejecuta su lógica primaria de una forma
    // particular. Recorre recursivamente todos sus hijos,
    // recopilando y recapitulando sus resultados. Debido a que
    // los hijos del compuesto pasan esas llamadas a sus propios
    // hijos y así sucesivamente, se recorre todo el árbol de
    // objetos como resultado.
    method draw() is
        // 1. Para cada componente hijo:
        //     - Dibuja el componente.
        //     - Actualiza el rectángulo delimitador.
        // 2. Dibuja un rectángulo de línea punteada utilizando
        // las coordenadas de delimitación.


// El código cliente trabaja con todos los componentes a través
// de su interfaz base. De esta forma el código cliente puede
// soportar componentes de hoja simples así como compuestos
// complejos.
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
```

```
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Combina componentes seleccionados para formar un
    // componente compuesto complejo.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // Se dibujarán todos los componentes.
        all.draw()
```

EJEMPLO DE LA VIDA REAL:

1. Monitoreo de Redes y trabajo a través de NetWorking o CoWorking
2. Consolidación de Portafolio de Contabilidad
3. Consolidar todo un portafolio bancario en interfaz
4. Estructura de Directorios en sistemas de archivos
5. Manejo de inventarios en almacenes

BALANCE TOTAL DE CUENTAS DEL CLIENTE

ESTADO DE CUENTA

Codigo:

Component.java

```java
import java.util.ArrayList;
import java.util.List;

public abstract class Component
{
    AccountStatement accStatement;

    protected List<Component> list = new ArrayList<>();

    public abstract float getBalance();

    public abstract AccountStatement getStatement();

    public void add(Component g) {
        list.add(g);
    }

    public void remove(Component g) {
        list.remove(g);
    }

    public Component getChild(int i) {
        return (Component) list.get(i);
    }
}
```

CompositeAccount.java

```
public class CompositeAccount extends Component
{
    private float totalBalance;
    private AccountStatement compositeStmt, individualStmt;

    public float getBalance() {
        totalBalance = 0;
        for (Component f : list) {
            totalBalance = totalBalance + f.getBalance();
        }
        return totalBalance;
    }

    public AccountStatement getStatement() {
        for (Component f : list) {
            individualStmt = f.getStatement();
            compositeStmt.merge(individualStmt);
        }
        return compositeStmt;
    }
}
```

AccountStatement.java

```
public class AccountStatement
{
    public void merge(AccountStatement g)
    {
        //Use this function to merge all account statements
    }
}
```

DepositAccount.java

```
public class DepositAccount extends Component
{
    private String accountNo;
    private float accountBalance;

    private AccountStatement currentStmt;

    public DepositAccount(String accountNo, float accountBalance) {
        super();
        this.accountNo = accountNo;
        this.accountBalance = accountBalance;
    }

    public String getAccountNo() {
        return accountNo;
    }

    public float getBalance() {
        return accountBalance;
    }

    public AccountStatement getStatement() {
        return currentStmt;
    }
}
```

SavingsAccount.java

```
public class SavingsAccount extends Component
{
    private String accountNo;
    private float accountBalance;

    private AccountStatement currentStmt;

    public SavingsAccount(String accountNo, float accountBalance) {
        super();
        this.accountNo = accountNo;
        this.accountBalance = accountBalance;
    }

    public String getAccountNo() {
        return accountNo;
    }

    public float getBalance() {
        return accountBalance;
    }

    public AccountStatement getStatement() {
        return currentStmt;
    }
}
```

Client.java

```
public class Client
{
    public static void main(String[] args)
    {
        // Creating a component tree
        Component component = new CompositeAccount();

        // Adding all accounts of a customer to component
        component.add(new DepositAccount("DA001", 100));
        component.add(new DepositAccount("DA002", 150));
        component.add(new SavingsAccount("SA001", 200));

        // getting composite balance for the customer
        float totalBalance = component.getBalance();
        System.out.println("Total Balance : " + totalBalance);

        AccountStatement mergedStatement = component.getStatement();
        //System.out.println("Merged Statement : " + mergedStatement);
    }
}

Output:

Total Balance : 450.0
```

Memento

También llamado: Recuerdo, Instantánea, Snapshot

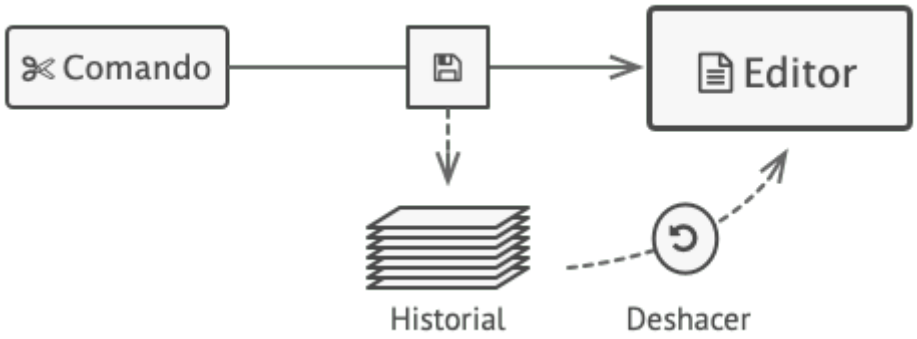
Propósito

Memento es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Problema

Imagina que estás creando una aplicación de edición de texto. Además de editar texto, tu programa puede formatearlo, asi como insertar imágenes en línea, etc.

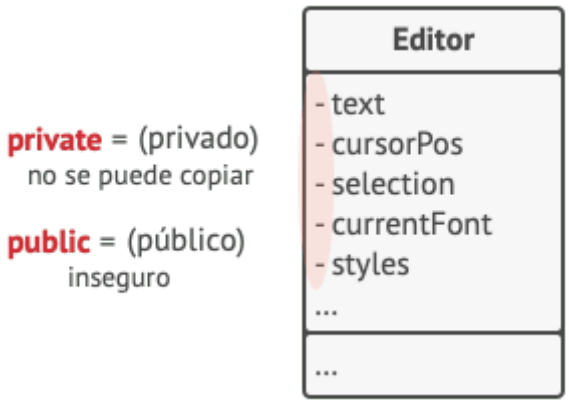
En cierto momento, decides permitir a los usuarios deshacer cualquier operación realizada en el texto. Esta función se ha vuelto tan habitual en los últimos años que hoy en día todo el mundo espera que todas las aplicaciones la tengan. Para la implementación eliges la solución directa. Antes de realizar cualquier operación, la aplicación registra el estado de todos los objetos y lo guarda en un almacenamiento. Más tarde, cuando un usuario decide revertir una acción, la aplicación extrae la última *instantánea* del historial y la utiliza para restaurar el estado de todos los objetos.



Antes de ejecutar una operación, la aplicación guarda una instantánea del estado de los objetos, que más tarde se puede utilizar para restaurar objetos a su estado previo.

Pensemos en estas instantáneas de estado. ¿Cómo producirías una, exactamente? Probablemente tengas que recorrer todos los campos de un objeto y copiar sus valores en el almacenamiento. Sin embargo, esto sólo funcionará si el objeto tiene unas restricciones bastante laxas al acceso a sus contenidos. Lamentablemente, la mayoría de objetos reales no permite a otros asomarse a su interior fácilmente, y esconden todos los datos significativos en campos privados.

Ignora ese problema por ahora y asumamos que nuestros objetos se comportan como hippies: prefieren relaciones abiertas y mantienen su estado público. Aunque esta solución resolvería el problema inmediato y te permitiría producir instantáneas de estados de objetos a voluntad, sigue teniendo algunos inconvenientes serios. En el futuro, puede que decidas refactorizar algunas de las clases editoras, o añadir o eliminar algunos de los campos. Parece fácil, pero esto también exige cambiar las clases responsables de copiar el estado de los objetos afectados.



¿Cómo hacer una copia del estado privado del objeto?

Pero aún hay más. Pensemos en las propias “instantáneas” del estado del editor. ¿Qué datos contienen? Como mínimo, deben contener el texto, las coordenadas del cursor, la posición actual de desplazamiento, etc. Para realizar una instantánea debes recopilar estos valores y meterlos en algún tipo de contenedor.

Probablemente almacenarás muchos de estos objetos de contenedor dentro de una lista que represente el historial. Por lo tanto, probablemente los contenedores acaben siendo objetos de una clase. La clase no tendrá apenas métodos, pero sí muchos campos que reflejen el estado del editor. Para permitir que otros objetos escriban y lean datos a y desde una instantánea, es probable que tengas que hacer sus campos públicos. Esto expondrá todos los estados del editor, privados o no. Otras clases se volverán dependientes de cada pequeño cambio en la clase de la instantánea, que de otra forma ocurriría dentro de campos y métodos privados sin afectar a clases externas.

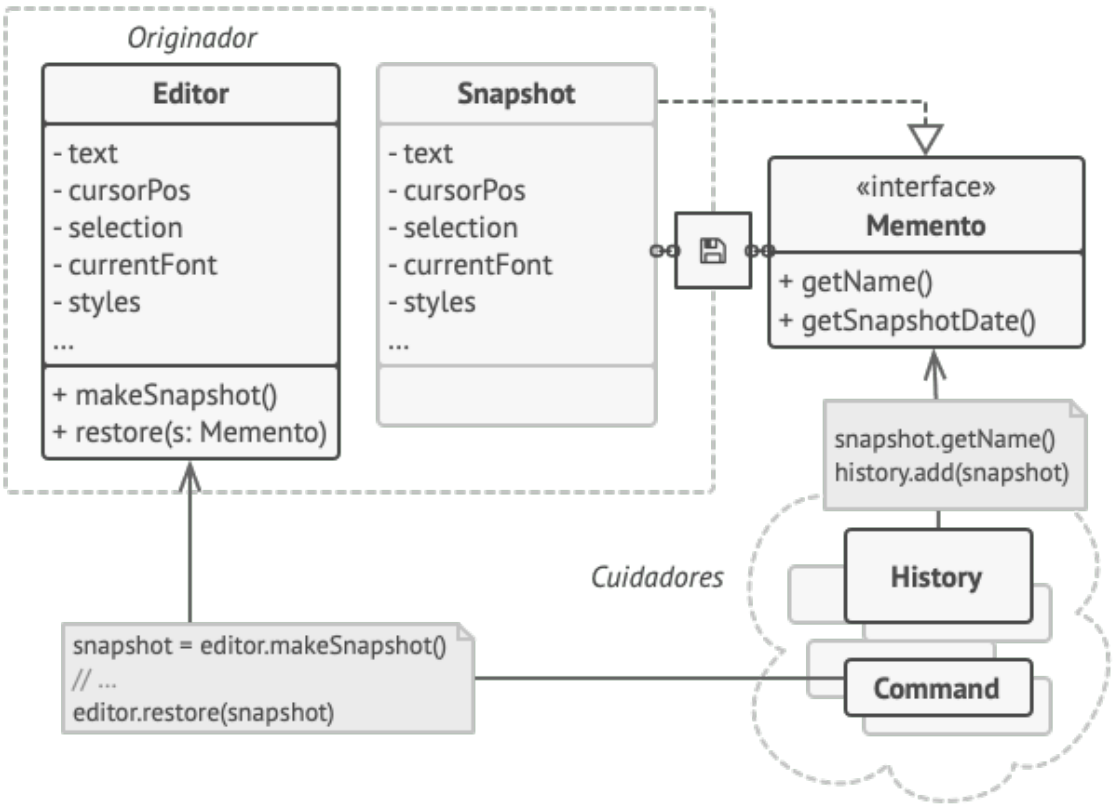
Parece que hemos llegado a un callejón sin salida: o bien expones todos los detalles internos de las clases, haciéndolas demasiado frágiles, o restringes el acceso a su estado, haciendo imposible producir instantáneas. ¿Hay alguna otra forma de implementar el "deshacer"?

Solución

Todos los problemas que hemos experimentado han sido provocados por una encapsulación fragmentada. Algunos objetos intentan hacer más de lo que deben. Para recopilar los datos necesarios para realizar una acción, invaden el espacio privado de otros objetos en lugar de permitir a esos objetos realizar la propia acción.

El patrón Memento delega la creación de instantáneas de estado al propietario de ese estado, el objeto *originador*. Por lo tanto, en lugar de que haya otros objetos intentando copiar el estado del editor desde el “exterior”, la propia clase editora puede hacer la instantánea, ya que tiene pleno acceso a su propio estado.

El patrón sugiere almacenar la copia del estado del objeto en un objeto especial llamado *memento*. Los contenidos del memento no son accesibles para ningún otro objeto excepto el que lo produjo. Otros objetos deben comunicarse con mementos utilizando una interfaz limitada que pueda permitir extraer los metadatos de la instantánea (tiempo de creación, el nombre de la operación realizada, etc.), pero no el estado del objeto original contenido en la instantánea.



El originador tiene pleno acceso al memento, mientras que el cuidador sólo puede acceder a los metadatos.

Una política tan restrictiva te permite almacenar mementos dentro de otros objetos, normalmente llamados *cuidadores*. Debido a que el cuidador trabaja con el memento únicamente a través de la interfaz limitada, no puede manipular el estado almacenado dentro del memento. Al mismo tiempo, el originador tiene acceso a todos los campos dentro del memento, permitiéndole restaurar su estado previo a voluntad.

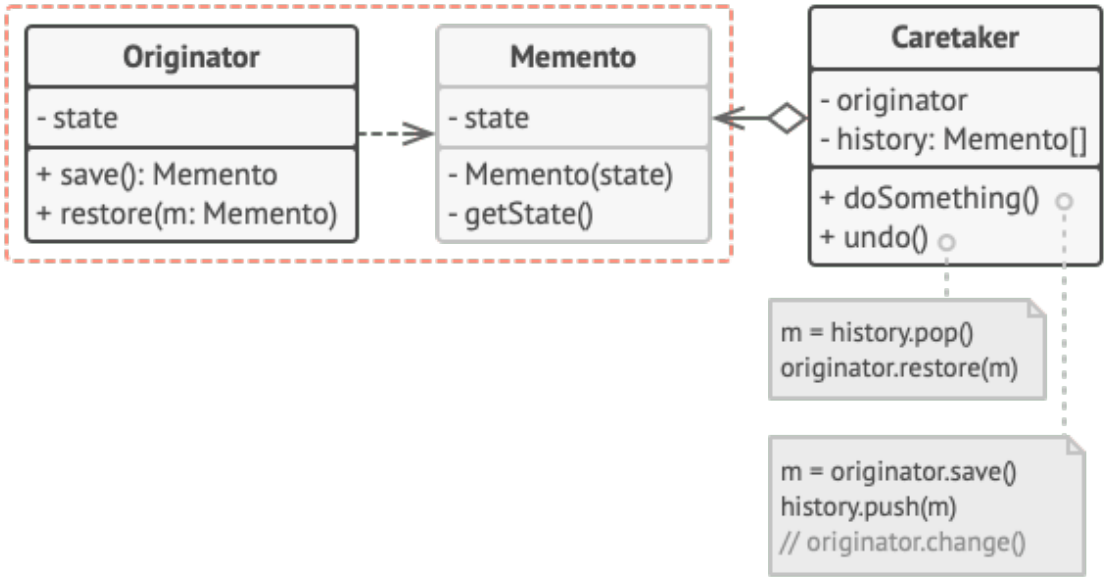
En nuestro ejemplo del editor de texto, podemos crear una clase separada de historial que actúe como cuidadora. Una pila de mementos almacenados dentro de la cuidadora crecerá cada vez que el editor vaya a ejecutar una operación. Puedes incluso presentar esta pila dentro de la UI de la aplicación, mostrando a un usuario el historial de operaciones previamente realizadas.

Cuando un usuario activa la función Deshacer, el historial toma el memento más reciente de la pila y lo pasa de vuelta al editor, solicitando una restauración. Debido a que el editor tiene pleno acceso al memento, cambia su propio estado con los valores tomados del memento.

Estructura

Implementación basada en clases anidadas

La implementación clásica del patrón se basa en el soporte de clases anidadas, disponible en varios lenguajes de programación populares (como C++, C# y Java).



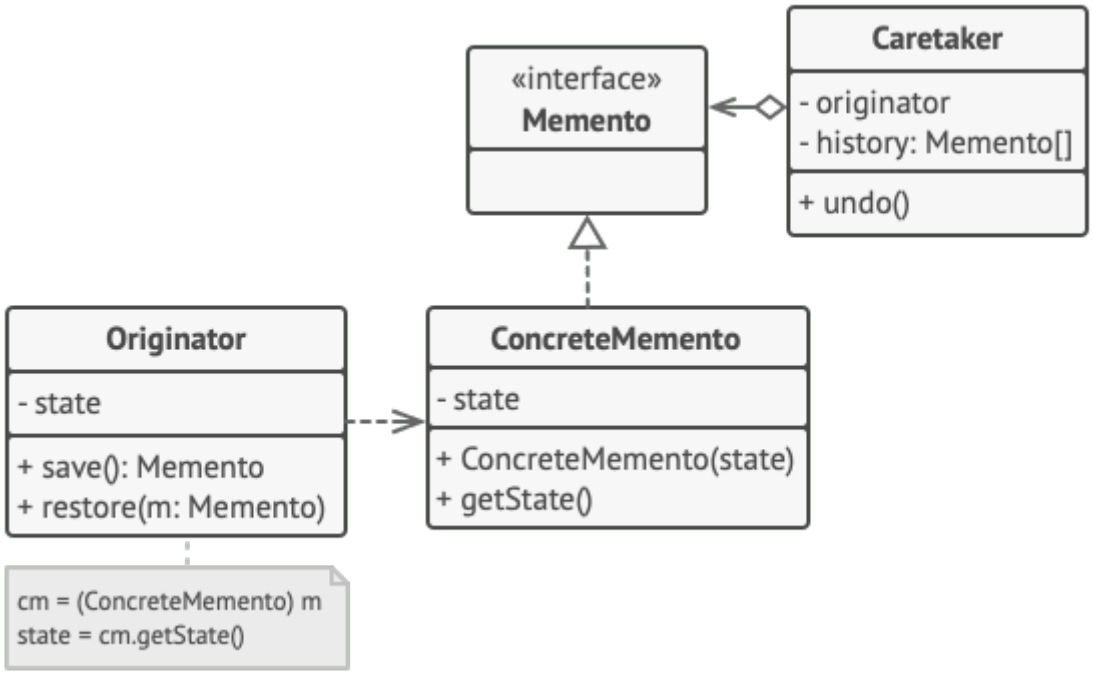
1. La clase **Originadora** puede producir instantáneas de su propio estado, así como restaurar su estado a partir de instantáneas cuando lo necesita.
2. El **Memento** es un objeto de valor que actúa como instantánea del estado del originador. Es práctica común hacer el memento inmutable y pasarle los datos solo una vez, a través del constructor.
3. La **Cuidadora** sabe no solo "cuándo" y "por qué" capturar el estado de la originadora, sino también cuándo debe restaurarse el estado.

   Una cuidadora puede rastrear el historial de la originadora almacenando una pila de mementos. Cuando la originadora deba retroceder en el historial, la cuidadora extraerá el memento de más arriba de la pila y lo pasará al método de restauración de la originadora.

4. En esta implementación, la clase memento se anida dentro de la originadora. Esto permite a la originadora acceder a los campos y métodos de la clase memento, aunque se declaren privados. Por otro lado, la cuidadora tiene un acceso muy limitado a los campos y métodos de la clase memento, lo que le permite almacenar mementos en una pila pero no alterar su estado.

   Implementación basada en una interfaz intermedia

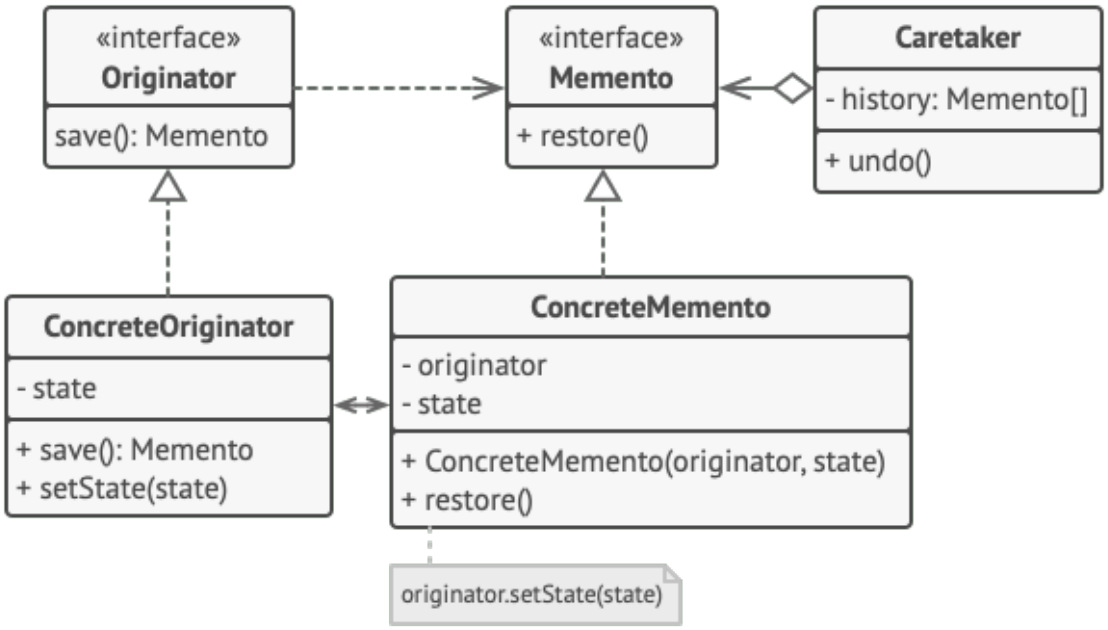
   Existe una implementación alternativa, adecuada para lenguajes de programación que no soportan clases anidadas (sí, PHP, estoy hablando de ti).



1. En ausencia de clases anidadas, puedes restringir el acceso a los campos de la clase memento estableciendo una convención de que las cuidadoras sólo pueden trabajar con una memento a través de una interfaz intermediaria explícitamente declarada, que sólo declarará métodos relacionados con los metadatos del memento.
2. Por otro lado, las originadoras pueden trabajar directamente con un objeto memento, accediendo a campos y métodos declarados en la clase memento. El inconveniente de esta solución es que debes declarar públicos todos los miembros de la clase memento.

Implementación con una encapsulación más estricta

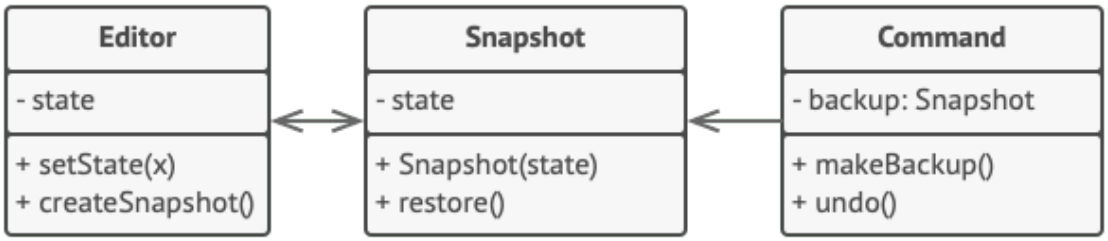
Existe otra implementación que resulta útil cuando no queremos dejar la más mínima opción a que otras clases accedan al estado de la originadora a través del memento.



1. Esta implementación permite tener varios tipos de originadoras y mementos. Cada originadora trabaja con una clase memento correspondiente. Ninguna de las dos expone su estado a nadie.
2. Las cuidadoras tienen ahora explícitamente restringido cambiar el estado almacenado en los mementos. Además, la clase cuidadora se vuelve independiente de la originadora porque el método de restauración se define ahora en la clase memento.
3. Cada memento queda vinculado a la originadora que lo produce. La originadora se pasa al constructor del memento, junto con los valores de su estado. Gracias a la estrecha relación entre estas clases, un memento puede restaurar el estado de su originadora, siempre que esta última haya definido los modificadores (setters) adecuados.

Pseudocódigo

Este ejemplo utiliza el patrón Memento junto al patrón **Command** para almacenar instantáneas del estado complejo del editor de texto y restaurar un estado previo a partir de estas instantáneas cuando sea necesario.



Guardar instantáneas del estado del editor de texto.

Los objetos de comando actúan como cuidadores. Buscan el memento del editor antes de ejecutar operaciones relacionadas con los comandos. Cuando un usuario intenta deshacer el comando más reciente, el editor puede utilizar el memento almacenado en ese comando para revertirse a sí mismo al estado previo.

La clase memento no declara ningún campo, consultor (getter) o modificador (setter) como público. Por lo tanto, ningún objeto puede alterar sus contenidos. Los mementos se vinculan al objeto del editor que los creó. Esto permite a un memento restaurar el estado del editor vinculado pasando los datos a través de modificadores en el objeto editor. Ya que los mementos están vinculados a objetos de editor específicos, puedes hacer que tu aplicación soporte varias ventanas de editor independientes con una pila centralizada para deshacer.

```
// El originador contiene información importante que puede
// cambiar con el paso del tiempo. También define un método para
// guardar su estado dentro de un memento, y otro método para
// restaurar el estado a partir de él.
class Editor is
    private field text, curX, curY, selectionWidth

    method setText(text) is
        this.text = text

    method setCursor(x, y) is
```

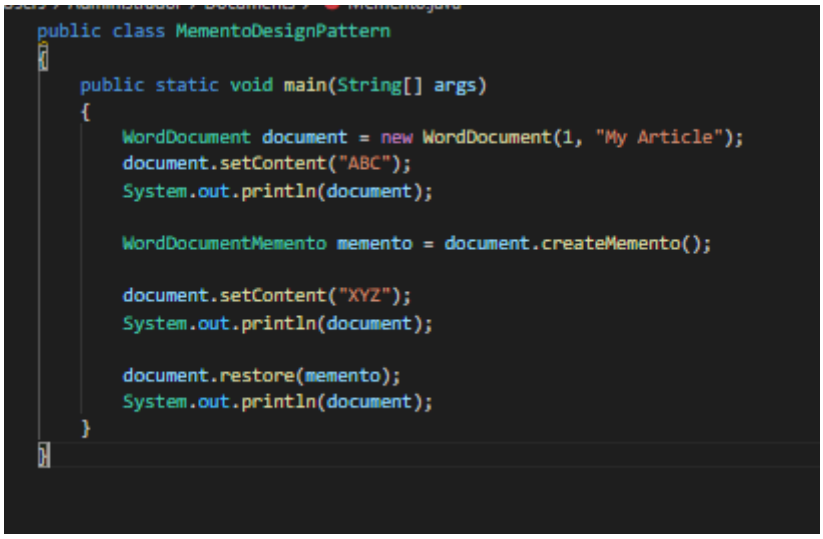
```
        this.curX = x
        this.curY = y

    method setSelectionWidth(width) is
        this.selectionWidth = width

    // Guarda el estado actual dentro de un memento.
    method createSnapshot():Snapshot is
        // El memento es un objeto inmutable; ese es el motivo
        // por el que el originador pasa su estado a los
        // parámetros de su constructor.
        return new Snapshot(this, text, curX, curY, selectionWidth)

// La clase memento almacena el estado pasado del editor.
class Snapshot is
    private field editor: Editor
    private field text, curX, curY, selectionWidth

    constructor Snapshot(editor, text, curX, curY, selectionWidth) is
        this.editor = editor
        this.text = text
        this.curX = x
        this.curY = y
        this.selectionWidth = selectionWidth

    // En cierto punto, puede restaurarse un estado previo del
    // editor utilizando un objeto memento.
    method restore() is
        editor.setText(text)
        editor.setCursor(curX, curY)
        editor.setSelectionWidth(selectionWidth)

// Un objeto de comando puede actuar como cuidador. En este
// caso, el comando obtiene un memento justo antes de cambiar el
// estado del originador. Cuando se solicita deshacer, restaura
// el estado del originador a partir del memento.
class Command is
    private field backup: Snapshot

    method makeBackup() is
        backup = editor.createSnapshot()

    method undo() is
        if (backup != null)
            backup.restore()
    // ...
```

EJEMPLO EN LA VIDA REAL:

RECUPERACION DEL ESTADO INICIAL DE UN DOCUMENTO DE WORD

```java
public class WordDocument
{
    private long id;
    private String title;
    private String heading;
    private String description;

    public WordDocument(long id, String title) {
        super();
        this.id = id;
        this.title = title;
    }

    public WordDocumentMemento createMemento()
    {
        WordDocumentMemento d = new WordDocumentMemento(id, title, heading, description);
        return d;
    }

    public void restore(WordDocumentMemento d) {
        this.id = d.getId();
        this.title = d.getTitle();
        this.heading = d.getHeading();
        this.description = d.getDescription();
    }

    @Override
    public String toString() {
        return "Word Document[id=" + id + ", title=" + title + ", heading="+ heading +", description=" + description + "]";
    }
```

```java
public final class WordDocumentMemento
{
    private final long id;
    private final String title;
    private final String heading;
    private final String description;

    public WordDocumentMemento(long id, String title, String heading, String description) {
        super();
        this.id = id;
        this.title = title;
        this.heading = heading;
        this.description = description;
    }

    public long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public String getHeading() {
        return heading;
    }

    public String getDescription() {
        return description;
    }
}

    }
```



```java
public class MementoDesignPattern
{
    public static void main(String[] args)
    {
        WordDocument document = new WordDocument(1, "My Article");
        document.setContent("ABC");
        System.out.println(document);

        WordDocumentMemento memento = document.createMemento();

        document.setContent("XYZ");
        System.out.println(document);

        document.restore(memento);
        System.out.println(document);
    }
}
```