Patrones de diseño en Java

Introducción

Los patrones de diseño son técnicas que permiten resolver problemas comunes en el desarrollo de software y en otros ámbitos referentes al diseño de interacción o interfaces.

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

Por tanto no es obligatorio utilizar los patrones. Sólo es aconsejable en el caso de detectar el mismo problema o uno similar para el cual existe un patrón que proporciona una solución probada, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. Abusar o forzar el uso de los patrones puede ser contraproducente.

- https://www.geeksforgeeks.org/software-design-patterns/
- https://www.geeksforgeeks.org/java-design-patterns

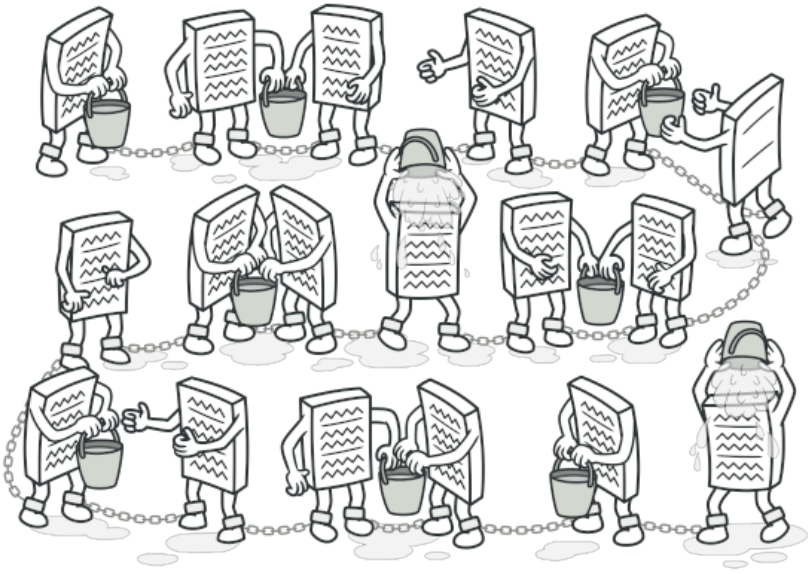
"Behavioral Patterns"

Los patrones de comportamiento se definen como patrones de diseño de software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

Los patrones de comportamiento son ***Chain of Responsibility***, ***Command***, ***Interpreter***, ***Iterator***, ***Mediator***, ***Memento***, ***Observer***, ***State***, ***Strategy***, ***Template Method*** y ***Visitor***.

- https://www.baeldung.com/java-behavioral-patterns-jdk

"Chain of Responsibility Pattern"

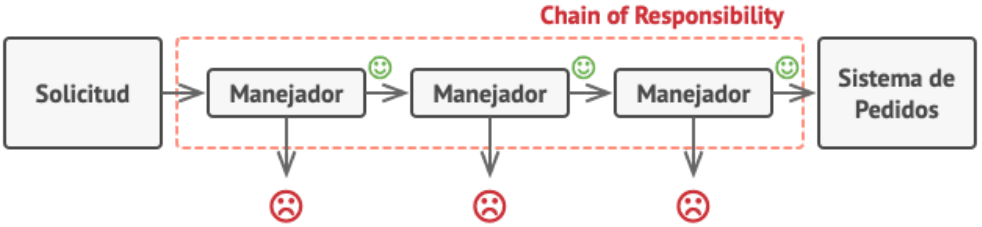


© Refactoring Guru

> *Evite acoplar el remitente de una solicitud a su receptor dándole a más de un objeto la oportunidad de manejar la solicitud. Encadene los objetos receptores y pase la solicitud a lo largo de la cadena hasta que un objeto la maneje.*
>
> *-- GoF*

Concepto

Este patrón evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición. Para ello, se encadenan los receptores y se pasa la petición a través de la cadena hasta que es procesada por algún objeto.



© Refactoring Guru

Este patrón puede aplicarse cuando:

- hay más de un objeto que puede manejar una petición, y el manejador no se conoce a priori, sino que debería determinarse automáticamente.
- se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
- el conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

Con este patrón se **reduce el acoplamiento**. El patrón libera a un objeto de tener que saber qué otro objeto maneja una petición. Ni el receptor ni el emisor se conocen explícitamente entre ellos, y un objeto de la cadena tampoco tiene que conocer la estructura de ésta. Por lo tanto, simplifica las interconexiones entre objetos. En vez de que los objetos mantengan referencias a todos los posibles receptores, sólo tienen una única referencia a su sucesor.

Además, **añade flexibilidad para asignar responsabilidades a objetos**. Se pueden añadir o cambiar responsabilidades entre objetos para tratar una petición modificando la cadena de ejecución en tiempo de ejecución. Esto se puede combinar con la herencia para especializar los manejadores estáticamente.

Por otra parte presenta el inconveniente de **no garantizar la recepción**. Dado que las peticiones no tienen un receptor explícito, no hay garantías de que sean manejadas. La petición puede alcanzar el final de la cadena sin haber sido procesada.

Caso práctico

En este patrón participan:

- **Manejador**: define una interfaz para tratar las peticiones. Opcionalmente, implementa el enlace al sucesor.
- **ManejadorConcreto**: trata las peticiones de las que es responsable; si puede manejar la petición, lo hace; en caso contrario la reenvía a su sucesor.
- **Cliente**: inicializa la petición a un manejador concreto de la cadena.

```
abstract class Manejador {
    Manejador sucesor;

    void setSucesor(Manejador sucesor) {
        this.sucesor = sucesor;
    }

    abstract void manejarPeticion(Peticion peticion);
}

class ManejadorConcreto1 extends Manejador {
    @Override
    public void manejarPeticion(Peticion peticion) {
        if (peticion.getValue() < 0) {
            System.out.println("Los valores negativos son manejados por " + getClass().getSimpleName());
            System.out.println("Valores peticion : " + peticion.getDescripcion() + peticion.getValue());
        } else {
            sucesor.manejarPeticion(peticion);
        }
    }

}

class ManejadorConcreto2 extends Manejador {
    @Override
    public void manejarPeticion(Peticion peticion) {
        if (peticion.getValue() >= 0) {
            System.out.println("Los valores mayores que 0 son manejados por " + getClass().getSimpleName());
            System.out.println("Valores peticion : " + peticion.getDescripcion() + peticion.getValue());
        } else {
            sucesor.manejarPeticion(peticion);
        }
    }
}

class Peticion {
    private int value;
    private String descripcion;

    Peticion(String descripcion, int value) {
        this.value = value;
        this.descripcion = descripcion;
    }

    int getValue() {
        return value;
    }

    String getDescripcion() {
        return descripcion;
    }
}
```

Consideraciones

Aplicar el patrón cuando se espera que su programa **procese diferentes tipos de solicitudes de varias maneras**, pero los tipos exactos de solicitudes y sus secuencias son desconocidos de antemano. El patrón le permite vincular varios manejadores en una cadena y, al recibir una solicitud, "preguntar" a cada manejador si puede procesarlo. De esta manera todos los manejadores tienen la oportunidad de procesar la solicitud.

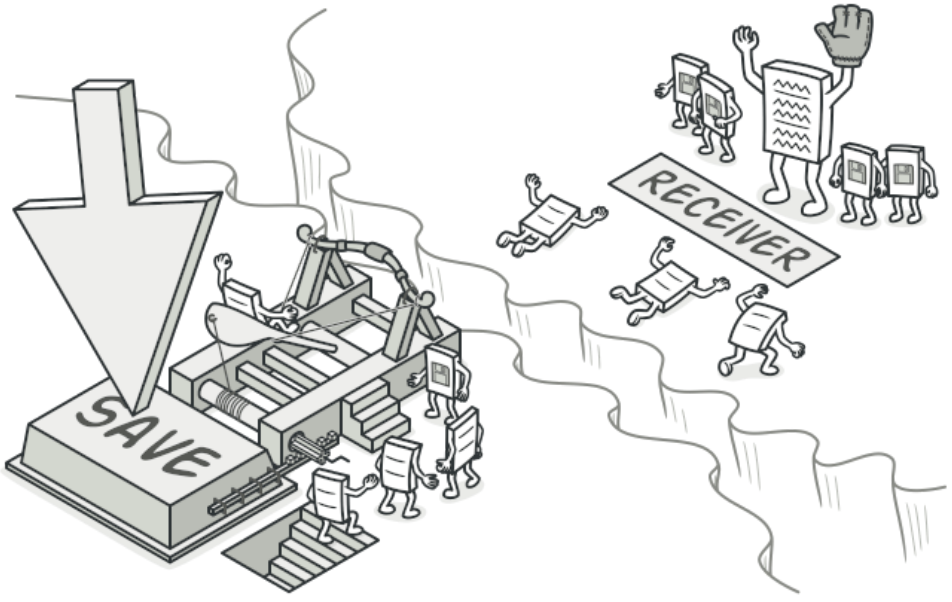
También se debería aplicar cuando sea esencial ejecutar varios manejadores en un orden particular. Como los manejadores de la cadena pueden vincularse en cualquier orden, todas las solicitudes pasarán a través de la cadena exactamente como deben hacerlo.

Por úlimo, cuando el conjunto de manejadores y su orden deban cambiar en el tiempo de ejecución también se debería aplicar este patrón. Si se proporcionan *'setters'* para un campo de referencia dentro de las clases de manejadores, podrá insertar, eliminar o ordenar los manejadores dinámicamente.

Referencias

- https://refactoring.guru/es/design-patterns/chain-of-responsibility
- https://sourcemaking.com/design_patterns/chain_of_responsibility
- https://www.baeldung.com/chain-of-responsibility-pattern
- https://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern
- https://www.digitalocean.com/community/tutorials/chain-of-responsibility-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/chain-of-responsibility/

"Command Pattern"



© Refactoring Guru

Concepto

> *Encapsule una solicitud como un objeto, lo que le permite parametrizar a los clientes con diferentes solicitudes, solicitudes de cola o registro y admite operaciones que no se pueden deshacer.*
>
> *-- GoF*

El ***'Command Pattern'*** permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además facilita la parametrización de

los métodos.

En general, se asocian cuatro términos: **invocador**, **cliente**, **comando** y **receptor**.

El objeto **cliente** crea y contiene el objeto invocador y los objetos de comando asociados con el receptor. El cliente decide cuál de estos comandos debe ejecutar en un momento determinado, como por ejemplo como respuesta a un botón pulsado en la UI. En ese momento, el cliente pasa el objeto de comando concreto al invocador para ejecutar la acción en el receptor asociada a ese comando.

Un **invocador** solo conoce la interfaz de comandos, pero desconoce totalmente los comandos concretos. El invocador recibe la orden de ejecución y el comando a ejecutar e invoca el método del comando.

Un objeto de **comando** es capaz de llamar a un método particular en el receptor. La lógica de la acción a ejecutar está definida en el método en el **receptor** que es invocado por el **comando**.

El uso del ***'Command Pattern'*** puede ser productivo en aquellas situaciones y escenarios en las que la relación directa entre el emisor de una orden y el receptor de la misma es insuficiente:

- La invocación directa afecta sólo a emisor y receptor por lo que resulta complicado ampliar dicha relación a otros actores, como por ejemplo barras de progreso, ayuda contextual, etc…
- A veces es necesario un modelo de ejecución transaccional (al igual que en las bases de datos) en las que es necesario que se ejecuten todas las órdenes o si en caso de que alguna falle o no sea posible ejecutar, se deshagan todas las órdenes relacionadas y se mantenga o restaure el estado anterior.
- En las aplicaciones multi-hilo, este patrón es un método sencillo para desacoplar productores y receptores de órdenes, dado que los productores y receptores pueden estar en diferentes hilos.
- En los juegos en red mayormente se necesitan ejecutar órdenes en todos los dispositivos participantes. Este patrón facilita la serialización de las órdenes ya que sólo hay que serializar los objetos que las representan.
- Muchos juegos añaden algún tipo de consola para interactuar directamente con el motor del juego empleando un intérprete de comandos. Mediante este patrón, se pueden sintetizar las órdenes como si hubieran sido producidas por el propio motor, facilitando la prueba y su depuración.
- En el uso de macros también se hace necesario el uso de este patrón.
- En el caso de los asistentes o *'wizards'* este patrón permite desacoplar el interfaz de usuario de las órdenes ya que una vez configurada la secuencia y tipo de órdenes desde el asistente, cuando el usuario las acepta es cuando se emite el mensaje y se ejecutan dichas órdenes.

Caso práctico

```
interface ICommand {
    void action();
}

// Command
class Redo implements ICommand {
    private Receiver receiver;

    public Redo(Receiver recv) {
        receiver = recv;
    }

    @Override
    public void action() {
        // Invoca la acción en el receptor
        receiver.performRedo();
```

```
    }
}

// El receptor contiene la lógica a ejecutar
class Receiver {
    void performUndo() {
        System.out.println("Executing - Undo");
    }

    void performRedo() {
        System.out.println("Executing - Redo");
    }
}

// Un invocador sólo conoce la interfaz de comandos
class Invoker {
    public void execute(ICommand cmd) {
        cmd.action();
    }
}
```

Consideraciones

Como se ha comentado, este patrón desacopla el objeto que invoca la operación del objeto que sabe realizarla.

Las órdenes son objetos y por tanto pueden ser manipulados y extendidos como cualquier otro objeto. Además, permite que sean gestionados en colas de objetos o mantener un registro de mensajes.

Las órdenes pueden incluir transacciones para garantizar la consistencia y estado de los objetos. Por tanto, existe la posibilidad de restaurar estados anteriores.

Al usar objetos se facilita el uso de parámetros a la hora de ejecutar las órdenes.

Este patrón es ampliamente utilizado para operaciones de deshacer/rehacer.

Una función de devolución de llamada se puede diseñar con este patrón.

Referencias

- https://refactoring.guru/es/design-patterns/command
- https://sourcemaking.com/design_patterns/command
- https://www.baeldung.com/java-command-pattern
- https://www.tutorialspoint.com/design_pattern/command_pattern
- https://www.digitalocean.com/community/tutorials/command-design-pattern
- https://java-design-patterns.com/es/patterns/command
- https://www.infoworld.com/article/3667498/how-to-use-the-command-pattern-in-java.html

---

"Interpreter Pattern"

> *Dado un lenguaje, defina una representación para su gramática junto con un intérprete que use la representación para interpretar oraciones en el lenguaje.*
>
> *-- GoF*

Concepto

Es un patrón de diseño que, dado un lenguaje, define una representación para su gramática junto con un intérprete del lenguaje.

El ***'Interpreter Pattern'*** habla de definir un lenguaje de dominio (es decir, la caracterización del problema) como una gramática de lenguaje simple, representar reglas de dominio como oraciones de lenguaje e interpretar estas oraciones para resolver el problema. El patrón usa una clase para representar cada regla gramatical. Dado que las gramáticas suelen tener una estructura jerárquica, una jerarquía de herencia de las clases de reglas se comporta bien.

El patrón sugiere modelar el dominio con una gramática recursiva. El intérprete se basa en el recorrido recursivo del ***'Composite Pattern'*** para interpretar las "oraciones" que se le solicita procesar.

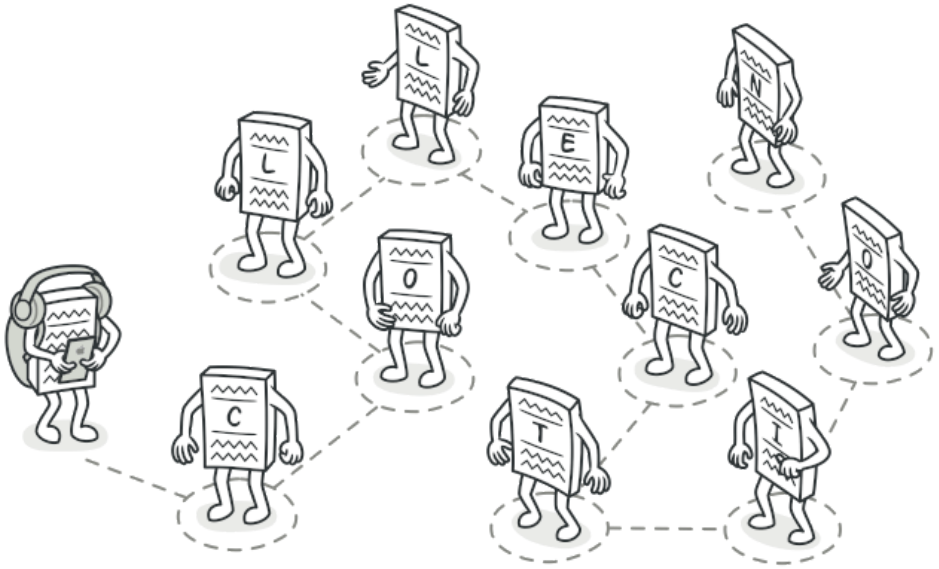
El patrón sugiere definir una gramática para un lenguaje simple definiendo una jerarquía de clases `Expression` e implementando una operación `interpret()`. Cada regla de la gramática es un 'compuesto' (una regla que hace referencia a otras reglas) o un 'terminal' (un nodo de hoja en una estructura de árbol).

La sentencia del lenguaje se representa mediante un árbol de sintaxis abstracta (AST) formado por instancias de `Expression`. Las sentencias se interpretan realizando un recorrido del AST e invocando `interpret()`.

Referencias

- https://sourcemaking.com/design_patterns/interpreter
- https://www.baeldung.com/java-interpreter-pattern
- https://www.tutorialspoint.com/design_pattern/interpreter_pattern
- https://java-design-patterns.com/es/patterns/interpreter/

"Iterator Pattern"



© Refactoring Guru

> *Proporcionar una forma de acceder a los elementos de un objeto agregado de forma secuencial sin exponer su representación subyacente.*
>
> *-- GoF*

Concepto

El ***'Iterator Pattern'*** se utiliza para ofrecer una interfaz de acceso secuencial a una determinada estructura ocultando la representación interna y la forma en que realmente se accede, o lo que es lo mismo, permite realizar recorridos sobre objetos compuestos independientemente de la implementación de éstos.

La solución que propone este patrón es añadir métodos que permitan recorrer la estructura sin referenciar explícitamente su representación, es decir, sin exponer su representación interna. La responsabilidad del recorrido se traslada a un objeto iterador.

El problema de introducir este objeto iterador reside en que los clientes necesitan conocer la estructura para crear el iterador apropiado. Esto se soluciona generalizando los distintos iteradores en una abstracción y dotando a las estructuras de datos de un método de fabricación que cree un iterador concreto.

Diferentes iteradores pueden presentar diferentes tipos de recorrido sobre la estructura. No sólo eso, sino que podrían incorporar funcionalidad extra como por ejemplo filtrado de elementos, etc..

El uso del ***'Iterator Pattern'*** es muy común ya que manejar colecciones de datos es algo muy habitual en el desarrollo de aplicaciones. Listas, pilas y, sobre todo, árboles son ejemplos de estructuras de datos muy presentes en los juegos y se utilizan de forma intensiva.

Una operación muy habitual es recorrer las estructuras para analizar y/o buscar los datos que contienen. Es posible que sea necesario recorrer la estructura de forma secuencial, de dos en dos o, simplemente, de forma aleatoria. Los clientes suelen implementar el método concreto con el que desean recorrer la estructura por lo que puede ser un problema si, por ejemplo, se desea recorrer una misma estructura de datos de varias formas distintas. Conforme aumenta las combinaciones entre los tipos de estructuras y métodos de acceso, el problema se agrava.

Caso práctico

La estructura de datos es la encargada de crear el iterador adecuado para ser accedida a través del método `iterator()`. Una vez que el cliente ha obtenido el iterador, puede utilizar los métodos de acceso que ofrecen tales como `next()` (para obtener el siguiente elemento) o `isDone()` para comprobar si existen más elementos.

```
class RandomData {
    int[] data;

    RandomData(int length) {
        data = new int[length];
        for (int i = 0; i < data.length; i++) {
            data[i] = new Random().nextInt(500);
        }
    }

    IteratorData iterator() {
        return new IteratorData(this);
    }
}

// Iterador que permite recorrer el objeto 'RandomData' sin conocer su implementación exacta
class IteratorData {
    private int[] data;
    private int pos;

    IteratorData(RandomData randomData) {
        data = randomData.data;
        pos = 0;
    }

    boolean hasNext() {
        return pos < data.length;
    }

    Object next() {
        int valor = data[pos];
        pos++;
        return valor;
    }
}

public class Client {
```

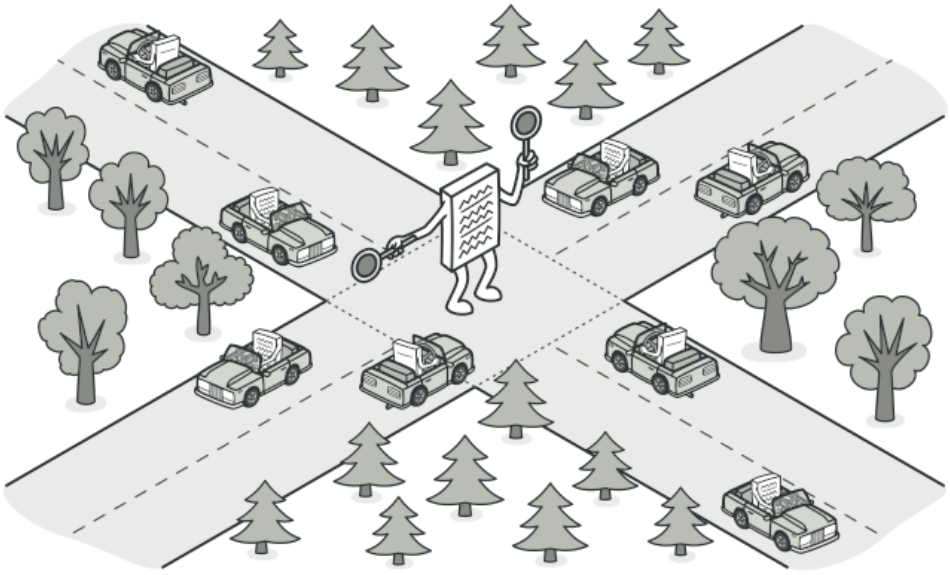
```
    public static void main(String[] args) {
        RandomData vector = new RandomData(5);

        IteratorData iterator = vector.iterator();

        while (iterator.hasNext())
            System.out.println(iterator.next());
    }
}
```

Referencias

- https://refactoring.guru/es/design-patterns/iterator
- https://sourcemaking.com/design_patterns/iterator
- https://www.baeldung.com/java-iterator
- https://www.tutorialspoint.com/design_pattern/iterator_pattern
- https://www.digitalocean.com/community/tutorials/iterator-design-pattern-java
- https://java-design-patterns.com/es/patterns/iterator
- https://www.infoworld.com/article/2461744/java-language-iterating-over-collections-in-java-8.html

---

"Mediator Pattern"



© Refactoring Guru

> *Defina un objeto que encapsule cómo interactúa un conjunto de objetos. El mediador promueve el acoplamiento débil evitando que los objetos se refieran entre sí explícitamente, y le permite variar su interacción de forma independiente.*
>
> *-- GoF*

Concepto

Normalmente los programas tienen un gran número de clases. A medida que se agregan más clases a un programa, especialmente durante el mantenimiento y/o refactorización, el problema de la comunicación entre estas clases puede volverse más complejo. Esto hace que el programa sea más difícil de leer y mantener. Además, puede resultar difícil cambiar el programa, ya que cualquier cambio puede afectar al código en otras clases.

Definir un conjunto de objetos que interactúan accediendo y actualizándose entre ellos de forma directa es inflexible porque acopla estos objetos entre sí y hace imposible cambiar la interacción de forma independiente sin tener que modificarlos. Esto

hace que los objetos sean poco reutilizables y difíciles de probar. Los objetos estrechamente acoplados son difíciles de implementar, cambiar, probar y reutilizar porque se refieren y conocen muchos objetos diferentes.

El ***'Mediator Pattern'*** define **un objeto que encapsula cómo un conjunto de objetos interactúan**. De esta forma los objetos no se comunican de forma directa entre ellos, sino que se comunican mediante el mediador. El mediador busca reducir el acoplamiento evitando que los objetos se relacionen entre ellos de forma explícita y permitiendo cambiar la interacción entre un conjunto de objetos de forma independiente.

Caso práctico

La esencia del ***'Mediator Pattern'*** es **definir un objeto que encapsule cómo interactúa un conjunto de objetos**. Promueve el acoplamiento débil evitando que los objetos se refieran entre sí explícitamente, y permite que su interacción se varíe de forma independiente. Las clases de clientes pueden usar el mediador para enviar mensajes a otros clientes y pueden recibir mensajes de otros clientes a través de un evento en la clase de mediadores.

En el ejemplo participan:

- ***Mediator***: Define la interfaz para la comunicación entre objetos de tipo `Colleague`.
- ***ConcreteMediator***: implementa la interfaz del mediador y coordina la comunicación entre los objetos de tipo `Colleague`. Es consciente de todos los objetos y sus propósitos con respecto a la intercomunicación.
- ***Colleague***: Define la interfaz para la comunicación con otros objetos `Colleague` a través de su mediador.
- ***ConcreteColleague***: implementa la interfaz `Colleague` y se comunica con otros objetos a través de su mediador

```
interface Mediator {
    void send(String message, Colleague colleague);
}

class ConcreteMediator implements Mediator {
    private ArrayList<Colleague> colleagues;

    ConcreteMediator() {
        colleagues = new ArrayList<>();
    }

    void addColleague(Colleague colleague) {
        colleagues.add(colleague);
    }

    public void send(String message, Colleague originator) {
        for (Colleague colleague : colleagues) {
            //don't tell ourselves
            if (colleague != originator) {
                colleague.receive(message);
            }
        }
    }
}

abstract class Colleague {
    private Mediator mediator;

    Colleague(Mediator m) {
        mediator = m;
    }

    //send a message via the mediator
    void send(String message) {
        mediator.send(message, this);
    }
```

```
    public abstract void receive(String message);
}

class ConcreteColleague extends Colleague {
    ConcreteColleague(Mediator mediator) {
        super(mediator);
    }

    public void receive(String message) {
        System.out.println("Colleague Received: " + message);
    }
}
```

Consideraciones

El ***'Facade Pattern'*** y el ***'Mediator Pattern'*** son muy similares ya que ambos tratan de organizar la colaboración entre un montón de clases estrechamente acopladas.

Sin embargo en el ***'Facade Pattern'*** los objetos pueden comunicarse entre sí mientras que en el ***'Mediator Pattern'*** los objetos sólo se comunican con el mediador.

Referencias

- https://refactoring.guru/es/design-patterns/mediator
- https://sourcemaking.com/design_patterns/mediator
- https://www.baeldung.com/java-mediator-pattern
- https://www.tutorialspoint.com/design_pattern/mediator_pattern
- https://www.digitalocean.com/community/tutorials/mediator-design-pattern-java
- https://java-design-patterns.com/es/patterns/mediator/

"Memento Pattern"

© Refactoring Guru

> *Sin violar la encapsulación, captura y externaliza el estado interno de un objeto para que el objeto pueda restaurarse a este estado más adelante.*
>
> *-- GoF*

Concepto

Es un patrón de diseño cuya finalidad es almacenar el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior.

Se usa este patrón cuando se quiere poder restaurar el sistema desde estados pasados y por otra parte, es usado cuando se desea facilitar el hacer y deshacer de determinadas operaciones, para lo que habrá que guardar los estados anteriores de los objetos sobre los que se opere (o bien recordar los cambios de forma incremental).

Caso práctico

El ***'Memento Pattern'*** define tres roles principales:

- ***Originator***: El *'originator'* puede producir instantáneas de su propio estado, así como restaurar su estado a partir de las instantáneas cuando sea necesario.

- ***Caretaker***: El *'caretaker'* sabe no solo "cuándo" y "por qué" capturar el estado del *'originator'*, sino también cuándo debe restaurarse el estado. Puede realizar un seguimiento del historial del *'originator'* almacenando una pila de *'mementos'*. Cuando el *'originator'* tiene que retroceder en la historia, el *'caretaker'* busca el recuerdo más alto de la pila y lo pasa al método de restauración del *'originator'*.

- ***Memento***: Es un objeto de valor que actúa como una instantánea del estado del *'originator'*. Es una práctica común hacer que el *'memento'* sea inmutable y pasar los datos solo una vez, a través del constructor.

La clase `Originator` implementa el método `createMemento()` que crea y devuelve un objeto de tipo `Memento` que almacena el estado interno actual del propio objeto de tipo `Originator` y el método `restore(Memento memento)` que restaura el estado a partir del objeto de tipo `Memento` pasado como argumento.

Para guardar el estado original, la clase `'Caretaker'` invoca el método `'createMemento()'` que creará un objeto de tipo `'Memento'` con el estado actual y lo retornará al invocador que lo custodiará (sin alterar su estado ni acceder a él). Cuando sea necesario restaurar el estado anterior, el objeto `'Caretaker'` invocará el método `'restore(memento)'` especificando el objeto `'Memento'` que guarda el estado que debe ser restaurado. El objeto *'originator'* recupera el estado del método `'getState()'` del objeto `'Memento'`.

```
// "Originator.java"
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento..

    void set(String state) {
        this.state = state;
        System.out.println("Originator: Setting state to " + state);
    }

    Memento createMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(this.state);
    }

    void restore(Memento memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }

    static class Memento {
        private final String state;
```

```
        Memento(String stateToSave) {
            state = stateToSave;
        }

        // accessible by outer class only
        private String getSavedState() {
            return state;
        }
    }
}

// "Caretaker.java"
class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.createMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.add(originator.createMemento());
        originator.set("State4");

        originator.restore(savedStates.get(1));
    }
}
```

Consideraciones

En un escenario real en que puede haber varios puntos de restauración o *'snapshots'* del estado de un objeto en varias instancias de tiempo se puede emplear una estructura como un `ArrayList` o similares. Un número alto de puntos de restauración o *'snapshots'* requiere espacio extra de almacenamiento.

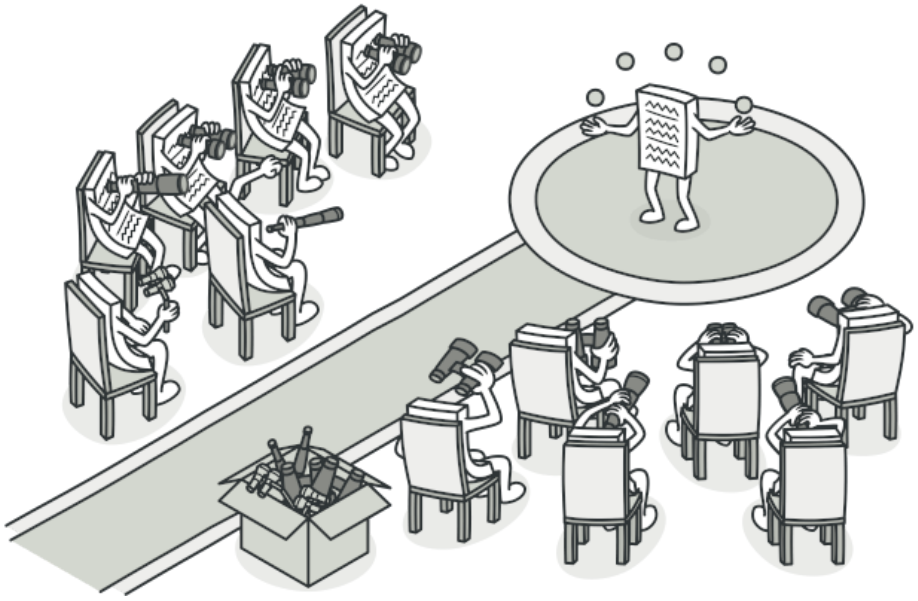
Este patrón se utiliza cuando queremos implementar operaciones como *'undo'* o *'rollback'*.

Este patrón se puede implementar utilizando la serialización, que es bastante común en Java. Si bien no es la única ni la forma más eficiente de hacer instantáneas del estado de un objeto, todavía permite almacenar copias de seguridad del estado mientras protege la estructura del *'originator'* frente a otros objetos.

Referencias

- https://refactoring.guru/es/design-patterns/memento
- https://sourcemaking.com/design_patterns/memento
- https://www.baeldung.com/java-memento-design-pattern
- https://www.tutorialspoint.com/design_pattern/memento_pattern
- https://java-design-patterns.com/es/patterns/memento

"Observer Pattern"



© Refactoring Guru

> *Define una dependencia de uno a muchos entre objetos para que cuando un objeto cambie de estado, todos sus dependientes sean notificados y actualizados automáticamente.*
>
> *-- GoF*

Concepto

El ***'Observer Pattern'*** se utiliza para definir relaciones 1 a *n* de forma que un objeto pueda notificar y/o actualizar el estado de otros automáticamente.

Este patrón proporciona un diseño con **poco acomplamiento** entre los *observadores* y el objeto *observado*.

Siguiendo la filosofía de ***publicación/suscripción***, los objetos observadores se deben registrar en el objeto observado llamado sujeto pasándole una referencia de si mismo. El sujeto mantiene una lista de las referencias de los observadores. Cuando un observador ya no necesita ser notificado simplemente se borra de la lista de observadores.

Además, los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el sujeto es capaz de notificar a los observadores suscritos los cambios que sufre para que todos ellos tengan constancia.

Por tanto, cuando el evento oportuno suceda, el sujeto recibirá una invocación y será el encargado de "notificar" a todos los elementos suscritos a él. Los observadores que reciben la notificación pueden realizar las acciones pertinentes.

Puede pensarse en aplicar este patrón cuando una modificación en el estado de un objeto requiere cambios de otros, y no se desea que se conozca el número de objetos que deben ser cambiados. También cuando queremos que un objeto sea capaz de notificar a otros objetos sin hacer ninguna suposición acerca de los objetos notificados y cuando una abstracción tiene dos aspectos diferentes, que dependen uno del otro; si encapsulamos estos aspectos en objetos separados permitiremos su variación y reutilización de modo independiente.

Caso práctico

El siguiente ejemplo es una situación en que tenemos un único observador y un único sujeto. El patrón se puede adaptar para los casos en que haya uno o varios observadores y uno o varios sujetos.

```
interface ISubject {
    void register(Observer observer);
    void unregister(Observer observer);
    void notifyObservers();
```

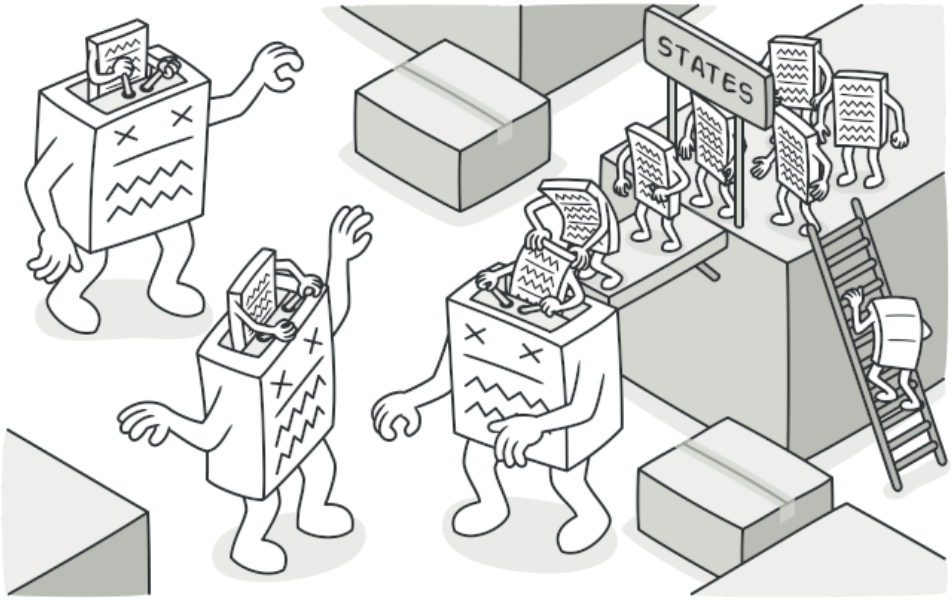
```
}

class Subject implements ISubject {
    private List<Observer> observerList = new ArrayList<>();
    private int flag;

    void setFlag(int flag) {
        this.flag = flag;
        //flag value changed. So notify observer(s)
        notifyObservers();
    }

    @Override
    public void register(Observer o) {
        observerList.add(o);
    }

    @Override
    public void unregister(Observer o) {
        observerList.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observerList) {
            observer.update();
        }
    }
}

class Observer {
    void update() {
        System.out.println("flag value changed in Subject");
    }
}
```

Referencias

- https://refactoring.guru/es/design-patterns/observer
- https://sourcemaking.com/design_patterns/observer
- https://www.baeldung.com/java-observer-pattern
- https://www.tutorialspoint.com/design_pattern/observer_pattern
- https://www.digitalocean.com/community/tutorials/observer-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/observer
- https://www.infoworld.com/article/3682139/intro-to-the-observable-design-pattern.html

---

"State Pattern"



© Refactoring Guru

> *Permitir que un objeto altere su comportamiento cuando cambia su estado interno. El objeto aparecerá para cambiar su clase.*
>
> *-- GoF*

Concepto

El ***'State Pattern'*** permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

Es muy común que en cualquier aplicación, incluido los videojuegos, existan estructuras que pueden ser modeladas directamente como **un autómata**, es decir, una colección de estados y unas transiciones dependientes de una entrada. En este caso, la entrada pueden ser invocaciones y/o eventos recibidos. También suele recibir el nombre de **máquina de estados**.

Por ejemplo, los estados de un personaje de un videojuego podrían ser: de pie, tumbado, andando y saltando. Dependiendo del estado en el que se encuentre y de la invocación recibida, el siguiente estado será uno u otro. Si por ejemplo, está de pie y recibe la orden de tumbarse, ésta se podrá realizar. Sin embargo, si ya está tumbado no tiene sentido volver a tumbarse, por lo que debe permanecer en ese estado.

El ***'State Pattern'*** permite encapsular el mecanismo de las transiciones que sufre un objeto a partir de los estímulos externos.

Caso práctico

A continuación se muestra un ejemplo de aplicación del mismo. La idea es crear una clase abstracta o interfaz que representa al estado del personaje. En ella se definen las mismas operaciones que puede recibir el personaje con una implementación por defecto. En este caso, la implementación es vacía.

Por cada estado en el que puede encontrarse el personaje, se crea una clase que hereda de la clase abstracta anterior, de forma que en cada una de ellas se implementen los métodos que producen cambio de estado, es decir, contendrán un método por cada posible transición.

Por ejemplo, según el diagrama, en el estado "de pie" se puede recibir la orden de caminar, tumbarse y saltar, pero no de levantarse. En caso de recibir esta última, se ejecutará la implementación por defecto, es decir, no hacer nada.

En definitiva, la idea es que las clases que representan a los estados sean las encargadas de cambiar el estado del personaje, de forma que los cambios de estados quedan encapsulados y delegados al estado correspondiente.

```
interface CharacterState {
    void walk();
    void getUp();
    void getDown();
    void jump();
}

// ...
class CharacterStanding implements CharacterState {
    private Character character;

    CharacterStanding(Character character) {
        this.character = character;
    }

    @Override
    public void walk() {
        System.out.println("Transición: De pie -> Andar");
        character.setState(new CharacterWalk(character));
    }

    @Override
    public void getUp() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void getDown() {
        System.out.println("Transición: De pie -> Agachado");
        character.setState(new CharacterLying(character));
    }

    @Override
    public void jump() {
        System.out.println("Transición: De pie -> Saltando");
        character.setState(new CharacterJump(character));
    }
}

// ...
class CharacterJump implements CharacterState {
    private Character character;

    CharacterJump(Character character) {
        this.character = character;
    }

    @Override
    public void walk() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void getUp() {
        System.out.println("Transición: Saltando -> Quieto");

        character.setState(new CharacterStanding(character));
    }

    @Override
    public void getDown() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void jump() {
        throw new UnsupportedOperationException();
    }
}
```

```
// ...
class Character {
    private CharacterState currentState;

    Character() {
        currentState = new CharacterStanding(this);
    }

    CharacterState getState() {
        return currentState;
    }

    void setState(CharacterState currentState) {
        this.currentState = currentState;
    }

    public void walk() {
        currentState.walk();
    }

    void getUp() {
        currentState.getUp();
    }

    void getDown() {
        currentState.getDown();
    }

    void jump() {
        currentState.jump();
    }
}
```

NOTA: Para que el ejemplo sea lo más claro posible se ha obviado de forma intencionada la aplicación del **"Principio de Segregación de Interfaces"** de los principos S.O.L.I.D. que si se aplicara correctamente se evitaría la obligación de implementar los métodos que lanzan `UnsupportedOperationException`.

Consideraciones

Este patrón puede usarse cuando un determinado objeto tiene diferentes estados y también diferentes responsabilidades según el estado en que se encuentre en un determinado instante. También puede utilizarse para simplificar casos en los que se tiene un complicado y extenso código de decisión que depende del estado del objeto.

Los componentes de diseño que se comporten como autómatas son buenos candidatos a ser modelados con el ***'State Pattern'***.

Es posible que una entrada provoque una situación de error estando en un determinado estado. Para ello es posible utilizar las excepciones para notificar dicho error.

Las clases que representan los estados **no deben mantener un estado intrínseco**, es decir, no se debe hacer uso de variables que dependan de un contexto.

Un sistema con muchos estados o si el número se incrementa significativamente se convierte en un sistema difícil de mantener.

Referencias

- https://refactoring.guru/es/design-patterns/state
- https://sourcemaking.com/design_patterns/state
- https://www.baeldung.com/java-state-design-pattern
- https://www.tutorialspoint.com/design_pattern/state_pattern
- https://www.digitalocean.com/community/tutorials/state-design-pattern-java
- https://java-design-patterns.com/es/patterns/state

"Strategy Pattern"

© Refactoring Guru

> *Defina una familia de algoritmos, encapsule cada uno de ellos y hágalos intercambiables. Este patrón permite que el algoritmo varíe independientemente de cliente a cliente.*
>
> *-- GoF*

Concepto

El ***'Strategy Pattern'*** se utiliza para encapsular el funcionamiento de una familia de algoritmos, de forma que se pueda intercambiar su uso sin necesidad de modificar los clientes. Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.

En muchas ocasiones, se suele proporcionar diferentes algoritmos para realizar una misma tarea. Por ejemplo, el nivel de habilidad de un jugador viene determinado por diferentes algoritmos y heurísticas que determinan el grado de dificultad. Utilizando diferentes tipos algoritmos podemos obtener desde jugadores que realizan movimientos aleatorios hasta aquellos que pueden tener cierta inteligencia y que se basan en técnicas de IA.

Lo deseable sería poder tener jugadores de ambos tipos y que, desde el punto de vista del cliente, no fueran tipos distintos de jugadores. Simplemente se comportan diferente porque usan distintos algoritmos internamente, pero todos ellos son jugadores.

Otro ejemplo para entender este patrón es el de un protagonista de un videojuego en el cual manejamos a un soldado que puede portar y utilizar varias armas distintas. La clase (o clases) que representan a nuestro soldado no deberían de preocuparse de los detalles de las armas que porta: debería bastar, por ejemplo, con un método de interfaz "atacar" que dispare el arma actual y otro método "recargar" que inserte munición en ésta (si se diera el caso). En un momento dado, otro método "cambiarArma" podrá sustituir el objeto equipado por otro, manteniendo la interfaz intacta.

Da igual que nuestro soldado porte un rifle, una pistola o un fusil: los detalles de cada estrategia estarán encapsulados dentro de cada una de las clases intercambiables que representan las armas. Nuestra clase cliente (el soldado) únicamente debe preocuparse de las acciones comunes a todas ellas: atacar, recargar y cambiar de arma. Éste último método, de hecho, será el encargado de realizar la operación de "cambio de estrategia" que forma parte del patrón.

Caso práctico

Mediante el uso de la herencia, el ***'Strategy Pattern'*** permite encapsular diferentes algoritmos para que los clientes puedan utilizarlos de forma transparente.

La idea es extraer los métodos que conforman el comportamiento que puede ser intercambiado y encapsularlo en una familia de algoritmos. En este ejemplo, el movimiento del jugador se extrae para formar una jerarquía de diferentes movimientos. Todos ellos implementan el método `move()` que recibe un contexto que incluye toda la información necesaria para llevar a cabo el algoritmo.

Para el cliente todo se produce de forma transparente. Al configurarse cada jugador, ambos son del mismo tipo de cara al cliente aunque ambos se comportarán de forma diferente al invocar al método `doBestMove()`.

```
// ...
interface Movement {
    void move();
}

// ...
class IAMovement implements Movement {
    @Override
    public void move() {
        System.out.println("IA Movement");
    }
}

// ...
class RandomMovement implements Movement {
    @Override
    public void move() {
        System.out.println("Random movement");
    }
}

// ...
class GamePlayer {
    private Movement movement;

    void setMovement(Movement movement) {
        this.movement = movement;
    }

    void doBestMove() {
        movement.move();
    }
}

public class Client {

    public static void main(String[] args) {
        GamePlayer player = new GamePlayer();

        Movement movement = new RandomMovement();
        player.setMovement(movement);

        player.doBestMove();
    }
}
```

Consideraciones

El ***'Strategy Pattern'*** es una buena alternativa a realizar subclases en las entidades que deben comportarse de forma diferente en función del algoritmo utilizado. Al extraer la heurística a una familia de algoritmos externos, obtenemos los siguientes beneficios:

- Se aumenta la reutilización de dichos algoritmos.
- Se evitan sentencias condicionales para elegir el comportamiento deseado.

- Los clientes pueden elegir diferentes implementaciones para un mismo comportamiento deseado, lo que es útil para depuración y pruebas donde se pueden escoger implementaciones más simples y rápidas.

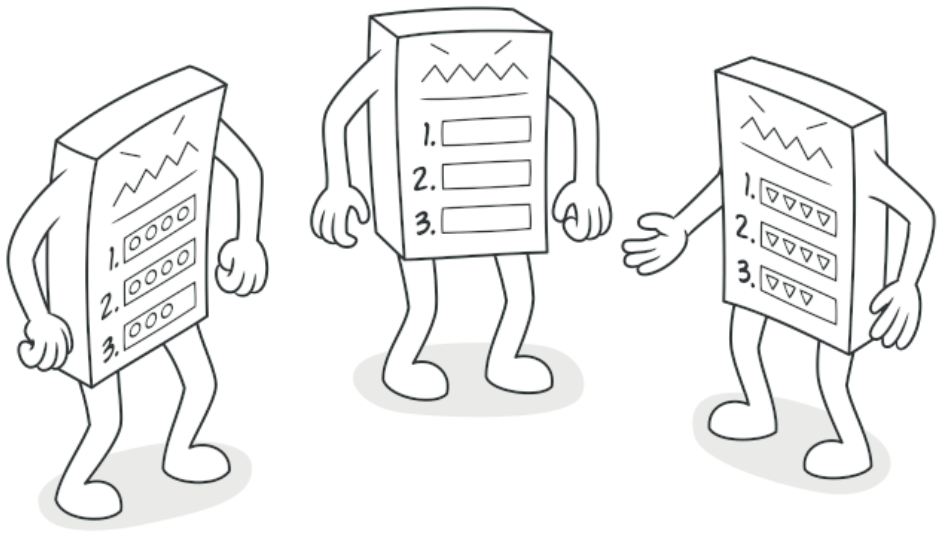
Este patrón es aconsejable, como ya hemos comentado, en situaciones en los que una misma operación (o conjunto de operaciones) puedan realizarse de formas distintas.

A grosso modo, este patrón realiza una tarea bastante similar al ***'Template Method Pattern'***. Sin embargo, mientras que el ***'Strategy Pattern'*** se basa en la **composición** y los algoritmos no están obligados a tener partes en común, el ***'Template Method Pattern'*** se basa en la **herencia** y hay pasos en común que permiten extraer una plantilla.

Referencias

- https://refactoring.guru/es/design-patterns/strategy
- https://sourcemaking.com/design_patterns/strategy
- https://www.baeldung.com/java-strategy-pattern
- https://www.tutorialspoint.com/design_pattern/strategy_pattern
- https://www.digitalocean.com/community/tutorials/strategy-design-pattern-in-java-example-tutorial
- https://java-design-patterns.com/es/patterns/strategy

"Template Method Pattern"



© Refactoring Guru

> *Define el esqueleto de un algoritmo en una operación, aplazando algunos pasos a las subclases. El método de la plantilla permite subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.*
>
> *-- GoF*

Concepto

El ***'Template Method Pattern'*** define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

En un buen diseño los algoritmos complejos se dividen en funciones más pequeñas, de forma que si se llama a dichas funciones en un determinado orden se consigue implementar el algoritmo completo. Conforme se diseña cada paso concreto, se suele ir detectando funcionalidad común con otros algoritmos.

Por ejemplo, supongamos que tenemos dos tipos de jugadores de juegos de mesa: ajedrez y damas. En esencia, ambos juegan igual; lo que cambia son las reglas del juego que, obviamente, condiciona su estrategia y su forma de jugar concreta. Sin embargo, en ambos juegos, los jugadores mueven en su turno, esperan al rival y esto se repite hasta que acaba la partida.

Este patrón consiste extraer este comportamiento común en una **clase padre** y definir en **las clases hijas** la funcionalidad concreta.

Si el ***'Command Pattern'*** nos permite encapsular una invocación a un método, el ***'Template Method Pattern'*** establece una forma de encapsular algoritmos. Este patrón se basa en un principio muy sencillo: si un algoritmo puede aplicarse a varios supuestos en los que únicamente cambie un pequeño número de operaciones, la idea será utilizar una clase para modelarlo a través de sus operaciones. Esta clase base se encargará de definir los pasos comunes del algoritmo, mientras que las clases que hereden de ella implementarán los detalles propios de cada caso concreto, es decir, el código específico para cada caso.

Caso práctico

Para aplicar este patrón se declara una clase abstracta, que será la plantilla o modelo. Esta clase definirá una serie de funciones y métodos. Aquellas que sean comunes estarán implementadas. Aquellas que dependan de cada caso concreto, se declararán como abstractas, obligando a las clases hijas a implementarlas.

Además, cada clase derivada implementará los métodos específicos, acudiendo a la clase base para ejecutar el código común.

La clase base también se encargará de la lógica del algoritmo, ejecutando los pasos en un orden preestablecido (las clases hijas no deberían poder modificar el algoritmo, únicamente definir la funcionalidad específica que tienen que implementar).

Dado que la clase padre es la que se encarga de llamar los métodos de las clases derivadas (los pasos del algoritmo estarán implementado en la clase base), se trata de una aplicación manifiesta del **'Principio de Inversión de Dependencias'**: la clase base no tiene por qué saber nada acerca de sus hijas, pero aún así, se encargará de invocar su funcionalidad cuando sea necesario. El **'Principio de Hollywood'** (“no nos llames, nosotros te llamaremos”) vuelve a entrar en escena.

La clase `GamePlayer` es la que implementa el método `play()` que es el que invoca a los otros métodos que son implementados por las clases hijas. Este método es el **método plantilla**:

```
// "GamePlayer"
abstract class GamePlayer {
    void play() {
        if (moveFirst()) {
            doBestMove();
        }

        while (!isOver()) {
            // Movimiento del rival
            //....

            if (!isOver()) {
                doBestMove();
            }
        }
    }

    abstract void doBestMove();

    abstract boolean moveFirst();

    abstract boolean isOver();
}

// "ChessPlayer.java"
class ChessPlayer extends GamePlayer {
    private int movements = 0;

    @Override
    boolean moveFirst() {
```

```
        System.out.println("El rival mueve primero");
        return false;
    }

    @Override
    void doBestMove() {
        System.out.println("Moviendo ficha - Movimiento " + movements);
        movements++;
    }

    // Como convención para el ejemplo, la partida acaba al alcanzar 50 movimientos
    @Override
    boolean isOver() {
        if (movements < 50) {
            return false;
        } else {
            System.out.println("Fin de la partida - Alcanzado los " + movements + " como máximo");
            return true;
        }
    }
}

// "CheckersPlayer.java"
class CheckersPlayer extends GamePlayer {
    private int movements = 0;

    @Override
    void doBestMove() {
        System.out.println("Moviendo ficha - Movimiento " + movements);
        movements++;
    }

    @Override
    boolean moveFirst() {
        System.out.println("Movemos primero");
        return true;
    }

    // Como convención para el ejemplo, la partida acaba al alcanzar 25 movimientos
    @Override
    boolean isOver() {
        if (movements < 25) {
            return false;
        } else {
            System.out.println("Fin de la partida - Alcanzado los " + movements + " como máximo");
            return true;
        }
    }
}
```

Consideraciones

Utilizando este patrón se suelen obtener estructuras altamente reutilizables. Esta reutilización de código es el objetivo primordial de este patrón. Es por ello que es ampliamente utilizado en bibliotecas de clases.

Además, introduce el concepto de operaciones *'hook'* que, en caso de no estar implementadas en las clases hijas, tienen una implementación por defecto. Las clases hijas pueden sobreescribirlas para añadir su propia funcionalidad.

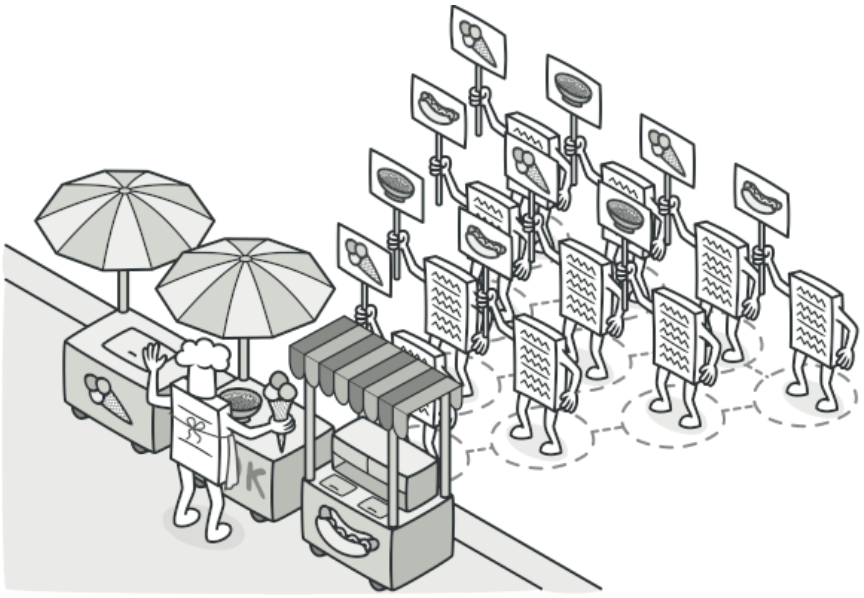
Hay que minimizar el número de métodos abstractos (métodos sin cuerpo). De lo contrario, cada una de las subclases debe sobreescribirlos y el proceso global perderá la efectividad de este patrón de diseño.

Referencias

- https://refactoring.guru/es/design-patterns/template-method
- https://sourcemaking.com/design_patterns/template_method
- https://www.baeldung.com/java-template-method-pattern

- https://www.tutorialspoint.com/design_pattern/template_pattern
- https://www.digitalocean.com/community/tutorials/template-method-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/template-method/

"Visitor Pattern"



© Refactoring Guru

> *Representa una operación a realizar en los elementos de una estructura de objeto. Este patrón le permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.*
>
> *-- GoF*

El ***'Visitor Pattern'*** proporciona un mecanismo para realizar diferentes operaciones sobre una jerarquía de objetos de forma que añadir nuevas operaciones no haga necesario cambiar las clases de los objetos sobre los que se realizan las operaciones.

Concepto

En el diseño de un programa, normalmente se obtienen jerarquías de objetos a través de herencia o utilizando el ***'Composite Pattern'***. Considerando una jerarquía de objetos que sea más o menos estable, es muy probable que necesitemos realizar operaciones sobre dicha jerarquía. Sin embargo, puede ser que cada objeto deba ser tratado de una forma diferente en función de su tipo. Por lo tanto la complejidad general aumentará según el número de objetos y de operaciones aplicables.

En el ***'Visitor Pattern'*** se distinguen dos participantes:

- **Visitables**: son los elementos de la estructura de objetos que aceptan a un determinado visitante y que le proporcionan toda la información a éste para realizar una determinada operación. Definen una operación `accept(v:Visitor)` que toma un visitante como argumento.

- **Visitantes:** jerarquía de objetos que realizan una operación determinada sobre dichos elementos.

Cada visitante concreto realiza una operación sobre la estructura de objetos. Es posible que al visitante no le interesen todos los objetos y, por lo tanto, la implementación de alguno de sus métodos sea vacía.

Sin embargo, lo importante de este patrón es que se pueden añadir nuevos tipos de visitantes concretos y, por lo tanto, realizar nuevas operaciones sobre la estructura sin la necesidad de modificar nada en la propia estructura. Por tanto se seguiría el **Principio 'Open/Closed'** (abierto a la extensión, cerrado a la modificación).

Caso práctico

```
interface Element {
    void accept(Visitor visitor);
}

class ElementA implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visitElementA(this);
    }
}

class ElementB implements Element {
    @Override
    public void accept(Visitor visitor) {
        visitor.visitElementB(this);
    }
}

interface Visitor {
    void visitElementA(Element element);
    void visitElementB(Element element);
}

class ConcreteVisitor1 implements Visitor {
    @Override
    public void visitElementA(Element element) {
        System.out.println("Visitando " + element.toString());
    }
    @Override
    public void visitElementB(Element element) {
        throw new UnsupportedOperationException();
    }
}

class ConcreteVisitor2 implements Visitor {
    @Override
    public void visitElementA(Element element) {
        throw new UnsupportedOperationException();
    }
    @Override
    public void visitElementB(Element element) {
        System.out.println("Visitando " + element.toString());
    }
}
```

Consideraciones

El ***'Visitor Pattern'*** es muy conveniente para recorrer estructuras arbóreas y realizar operaciones en base a los datos almacenados.

La forma en que se recorra la estructura influirá notablemente en el rendimiento del análisis de la estructura. Se puede hacer uso del ***'Iterator Pattern'*** para decidir cómo escoger el siguiente elemento.

Uno de los problemas de este patrón es que no es recomendable si la estructura de objetos cambia frecuentemente o es necesario añadir nuevos tipos de objetos de forma habitual. Cada nuevo objeto que sea susceptible de ser visitado puede provocar grandes cambios en la jerarquía de los visitantes.

Referencias

- https://refactoring.guru/es/design-patterns/visitor
- https://sourcemaking.com/design_patterns/visitor
- https://www.baeldung.com/java-visitor-pattern

- https://www.tutorialspoint.com/design_pattern/visitor_pattern
- https://www.digitalocean.com/community/tutorials/visitor-design-pattern-java
- https://java-design-patterns.com/es/patterns/visitor

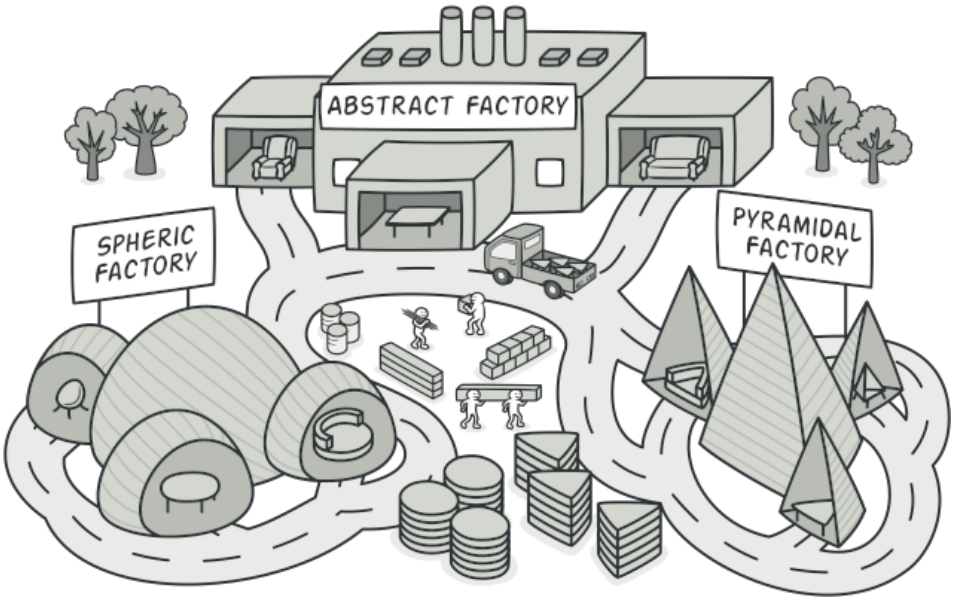
Creational Patterns

Los patrones de creación corresponden a patrones de diseño de software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstraer dicha creación.

Los patrones creacionales son ***Abstract Factory***, ***Builder***, ***Factory Method***, ***Prototype*** y ***Singleton***.

- https://www.baeldung.com/creational-design-patterns
- https://www.baeldung.com/java-creational-design-patterns

"Abstract Factory Pattern"



© Refactoring Guru

> *Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.*
>
> *-- GoF*

El ***'Abstract Factory Pattern'*** permite crear diferentes tipos o familias de instancias, aislando al cliente sobre como se debe crear cada una de ellas.

Concepto

El ***'Abstract Factory Pattern'*** recomienda crear las siguientes entidades:

- **Factoría abstracta** que defina una interfaz para que los clientes puedan crear los distintos tipos de objetos.
- **Factorías concretas** que realmente crean las instancias finales y que son hijas de la factoría abstracta.

El patrón ***'Abstract Factory'*** puede ser aplicable cuando:

- el sistema de creación de instancias debe aislarse.

- es necesaria la creación de varias instancias de objetos para tener el sistema configurado.
- cuando la creación de instancias implican la imposición de restricciones u otras particularidades propias de los objetos que se construyen.
- los objetos que deben construirse en las factorías no cambian excesivamente en el tiempo.

Añadir nuevos tipos implica cambiar todas las factorías. Por ello, se recomienda aplicar este patrón sobre diseños con un cierto grado de estabilidad.

Un ejemplo de uso de este patrón podría ser las interfaces gráficas o UI. Las bibliotecas para crear interfaces gráficas suelen utilizar este patrón. Cada familia aplica a un SO distinto.

Así pues, el usuario declara un elemento como podría ser un *Button* pero de forma más interna lo que está creando es un *WindowsButton* o un *LinuxButton* según el SO, siendo transparente para el cliente o usuario.

Otro ejemplo podría ser la jerarquía de objetos existentes en cualquier juego.

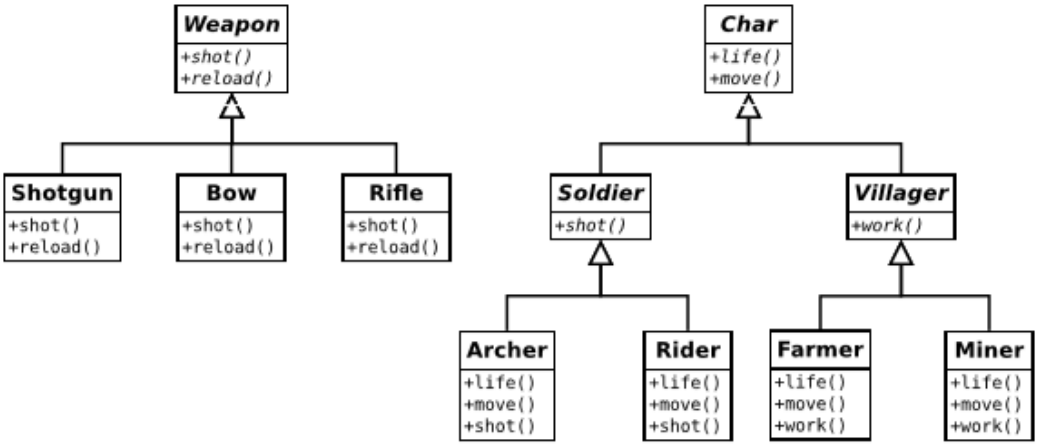


Figura 4.2: Ejemplos de jerarquías de clases

En ella, se muestra jerarquías de clases que modelan los diferentes tipos de personajes de un juego y algunas de sus armas. Para construir cada tipo de personaje es necesario saber cómo construirlo y con qué otro tipo de objetos tiene relación. Por ejemplo, restricciones del tipo "la gente del pueblo no puede llevar armas" o "los arqueros sólo pueden puede tener un arco", es conocimiento específico de la clase que se está construyendo.

Caso práctico

En primer lugar se define una factoría abstracta que será la que utilice el cliente `Game` para crear los diferentes objetos. La clase `SoldierFactory` es una factoría que sólo define métodos abstractos y que serán implementados por sus clases hijas.

Las clases hijas son factorías concretas de cada tipo de raza, como por ejemplo `ManFactory` y `OrcFactory`. Estas factorías son las encargadas de crear las instancias concretas de objetos de tipo `Archer` y/o `Rider` para cada una de las razas.

El ***'Abstract Factory Pattern'*** es de ayuda en este tipo de situaciones en las que es necesario crear diferentes tipos de objetos utilizando una jerarquía de componentes. Dada la complejidad que puede llegar a tener la creación de una instancia es deseable aislar la forma en que se construye cada clase de objeto.

```
// Jerarquía de tipos
abstract class Soldier {
    String name;
    int life;
```

```
    abstract int shotsPerSecond();
}

class Rider extends Soldier {
    Rider(String name, int life) {
        this.name = name;
        this.life = life;
    }

    @Override
    public int shotsPerSecond() {
        return 5;
    }
}

class Archer extends Soldier {
    Archer(String name, int life) {
        this.name = name;
        this.life = life;
    }

    @Override
    public int shotsPerSecond() {
        return 2;
    }
}

// Factoría abstracta
interface SoldierFactory {
    Archer makeArcher();
    Rider makeRider();
}

// Factoría concreta
class OrcFactory implements SoldierFactory {
    @Override
    public Archer makeArcher() {
        return new Archer("Orc Archer", 200);
    }
    @Override
    public Rider makeRider() {
        return new Rider("Orc Rider", 250);
    }
}

// Factoría concreta
class ManFactory implements SoldierFactory {
    @Override
    public Archer makeArcher() {
        return new Archer("Man Archer", 100);
    }

    @Override
    public Rider makeRider() {
        return new Rider("Man Rider", 150);
    }
}
```

Consideraciones

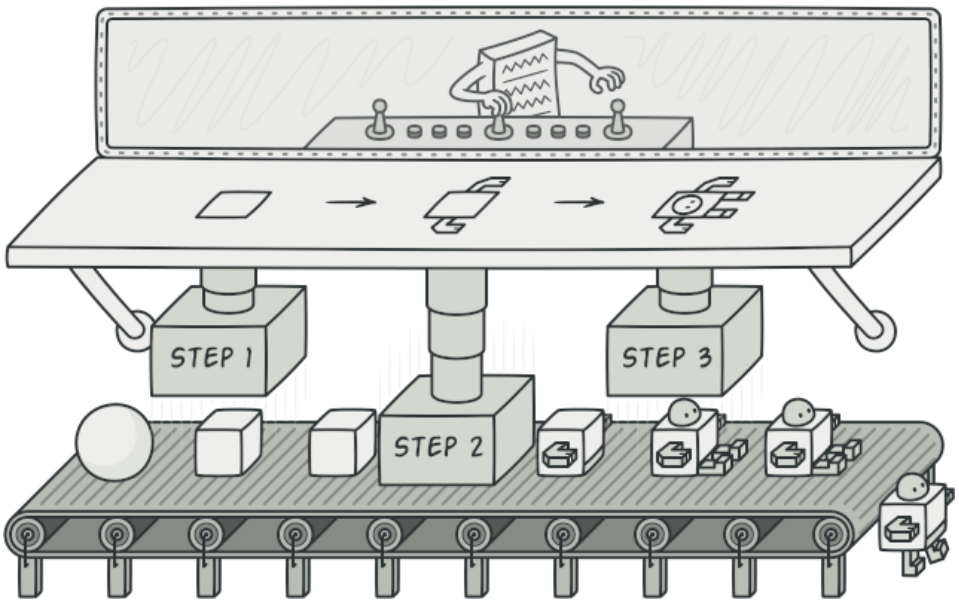
Este patrón se utiliza cuando a nuestro sistema no le importa cómo se crearán o compondrán sus productos o cuando necesitamos tratar con varias *"fábricas"* o *"factorías"*.

Este patrón separa las clases concretas y facilita el intercambio de productos. También puede mejorar la fiabilidad entre los productos. Pero, al mismo tiempo, debemos reconocer el hecho de que crear un nuevo producto es difícil con este patrón (porque necesitamos extender la interfaz y, como resultado, se requerirán cambios en todas las subclases que ya implementaron la interfaz).

Referencias

- https://refactoring.guru/es/design-patterns/abstract-factory
- https://sourcemaking.com/design_patterns/abstract_factory
- https://www.baeldung.com/java-abstract-factory-pattern
- https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern
- https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/abstract-factory/

"Builder Pattern"



© Refactoring Guru

> *Separa la construcción de un objeto complejo de su representación para que los mismos procesos de construcción puedan crear diferentes representaciones.*
>
> *-- GoF*

Concepto

El ***'Builder Pattern'*** es útil para crear **objetos complejos que tienen varias partes**. El mecanismo de creación de un objeto debe ser independiente de estas partes ya que al proceso de construcción no le importa como se ensamblan estas piezas. El mismo proceso de construcción debe permitirnos crear diferentes representaciones de los objetos.

Crear y ensamblar las partes de un objeto complejo directamente dentro de una clase es inflexible. Se compromete a la clase a crear una representación particular del objeto complejo y hace que sea imposible cambiar la representación más tarde de forma independiente de la clase sin tener que cambiarla.

Ventajas de utilizar este patrón:

- Reduce el acoplamiento.
- Permite variar la representación interna de un producto.
- Encapsula el código para la construcción y la representación.
- Proporciona control sobre los pasos del proceso de construcción.

Desventajas del patrón:

- Requiere crear un constructor concreto para cada tipo diferente de producto.
- Requiere que las clases del constructor sean mutables.
- No se garantiza la inicialización de los datos de los miembros de la clase.
- La inyección de dependencia puede ser menos compatible.

Caso práctico

Entidades que intervienen:

- ***Builder***: Interfaz abstracta para crear objetos (producto).
- ***ConcreteBuilder***: Proporciona implementación para *'Builder'*. Es un objeto capaz de construir otros objetos. Construye y ensambla partes para construir los objetos.
- ***Product***: el objeto complejo que se construye.
- ***Director***: invoca el objeto *'Builder'* para la construcción del producto.

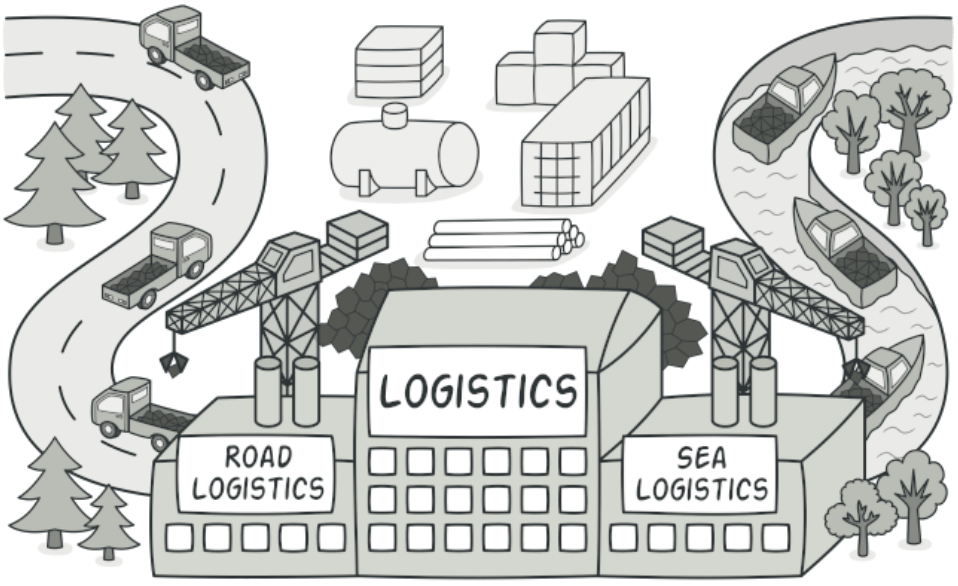
```
interface Builder {
    Product build();
    ConcreteBuilder setName(String name);
    ConcreteBuilder setColor(String color);
}

class ConcreteBuilder implements Builder {
    private Product product;

    ConcreteBuilder() {
        product = new Product();
    }

    @Override
    public Product build() {
        Product product = new Product();
        product.setColor(this.product.getColor());
        product.setName(this.product.getName());
        return product;
    }

    @Override
    public ConcreteBuilder setColor(final String color) {
        product.setColor(color);
        return this;
    }

    @Override
    public ConcreteBuilder setName(final String name) {
        product.setName(name);
        return this;
    }
}

class Product {
    // ...
}

public class Director {

    private Builder builder;
```

```
    private Director(final Builder builder) {
        this.builder = builder;
    }

    public static void main(final String ... arguments) {
        final Builder builder = new ConcreteBuilder();
        final Director director = new Director(builder);
        System.out.println(director.construct());
    }

    private Product construct() {
        return builder.setName("ProductA")
                .setColor("Red")
                .build();
    }
}
```

Referencias

- https://refactoring.guru/es/design-patterns/builder
- https://sourcemaking.com/design_patterns/builder
- https://www.baeldung.com/java-builder-pattern
- https://www.baeldung.com/java-builder-pattern-inheritance
- https://www.tutorialspoint.com/design_pattern/builder_pattern
- https://www.digitalocean.com/community/tutorials/builder-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/builder/

"Factory Method Pattern"



© Refactoring Guru

> *Defina una interfaz para crear un objeto, pero deje que las subclases decidan qué clase instanciar. Este patrón permite que una clase difiera la creación de instancias a subclases.*
>
> *-- GoF*

Concepto

El ***'Factory Method Pattern'*** centraliza en una clase constructora la **creación de objetos de un subtipo de un tipo determinado**, ocultando al usuario la diversidad de casos particulares que se pueden preveer en la elección del subtipo.

Al igual que ocurre con el ***'Abstract Factory Pattern'*** el problema que se pretende resolver es la creación de diferentes instancias de objetos abstrayendo la forma en que realmente se crean.

En el ***'Factory Method Pattern'*** se utiliza una clase constructora abstracta con métodos definidos y otro(s) abstracto(s) dedicado(s) a la construcción de objetos de un subtipo de un tipo determinado. Es una simplificación del ***'Abstract Factory Pattern'*** en la que la clase abstracta tiene métodos concretos.

A diferencia del ***'Abstract Factory Pattern'*** no es necesario tener una factoría o una jerarquía de factorías para la creación de objetos. Permite diseños más adaptados a la realidad.

Caso práctico

Las clases principales del ejemplo son el **creador** y el **producto**. El creador necesita crear instancias de productos, pero el tipo concreto de producto no debe ser forzado en las subclases del creador porque las posibles subclases del creador deber poder especificar subclases del producto a utilizar.

La solución para esto es hacer un método abstracto (el método de la fábrica) que se define en el creador. Este método abstracto se define para que devuelva un producto. Las subclases del creador pueden sobreescribir este método para devolver subclases apropiadas del producto.

```
interface Product {
    void operacion();
}

class ConcreteProductA implements Product {
    @Override
    public void operacion() {
        System.out.println("ConcreteProductA");
    }
}

class ConcreteProductB implements Product {
    @Override
    public void operacion() {
        System.out.println("ConcreteProductB");
    }
}

abstract class Creator {
    abstract Product factoryMethod();
}

class ConcreteCreatorA extends Creator {
    @Override
    Product factoryMethod() {
        return new ConcreteProductA();
    }
}

class ConcreteCreatorB extends Creator {
    @Override
    Product factoryMethod() {
        return new ConcreteProductB();
    }
}
```

Consideraciones

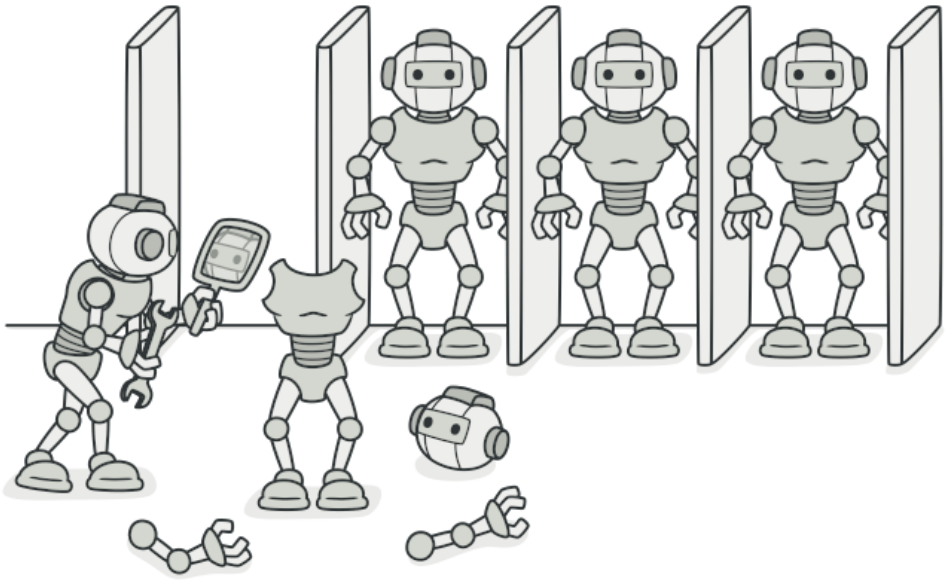
Este patrón es útil cuando las clases delegan las responsabilidades de la creación de objetos a sus subclases.

Este patrón también es útil cuando se implementan jerarquías de clases paralelas (cuando algunas de las responsabilidades cambian de una clase a otra) y cuando es posible crear un sistema con acoplamiento débil.

Referencias

- https://refactoring.guru/es/design-patterns/factory-method
- https://sourcemaking.com/design_patterns/factory_factory
- https://www.baeldung.com/java-factory-pattern
- https://www.tutorialspoint.com/design_pattern/factory_pattern
- https://www.digitalocean.com/community/tutorials/factory-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/factory/

"Prototype Pattern"



© Refactoring Guru

> *Especifique los tipos de objetos para crear utilizando una instancia prototípica y cree nuevos objetos copiando este prototipo.*
>
> *-- GoF*

Concepto

El ***'Prototype Pattern'*** proporciona abstracción a la hora de crear diferentes objetos en un contexto donde se desconoce cúantos y cuáles deben ser creados a priori. La idea principal es que los objetos deben poder clonarse o copiarse en tiempo de ejecución. Por tanto, este patrón tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente. Dado que crear una nueva instancia normalmente suele ser una operación costosa, este patrón ayuda a lidiar con ese problema.

Los ***'Factory Method Pattern'*** y ***'Abstract Factory Pattern'*** tiene el problema de que se basan en la herencia e implementación de métodos abstractos por subclases para definir cómo se construye cada producto concreto. Para sistemas donde el número de productos concretos puede ser elevado o indeterminado esto puede ser un problema.

La clase de los objetos que servirán de prototipo para la copia deberán incluir en su interfaz la manera de solicitar esa copia, que será desarrollada luego por las clases concretas de prototipos.

Este patrón se utiliza en casos como:

- Evitar las subclases de un objeto creador como hace el patrón ***'Abstract Factory'***

- Evitar el costo inherente a la creación de un objeto nuevo mediante el operador `new` cuando esto es demasiado costoso para la aplicación.
- La decisión del tipo de objeto necesario se decide en tiempo de ejecución en función de determinados parámetros, configuraciones o condiciones en un momento dado.

Este patrón, dicho de otro modo, propone la creación de distintas variantes de objetos que la aplicación necesite en el momento y contexto adecuados. Toda la lógica necesaria para la decisión sobre el tipo de objetos que usará la aplicación es su ejecución se hace independiente, de manera que el código que utiliza estos objetos solicitará una copia del objeto que necesite. En este contexto, una copia significa otra instancia del objeto. El único requisito que debe cumplir este objeto es suministrar la funcionalidad de clonarse.

Caso práctico

Para implementar este patrón se declara una clase base abstracta que tiene un método `clone()`. Cualquier clase que necesite un constructor deriva de la clase abstracta e implementa el método `clone()`.

El cliente, en vez de escribir código que hace uso del generador `new` sobre una clase específica, llama al método `clone()` de la clase prototipo, o llama a un método factoría con un parámetro que especifíca la clase deseada, o invoca el método `clone()` de la clase de alguna otra forma.

Por ejemplo, es posible usar un **gestor de prototipos** que permita cargar y descargar los prototipos disponibles en tiempo de ejecución.

Aunque en un principio este patrón parece que entra en conflicto con ***'Abstract Factory'*** es posible utilizar ambas aproximaciones en una ***'Prototype Abstract Factory'*** de forma que la factoría se configura con los prototipos concretos que puede crear y ésta sólo invoca al método `clone()`.

```
interface Producto {
    Producto copy();
}

class ProductoA implements Producto {
    private String name = "ProductoA";

    @Override
    public Producto copy() {
        return new ProductoA();
    }

    @Override
    public String toString() {
        return name;
    }
}

// Factoría que crea y copia los objetos
class FactoriaPrototipo {
    private Producto productoA;

    FactoriaPrototipo() {
        productoA = new ProductoA();
    }

    Producto create() {
        return productoA.copy();
    }
}
```

Consideraciones

Este patrón es útil para implementar *"plugins"* o cuando se cargan en tiempo de ejecución liberías dinámicas, cuando el sistema no se preocupa por el mecanismo de creación de los productos o cuando necesitamos instanciar clases en tiempo de ejecución.

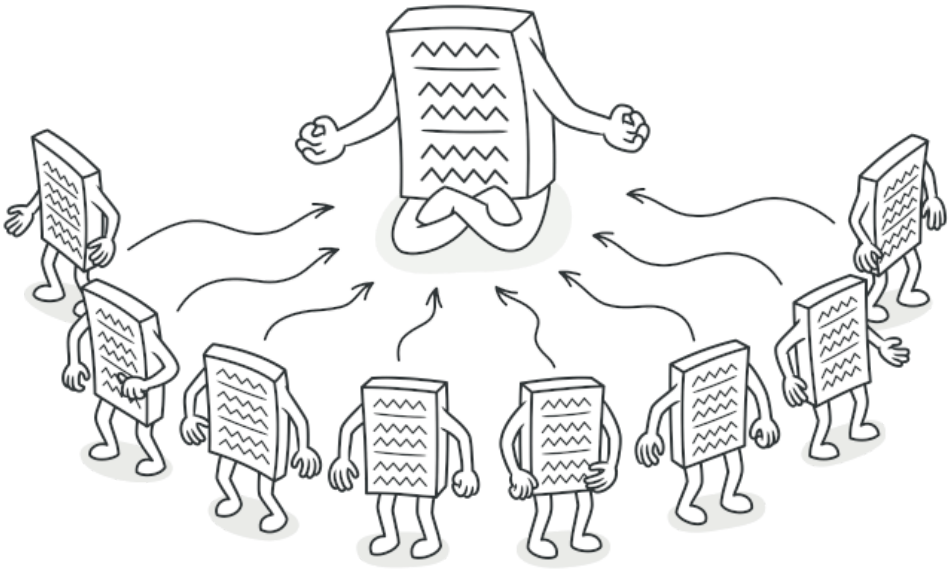
Una de las ventajas de este patrón es que ayuda a crear nuevas instancias con un coste bajo.

Por el contrario, una de las desventajas es que cada subclase tiene que implementar el mecanismo de clonación. Implementar el mecanismo de clonación puede ser un desafío si los objetos en consideración no admiten la copia o si hay algún tipo de referencia circular.

Referencias

- https://refactoring.guru/es/design-patterns/prototype
- https://sourcemaking.com/design_patterns/prototype
- https://www.baeldung.com/java-pattern-prototype
- https://www.tutorialspoint.com/design_pattern/prototype_pattern
- https://www.digitalocean.com/community/tutorials/prototype-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/prototype/

"Singleton Pattern"



© Refactoring Guru

> *Asegúrese de que una clase solo tenga una instancia y proporcione un punto de acceso global a ella.*
>
> *-- GoF*

Concepto

El ***'Singleton Pattern'*** garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia. Restringe la instanciación de una clase o valor de un tipo a un solo objeto.

En Java, los objetos normalmente se crean con el operador `new`. Sin embargo, es posible que en un determinado momento sea necesario que sólo exista una instancia de una clase concreta (prevención de errores, seguridad, creación costosa, etc..).

Para garantizar que sólo existe una instancia de una clase es necesario que los clientes no puedan acceder directamente al constructor. Por ello, cuando se aplica el ***'Singleton Pattern'*** el constructor es, por lo menos, `protected` o `private`. A cambio se debe proporcionar un único punto controlado por el cual se pide la instancia única.

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto.

El ***'Singleton Pattern'*** puede ser utilizado para modelar:

- Gestores de acceso a base de datos, sistemas de ficheros, *render* de gráficos, etc..
- Estructuras que representan la configuración del programa para que sea accesible por todos los elementos en cualquier instante.

Caso práctico

El ***'Singleton Pattern'*** provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el contructor de clase como privado para que no sea instanciable directamente.

En este ejemplo, primero hemos hecho que el constructor sea privado, de modo que no podemos crear instancias de manera normal. Además, al proporcionar un constructor privado, evitamos que el compilador genere un constructor por defecto. Cuando intentamos crear una instancia de la clase, estamos comprobando si ya tenemos una copia disponible. Si no tenemos una copia de ese tipo, la crearemos; de lo contrario, simplemente reutilizaremos la copia existente.

La inicialización de la instancia se denomina *'lazy initialization'* ya que la instancia no se crea hasta que el método `getINSTANCE()` se invoca.

```
public class Singleton {
    private static Singleton instance = null;

    // Constructor privado
    private Singleton() {}

    // Lazy initialization
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

// Modo 'Lazy Initialization'
public class Singleton {
    private static Singleton instance = null;

    // Constructor privado
    private SingletonSyncronized() {}

    // Creador sincronizado para protegerse de posibles problemas multi-hilo
    private static synchronized void createInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
    }

    public static Singleton getInstance() {
        if (instance == null) {
            createInstance();
        }
        return instance;
    }
```

```
}

// Modo 'Early Initialization'
public class Singleton {

    // Early initialization
    private static Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

Consideraciones

El ***'Singleton Pattern'*** es un caso particular de un patrón de diseño más general llamado ***'Object Pool'***, que permite crear **n instancias de objetos de forma controlada**.

Referencias

- https://refactoring.guru/es/design-patterns/singleton
- https://sourcemaking.com/design_patterns/singleton
- https://www.baeldung.com/java-singleton
- https://www.tutorialspoint.com/design_pattern/singleton_pattern
- https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples
- https://java-design-patterns.com/es/patterns/singleton/

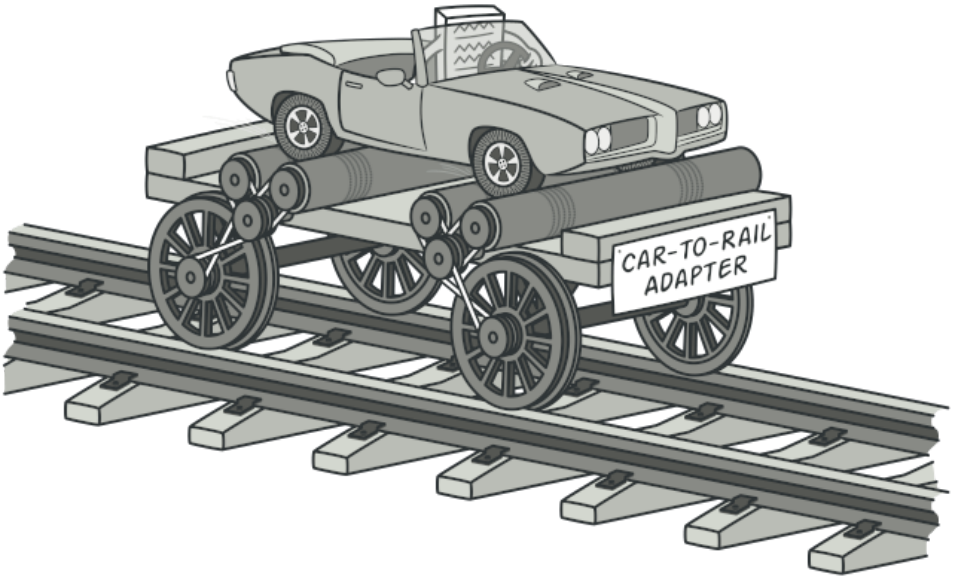
Structural Patterns

Los patrones estructurales son los patrones de diseño de software que solucionan problemas de composición (agregación) de clases y objetos.

Los patrones estructurales son ***Adapter***, ***Bridge***, ***Composite***, ***Decorator***, ***Facade***, ***Flyweight*** y ***Proxy***.

- https://www.baeldung.com/java-core-structural-patterns

"Adapter Pattern"



© Refactoring Guru

> *Convierte la interfaz de una clase en otra interfaz que los clientes esperan. Este patrón permite que las clases trabajen juntas que de otra manera no podrían hacerlo debido a interfaces incompatibles.*
>
> *-- GoF*

Concepto

El ***'Adapter Pattern'***, también conocido como ***'wrapper'*** (envoltorio), se utiliza para proporcionar una interfaz que, por un lado, cumpla con las demandas de los clientes y, por otra, haga compatible otra interfaz que, a priori, no lo es. Dicho de otra forma, se utiliza para **transformar una interfaz en otra**, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Conforme avanza la construcción de una aplicación, el diseño de las interfaces que ofrecen los componentes pueden no ser las adecuadas o, al menos, las esperadas por los usuarios de los mismos.

Una solución rápida podría ser adaptar dichas interfaces a las necesidades de la aplicación. Sin embargo, esto no siempre es posible debido a que no se pueda modificar el código porque sea un requisito funcional, sea una biblioteca de terceros, etc…

Usando el ***'Adapter Pattern'*** es posible crear una nueva interfaz de acceso a un determinado objeto, por lo que proporciona un mecanismo de **adaptación** entre las demandas del objeto cliente y el objeto servidor que proporciona la funcionalidad.

Caso práctico

El cliente no utiliza el sistema adaptado, si no que hace uso del *adaptador*. Este es el que transforma la invocación a `method()` en `otherMethod()`.

Es posible que el adaptador también incluya nueva funcionalidad relacionada con la adaptación como por ejemplo:

- Comprobación de la corrección de los parámetros.
- Transformación de los parámetros para ser compatibles con el sistema adaptado.

```
interface Target {
    String method();
}

class Adapter implements Target {
```

```
    @Override
    public String method() {
        OtherSystem otherSystem = new OtherSystem();
        return otherSystem.otherMethod();
    }
}

class OtherSystem {
    String otherMethod() {
        return OtherSystem.class.getSimpleName();
    }
}

public class Client {
    public static void main(String[] args) {
        Target target = new patterns.structural.adapter.example.Adapter();
        System.out.println("Método adaptado: " + target.method());
    }
}
```

Consideraciones

Tener sistemas muy reutilizables puede hacer que las interfaces no puedan ser compatibles con una interfaz en común. En ese caso el ***'Adapter Pattern'*** es una buena solución.

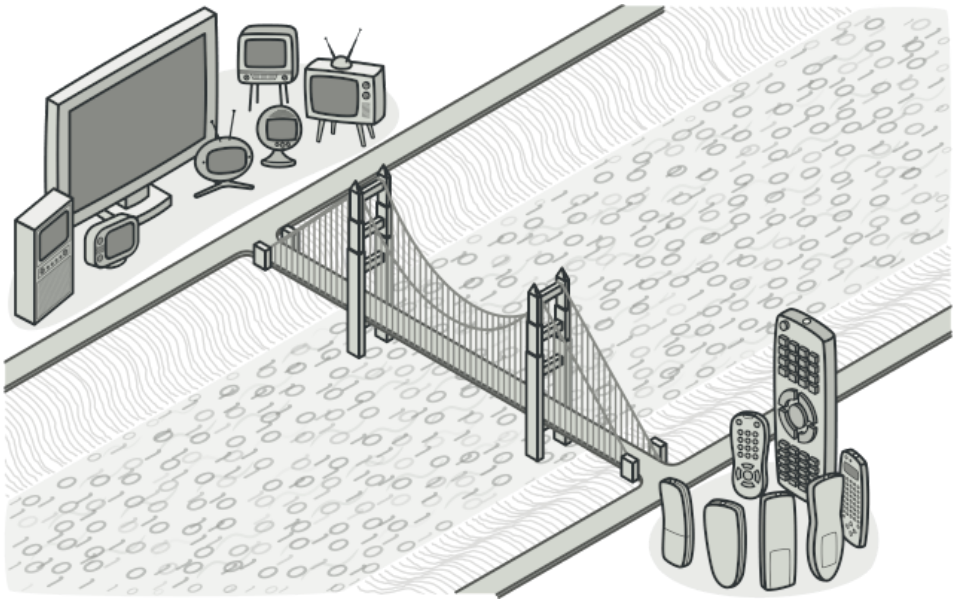
Además, un mismo adaptador puede usarse en varios sistemas.

Este patrón se parece al ***'Decorator Pattern'***. Sin embargo, difieren en que la finalidad de éste es proporcionar una interfaz completa del objeto adaptador, mientras que el decorador pueden centrarse en una sola parte.

Referencias

- https://refactoring.guru/es/design-patterns/adapter
- https://sourcemaking.com/design_patterns/adapter
- https://www.baeldung.com/java-adapter-pattern
- https://www.tutorialspoint.com/design_pattern/adapter_pattern
- https://www.digitalocean.com/community/tutorials/adapter-design-pattern-java
- https://java-design-patterns.com/es/patterns/adapter/

"Bridge Pattern"



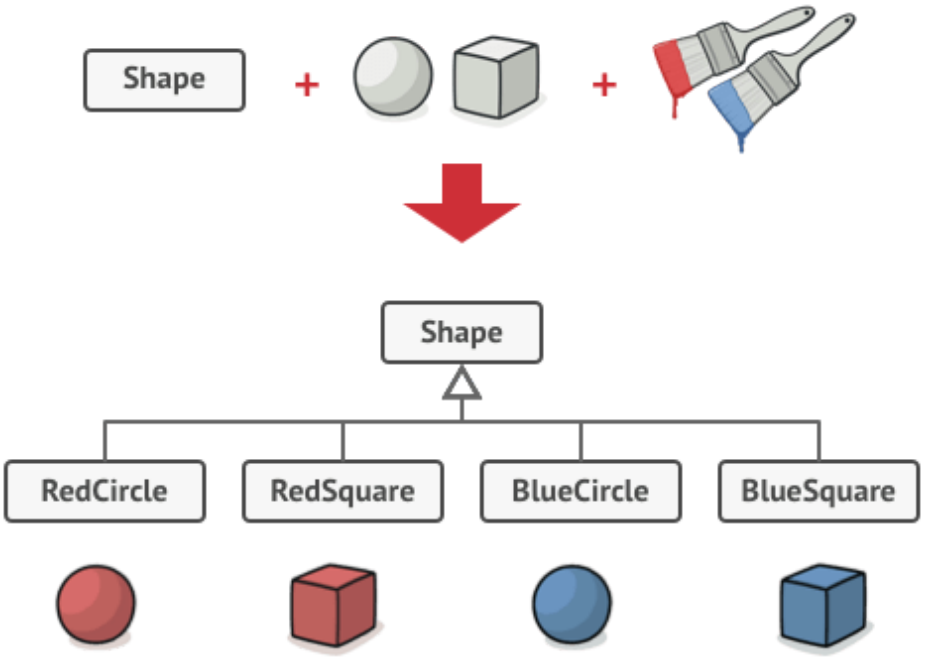
© Refactoring Guru

> *Desacoplar una abstracción de su implementación para que ambos puedan variar independientemente.*
>
> *-- GoF*

Concepto

Este patrón es una técnica usada en programación para **desacoplar una abstracción de su implementación**, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra. Esto es, se desacopla una abstracción de su implementación para que puedan variar independientemente.

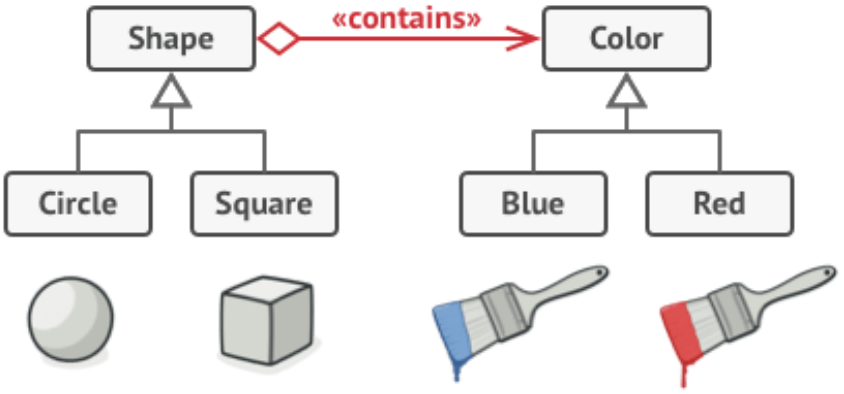
Digamos que tenemos la clase `Shape` con un par de subclases: `Circle` y `Square`. Si incorporamos el color a la jerarquía de clases debemos combinar las subclases con el nuevo requisito creando subclases como `BlueCircle` o `RedSquare`. Si en un futuro es necesario incorporar una nueva forma o un nuevo color, deberemos combinar esta nueva subclase o color con las formas o colores ya existentes, aumentando exponencialmente el número de subclases.



© Refactoring Guru

Este problema ocurre porque estamos tratando de extender las clases de formas en dos dimensiones independientes: por forma y por color. Ese es un problema muy común con la herencia de clases.

El ***'Bridge Pattern'*** intenta resolver este problema cambiando de herencia a composición. Lo que esto significa es que extrae una de las dimensiones en una jerarquía de clases separada, de modo que las clases originales hagan referencia a un objeto de la nueva jerarquía, en lugar de tener todos sus estados y comportamientos dentro de una clase.



© Refactoring Guru

En la definición de **GoF** se introducen los conceptos de *"abstración"* e *"implementación"*. La abstracción y la implementación se pueden representar a través de una interfaz o una clase abstracta, pero la abstracción contiene una referencia a su implementador. Normalmente, un hijo de una abstracción se llama una *'refined abstraction'* y un hijo de una implementación se llama *'concrete implementation'*.

- ***Abstraction*** define una interfaz abstracta. Mantiene una referencia a un objeto de tipo *'Implementor'*.
- ***RefinedAbstraction*** extiende la interfaz definida por *'Abstraction'*.
- ***Implementor*** define la interfaz para la implementación de clases. Esta interfaz no se tiene que corresponder exactamente con la interfaz de *'Abstraction'*; de hecho, las dos interfaces pueden ser bastante diferente. Típicamente la interfaz *'Implementor'* provee sólo operaciones primitivas, y *'Abstraction'* define operaciones de alto nivel basadas en estas primitivas.
- ***ConcreteImplementor*** implementa la interfaz de Implementor y define su implementación concreta.

Este patrón mejora la extensibilidad ya que permite que se puedan extender las jerarquías de clases de forma independientemente, sin afectarse entre sí.

Esto se debe a que se desacopla la interfaz de la implementación. Una implementación no es limitada permanentemente a una interfaz. La implementación de una abstracción puede ser configurada en tiempo de ejecución. Además le es posible a un objeto cambiar su implementación en tiempo de ejecución.

Desacoplando *'Abstraction'* e *'Implementor'* también elimina las dependencias sobre la implementación en tiempo de compilación. Cambiar una clase de implementación no requiere recompilar la clase *'Abstraction'* ni sus clientes. Es más, este desacoplamiento fomenta las capas, que pueden conducir a un sistema mejor estructurado. La parte de alto nivel de un sistema sólo tiene que conocer *'Abstraction'* e *'Implementor'*.

Caso práctico

1. Identificar las dimensiones independientes en tus clases. Estos conceptos independientes podrían ser: abstracción/plataforma, dominio/infraestructura, front-end/back-end o interfaz/implementación.

2. Ver qué operaciones necesita el cliente y definirlas en la clase de abstracción base.

3. Determinar las operaciones disponibles en todas las plataformas. Declare los que necesita la abstracción en la interfaz de implementación general.

4. Para todas las plataformas en su dominio, cree clases de implementación concretas, pero asegúrese de que todas sigan la interfaz de implementación.

5. Dentro de la clase de abstracción, agregue un campo de referencia para el tipo de implementación. La abstracción delega la mayor parte del trabajo al objeto de implementación al que se hace referencia en ese campo.

6. Si tiene varias variantes de lógica de alto nivel, cree abstracciones refinadas para cada variante extendiendo la clase de abstracción base.

7. El código del cliente debe pasar un objeto de implementación al constructor de la abstracción para asociar uno con el otro. Después de eso, el cliente puede olvidarse de la implementación y trabajar solo con el objeto de abstracción.

```
interface Abstraction {
    void operation();
}

interface Implementor {
    void operation();
}

class RefinedAbstraction implements Abstraction {
    private Implementor implementor;

    RefinedAbstraction(Implementor implementor) {
        this.implementor = implementor;
    }

    public void operation() {
        implementor.operation();
    }
}

class ImplementorA implements Implementor {
    @Override
    public void operation() {
        // ...
    }
}

class ImplementorB implements Implementor {
    @Override
    public void operation() {
        // ...
    }
}
```

Consideraciones

Utilice el ***'Bridge Pattern'*** cuando desee dividir y organizar una clase monolítica que tenga varias variantes de alguna funcionalidad (por ejemplo, si la clase puede trabajar con varios servidores de bases de datos). Este patrón le permite dividir la clase monolítica en varias jerarquías de clase. Después de esto, puede cambiar las clases en cada jerarquía independientemente de las clases en las otras. Este enfoque simplifica el mantenimiento del código y minimiza el riesgo de romper el código existente.

Use el patrón cuando necesite extender una clase en varias dimensiones ortogonales (independientes). El patrón sugiere que extraiga una jerarquía de clases separada para cada una de las dimensiones. La clase original delega el trabajo relacionado a los objetos que pertenecen a esas jerarquías en lugar de hacer todo por sí solo

También cuando se necesite poder cambiar de implementación en tiempo de ejecución. Aunque es opcional, este patrón permite reemplazar el objeto de implementación dentro de la abstracción. Es tan fácil como asignar un nuevo valor a un campo.

El uso de este patrón tiene ciertas ventajas:

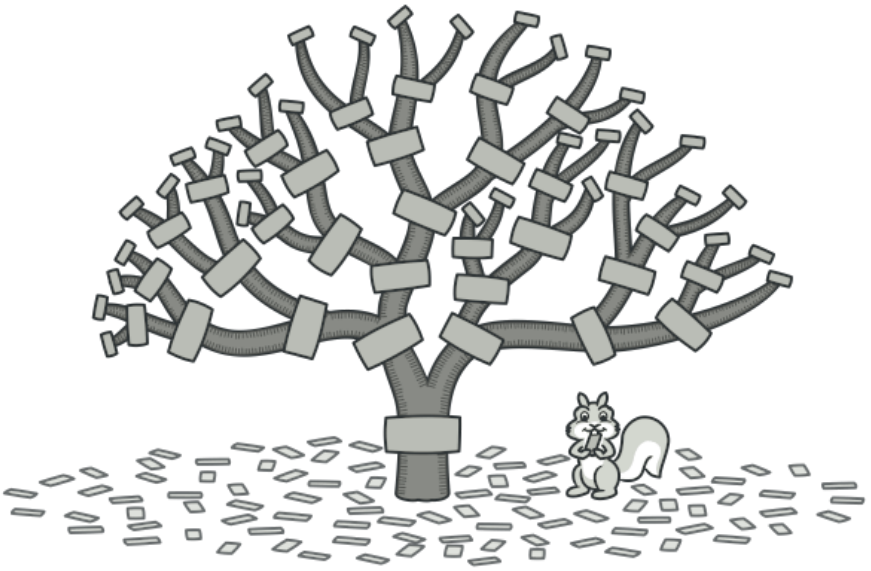
El código del cliente funciona con abstracciones de alto nivel y no está expuesto a los detalles de la plataforma.

La desventaja de este patrón es que puede hacer que el código sea más complicado aplicando el patrón a una clase altamente cohesiva.

Referencias

- https://refactoring.guru/es/design-patterns/bridge
- https://sourcemaking.com/design_patterns/bridge
- https://www.baeldung.com/java-bridge-pattern
- https://www.tutorialspoint.com/design_pattern/bridge_pattern
- https://www.digitalocean.com/community/tutorials/bridge-design-pattern-java
- https://java-design-patterns.com/es/patterns/bridge/

"Composite Pattern"



© Refactoring Guru

> *Componer objetos en estructuras de árbol para representar jerarquías parciales. Este patrón permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme.*
>
> *-- GoF*

Concepto

El ***'Composite Pattern'*** sirve para construir objetos complejos a partir de otros más simples y similares entre sí gracias a la composición recursiva y a una estructura en forma de árbol.

Esto simplifica el tratamiento de los objetos creados ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera. Un cliente puede tratar un objeto compuesto como si fuera un solo objeto.

Una buena manera de identificar la situación en que se puede aplicar este patrón es cuando tengo "un X y tiene varios objetos X".

Caso práctico

Para ilustrar el problema supóngase un juego de estrategia en el que los jugadores pueden recoger objetos o items, los cuales tienen una serie de propiedades como "precio", "descripción", etc. Cada item, a su vez, puede contener otros items. Por ejemplo, un bolso de cuero puede contener una pequeña caja de madera que, a su vez, contiene un pequeño reloj dorado, etc..

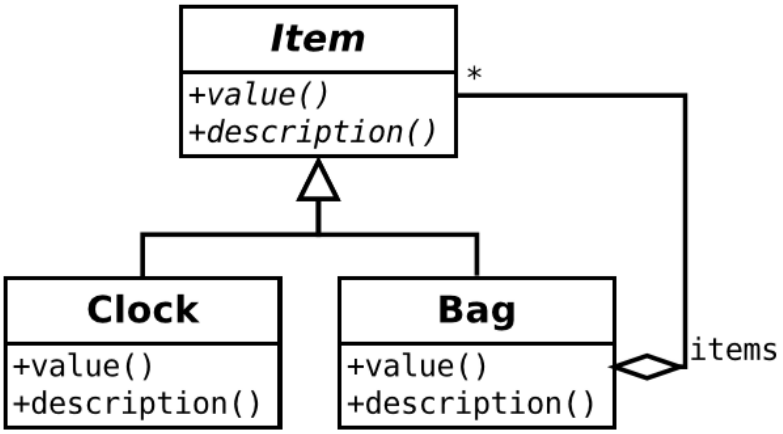


Figura 4.6: Ejemplo de aplicación del patrón Composite

En definitiva, el ***'Composite Pattern'*** habla sobre cómo diseñar estructuras recursivas donde la composición homogénea de objetos recuerda a una estructura arbórea.

Naturalmente, los objetos compuestos suelen ofrecer también operaciones para añadir, eliminar y actualizar.

```
class Item {
    void value() {
        // ...
    }

    void description() {
        // ...
    }
}

class Clock extends Item {
    @Override
    void value() {
        super.value();
    }

    @Override
    void description() {
        super.description();
    }
}

class Bag extends Item {
    private List<Item> bag = new ArrayList<>();

    void addItem(Item item) {
        bag.add(item);
    }

    boolean removeItem(Item item) {
```

```
            return bag.contains(item) && bag.remove(item);
        }
    }
```

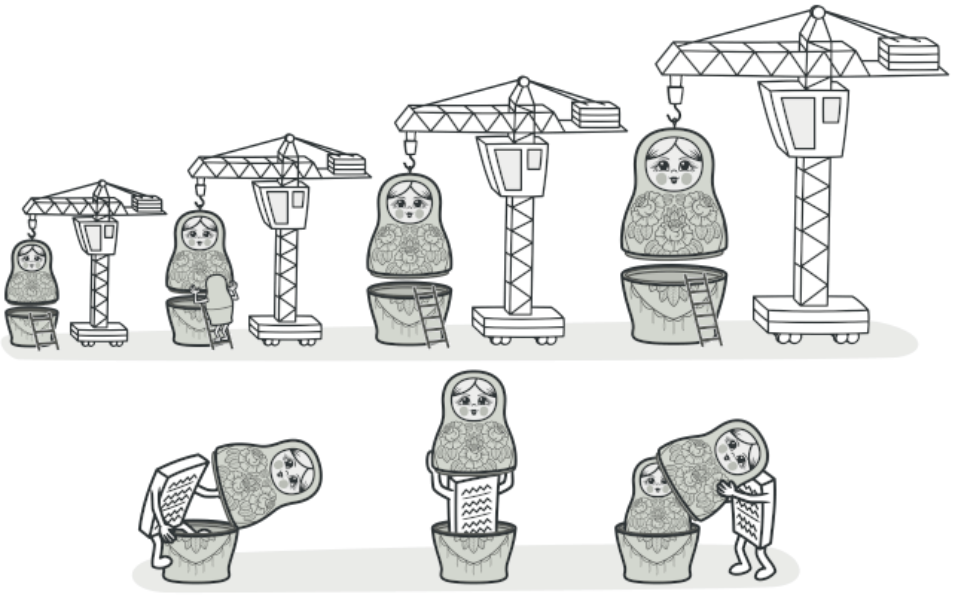
Consideraciones

Los clientes pueden añadir nuevos tipos de componentes fácilmente.

Referencias

- https://refactoring.guru/es/design-patterns/composite
- https://sourcemaking.com/design_patterns/composite
- https://www.baeldung.com/java-composite-pattern
- https://www.tutorialspoint.com/design_pattern/composite_pattern
- https://www.digitalocean.com/community/tutorials/composite-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/composite/

"Decorator Pattern"



© Refactoring Guru

> *Asignar responsabilidades adicionales a un objeto de forma dinámica. Los decoradores ofrecen una alternativa flexible a la subclasificación para ampliar la funcionalidad.*
>
> *-- GoF*

Concepto

El ***'Decorator Pattern'*** sirve para **añadir y/o modificar la responsabilidad, funcionalidad o propiedades de un objeto en tiempo de ejecución**. Permite organizar el diseño de forma que la incorporación de nueva funcionalidad en tiempo de ejecución sea transparente desde el punto de vista del usuario de la **clase decorada**.

Este patrón permite tener una jerarquía de clases compuestas, formando una estructura más dinámica y flexible que la herencia estática. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Además, evita la utilización de la herencia con muchas clases y también, la herencia múltiple (si fuera posible en el lenguaje utilizado).

Caso práctico

Supongamos que el personaje de un videojuego porta un arma que utiliza para eliminar a sus enemigos. Dicha arma, por ser de un tipo determinado, tiene una serie de propiedades como podrían ser "radio de acción", "número de balas", etc.. Sin embargo, es posible que el personaje incorpore elementos al arma que pueden cambiar estas propiedades como un silenciador o un cargador extra.

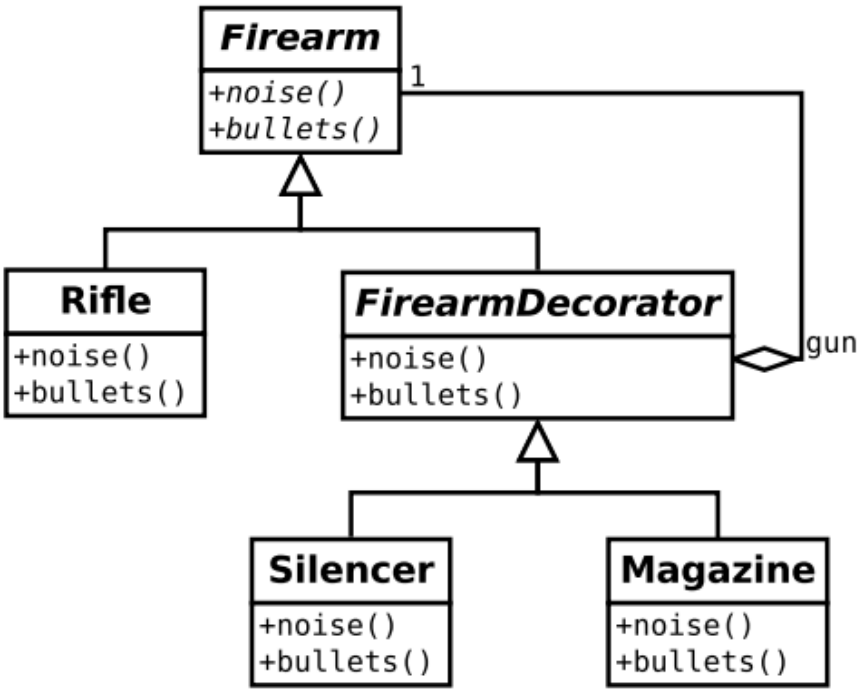


Figura 4.7: Ejemplo de aplicación del patrón Decorator

Básicamente, los diferentes tipos de armas de fuego implementan una clase abstracta llamada `Firearm`. Una de sus subclases es `FirearmDecorator` que a su vez es la superclase de todos los componentes que *"decoran"* a un objeto `Firearm`. Nótese que este decorador implementa la interfaz propuesta por `Firearm` y está compuesto por un objeto *'gun'*, el cual decora.

```
class Firearm {
    private static final int MAX_BULLETS = 5;
    private static final float GENERIC_NOISE = 150.0f;

    float noise() {
        return GENERIC_NOISE;
    }

    int bullets() {
        return MAX_BULLETS;
    }
}

class Rifle extends Firearm {
    @Override
    float noise() {
        return super.noise();
    }

    @Override
    int bullets() {
        return super.bullets();
    }
}

class FirearmDecorator extends Firearm {
```

```
    private Firearm gun;

    FirearmDecorator(Firearm gun) {
        this.gun = gun;
    }

    @Override
    float noise() {
        return gun.noise();
    }

    @Override
    int bullets() {
        return gun.bullets();
    }
}

class Silencer extends FirearmDecorator {
    Silencer(Firearm gun) {
        super(gun);
    }

    @Override
    float noise() {
        return super.noise() - 55;
    }

    @Override
    int bullets() {
        return super.bullets();
    }
}

class Magazine extends FirearmDecorator {
    Magazine(Firearm gun) {
        super(gun);
    }

    @Override
    float noise() {
        return super.noise();
    }

    @Override
    int bullets() {
        return super.bullets() + 5;
    }
}
```

Consideraciones

El ***'Decorator Pattern'*** está más centrado en la extensión de la funcionalidad que en la composición de objetos para la generación de una jerarquía como ocurre con el ***'Composite Pattern'***.

Normalmente, sólo existe una objeto decorado y no un vector de objetos (aunque también es posible).

Este patrón puede generar gran cantidad de objetos pequeños y parecidos que dificulta su identificación.

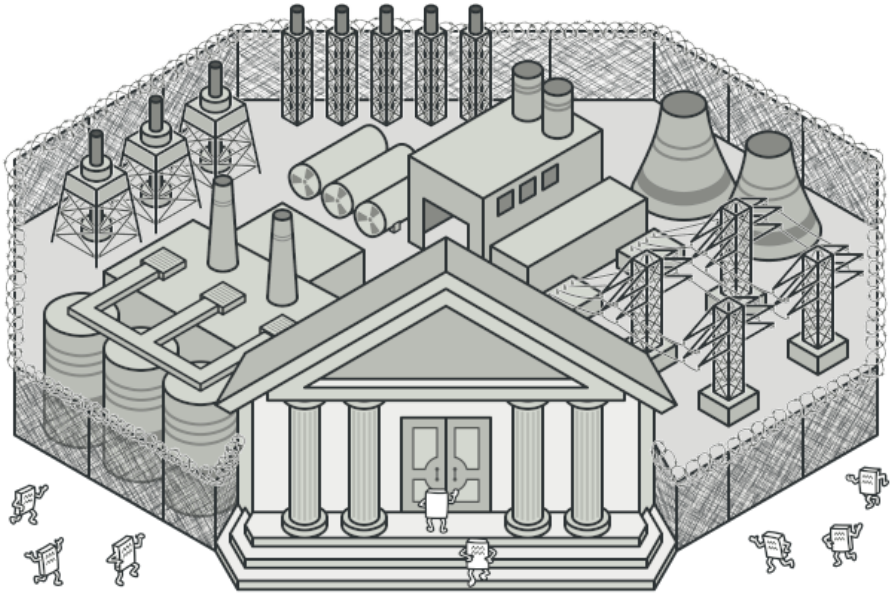
Este patrón se diferencia de la simple herencia en que podemos añadir o quitar responsabilidades simplemente adjuntando o quitando decoradores. Pero únicamente con la herencia, necesitamos crear una nueva clase para nuevas responsabilidades. Por lo tanto, habrá muchas clases dentro del sistema, lo que a su vez puede hacer que el sistema sea complejo.

Referencias

- https://refactoring.guru/es/design-patterns/decorator

- https://sourcemaking.com/design_patterns/adapter
- https://www.baeldung.com/java-decorator-pattern
- https://www.tutorialspoint.com/design_pattern/decorator_pattern
- https://www.digitalocean.com/community/tutorials/decorator-design-pattern-in-java-example
- https://java-design-patterns.com/es/patterns/decorator/

"Facade Pattern"



© Refactoring Guru

> *Proporcionar una interfaz unificada a un conjunto de interfaces en un sistema. Este patrón define una interfaz de nivel superior que facilita el uso del subsistema.*
>
> *-- GoF*

Concepto

El ***'Facade Pattern'*** eleva el nivel de abstracción de un determinado sistema para ocultar ciertos detalles de implementación y hacer más sencillo su uso.

Este patrón se aplica para proveer de una interfaz unificada simple que permita acceder a una interfaz o grupo de interfaces de un sistema complejo o cuando se quiera estructurar varios subsistemas en capas ya que las fachadas serían el punto de entrada a cada nivel.

Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable.

También es útil cuando haya que controlar el acceso y la forma en que se utiliza un sistema determinado.

Este patrón se relaciona con el ***'Singleton Pattern'*** ya que normalmente las fachadas suelen ser instancias únicas.

Caso práctico

Entidades de este patrón:

- **Fachada** - conoce qué clases del subsistema son responsables de una determinada peticíón y delega esas peticiones de los clientes a los objetos apropiados del subsistema. Los clientes no acceden directamente a las subclases.

- **Subclases** (ModuleA, ModuleB, ModuleC…) - implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la fachada. No conocen la existencia de la fachada.

```
class ModuleA {
    String getInfo() {
        return ModuleA.class.getSimpleName();
    }
}

class ModuleB {
    String getInfo() {
        return ModuleB.class.getSimpleName();
    }
}

class ModuleC {
    String getInfo() {
        return ModuleC.class.getSimpleName();
    }
}

class Facade {
    private ModuleA moduleA;
    private ModuleB moduleB;
    private ModuleC moduleC;

    Facade() {
        moduleA = new ModuleA();
        moduleB = new ModuleB();
        moduleC = new ModuleC();
    }

    String getInfoA() {
        return moduleA.getInfo();
    }

    String getInfoB() {
        return moduleB.getInfo();
    }

    String getInfoC() {
        return moduleC.getInfo();
    }
}
```

Consideraciones

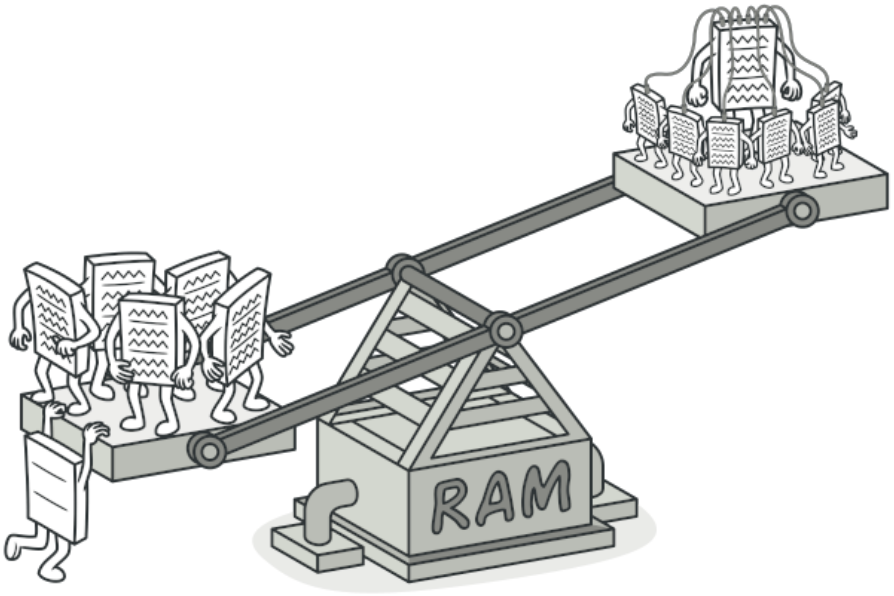
La principal ventaja al utilizar este patrón consiste en que para modificar las clases del subsistema sólo hay que realizar los cambios en la interfaz/fachada. Los clientes pueden permanecer ajenos a ello. Esto permite el desacople de los clientes y eleva el nivel de abstracción.

Un problema derivado del uso de este patrón es que si no se dividen correctamente las responsabilidades se crean clases fachada con un tamaño desproporcionado al incluir demasiada funcionalidad.

Referencias

- https://refactoring.guru/es/design-patterns/facade
- https://sourcemaking.com/design_patterns/facade
- https://www.baeldung.com/java-facade-pattern
- https://www.tutorialspoint.com/design_pattern/facade_pattern
- https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java
- https://java-design-patterns.com/es/patterns/facade/

"Flyweight Pattern"



© Refactoring Guru

> *Compartir una parte común del estado de un objeto para hacer más eficiente la gestión de un número elevado de objetos de grano más fino.*
>
> *-- GoF*

Concepto

El ***'Flyweight Pattern'*** (u objeto ligero) es un patrón estructural que sirve para eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos que contienen información idéntica, además de lograr un equilibrio entre flexibilidad y rendimiento (uso de recursos). Permite ajustar más objetos en la cantidad de RAM disponible al compartir partes comunes de estado entre múltiples objetos en lugar de mantener todos los datos en cada objeto.

El patrón es útil cuando se necesita una gran cantidad de objetos similares que en general la mayoría de atributos son comunes pero son únicos en términos de solo unos pocos parámetros. Con lo cual se intenta minimizar el uso de la memoria compartiendo los datos tanto como sea posible con los otros objetos similares. Compartir objetos puede permitir su uso a granularidades finas con costos mínimos.

Se utilizan dos términos comunes en este contexto: **"extrinsic"** e **"intrinsic"**. A grandes rasgos, se basa en dividir un objeto en dos partes: una parte “común” a un conjunto grande de los objetos de la clase (parte intrínseca), y una parte “privada” que será accesible y modificable únicamente por un objeto en concreto (parte extrínseca).

El ***'Flyweight Pattern'*** está íntimamente ligado al ***'Factory Pattern'***. El motivo no es otro que permitir que sea el objeto que implementa este patrón el que gestione la separación entre la parte “común” (denominada intrínseca) y la parte “privada” (denominada extrínseca), centralizando el proceso y evitando así que perdamos referencias por el camino si realizamos el proceso de una forma un poco más artesanal.

Por lo tanto, dentro de este patrón distinguiremos entre estos dos tipos de datos:

- **Intrínsecos**: son los datos compartidos por todos los objetos de un subtipo determinado. Por norma general, son datos que no cambiarán a lo largo del tiempo, y si cambian, alterarán el estado de todos los objetos que hagan uso de ellos.

- **Extrínsecos**: se calculan “al vuelo” fuera del objeto *"flyweight"*. Este cálculo suele realizarse a partir de los datos intrínsecos y de los parámetros recibidos por los métodos del objeto *"flyweight"*. La idea detrás de los datos extrínsecos radica en que, o bien sean calculados a partir de los datos intrínsecos o bien ocupen una cantidad de memoria mínima en comparación a éstos.

Caso práctico

1. Dividir los campos de una clase que se convertirá en *'flyweight'* en dos partes:

    i. **Estado intrínseco**: los campos que contienen datos invariables duplicados en muchos objetos.

    ii. **Estado extrínseco**: los campos que contienen datos contextuales únicos para cada objeto.

2. Dejar los campos que representan el estado intrínseco en la clase, pero asegurándose de que sean inmutables. Deben tomar sus valores iniciales solo dentro del constructor.

3. Repasar los métodos que utilizan campos del estado extrínseco. Para cada campo utilizado en el método, introducir un nuevo parámetro y utilizarlo en lugar del campo.

4. Opcionalmente, crear una clase *'Factory'* para administrar un conjunto o *'pool'* de objetos *'flyweight'*. Debe verificar si hay un objeto *'flyweight'* existente antes de crear uno nuevo. Una vez que la *'factory'* está en su lugar, los clientes solo deben solicitar los objetos *'flyweight'* a través de ella. Deben describir el *'flyweight'* deseado pasando su estado intrínseco a la *'factory'*.

```
interface Bird {
    void draw();
}

class AngryBird implements Bird {
    private String color;

    AngryBird(String color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.println("Angry Bird: Draw() [Color : " + color + "]");
    }
}

class BirdFactory {
    private static final HashMap<String, Bird> angrybirdMap = new HashMap<>();

    static Bird getAngryBird(String color) {
        AngryBird angrybird = (AngryBird) angrybirdMap.get(color);

        if (angrybird == null) {
            angrybird = new AngryBird(color);
            angrybirdMap.put(color, angrybird);
            System.out.println("Creating Angry Bird of color : " + color);
        }
        return angrybird;
    }
}
```

Consideraciones

Utilizar este patrón solo cuando el programa o código debe admitir una gran cantidad de objetos que apenas caben en la RAM disponible.

El beneficio de aplicar el patrón depende en gran medida de cómo y dónde se usa. Es más útil cuando una aplicación necesita generar una gran cantidad de objetos similares y esto drena toda la RAM disponible en el dispositivo de destino. Además de esto, los objetos contienen estados duplicados que pueden extraerse y compartirse entre múltiples objetos.

Este patrón tiene la ventaja de que puede guardar una gran cantidad de RAM, asumiendo que su programa tiene toneladas de objetos similares.

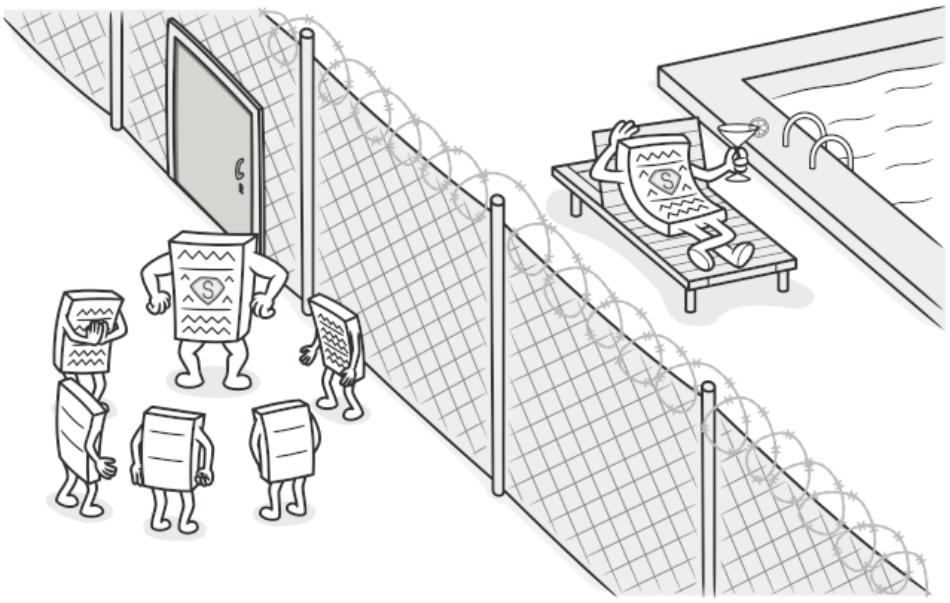
Como inconveniente, es posible que esté intercambiando RAM a través de ciclos de CPU cuando algunos de los datos de contexto se deben recalcular cada vez que alguien llama a un método *'flyweight'*.

Adenás, el código se vuelve mucho más complicado.

Referencias

- https://refactoring.guru/es/design-patterns/flyweight
- https://sourcemaking.com/design_patterns/flyweight
- https://www.baeldung.com/java-flyweight
- https://www.tutorialspoint.com/design_pattern/flyweight_pattern
- https://www.digitalocean.com/community/tutorials/flyweight-design-pattern-java
- https://java-design-patterns.com/es/patterns/flyweight/

"Proxy Pattern"



© Refactoring Guru

> *Proporcione un sustituto o marcador de posición para que otro objeto controle el acceso a él.*
>
> *-- GoF*

Concepto

El ***'Proxy Pattern'*** tiene como propósito proporcionar un **sustituto o intermediario de un objeto para controlar su acceso**. Proporciona mecanismos de abstracción y control para acceder a un determinado objeto "simulando" que se trata del objeto real.

Muchos de los objetos que pueden integrar una aplicación pueden representar diferentes problemas a la hora de ser utilizados por la aplicación:

- **Coste computacional** - Es posible que un objeto sea costoso de manipular y cargar.
- **Acceso remoto** - el acceso por red es un componente cada vez más común.
- **Acceso seguro** - es posible que para usar determinados objetos se necesiten determinados privilegios.

- **Dobles de prueba -** a la hora de diseñar y probar el código se necesitan objetos *"dobles"* que simulen a los objetos reales.

Caso práctico

El coste de manipular y cargar una imagen puede ser importante según el sistema, tipo de manipulación, etc… Con el patrón ***'Proxy'*** se crea un objeto intermedio `ImageProxy` que represente al objeto real `Image` y que se utilice de la misma forma desde el punto de vista del cliente.

De esta forma, el objeto `Proxy` puede cargar una única vez la imagen y mostrarla tantas veces como el cliente lo solicite.

```
class Graphic {
    String display() {
        return "Graphic";
    }
}

class Image extends Graphic {
    @Override
    String display() {
        return "Display image";
    }

    void load() {
        // ...
    }
}

class ImageProxy extends Graphic {
    private Image image;

    @Override
    String display() {
        if (image == null) {
            image = new Image();
            image.load();
        }
        return image.display();
    }
}
```

Consideraciones

En los sistemas de autenticación, dependiendo de las credenciales presentadas por el cliente, se devuelve un proxy u otro que permiten realizar distintos tipos de operaciones, acceder a determinados recursos, etc…

Referencias

- https://refactoring.guru/es/design-patterns/proxy
- https://sourcemaking.com/design_patterns/proxy
- https://www.baeldung.com/java-proxy-pattern
- https://www.tutorialspoint.com/design_pattern/proxy_pattern
- https://www.digitalocean.com/community/tutorials/proxy-design-pattern
- https://java-design-patterns.com/es/patterns/proxy/

Otros patrones estructurales

"Reactor Pattern"

El ***'Reactor Pattern'*** es un patrón de programación concurrente para manejar los pedidos de servicio entregados de forma concurrente a un manejador de servicio desde una o más entradas. El manejador de servicio demultiplexa los pedidos entrantes y los entrega de forma sincrónica a los manejadores de pedidos asociados.

Concepto

El ***'Reactor Pattern'*** es un patrón arquitectural para resolver el problema de cómo atender peticiones concurrentes a través de señales y manejadores de señales.

Existen aplicaciones, como los servidores web, cuyo comportamiento es *reactivo*, es decir, a partir de la ocurrencia de un evento externo se realizan todas las operaciones necesarias para atender a ese evento externo. En el caso del servidor web, una conexión entrante **(evento)** dispararía la ejecución del código pertinente que crearía un hilo de ejecución para atender a dicha conexión. Pero también pueden tener comportamiento *proactivo*. Por ejemplo, una señal interna puede indicar cuándo destruir una conexión con un cliente que lleva demasiado tiempo sin estar accesible.

En los videojuegos ocurre algo muy similar: diferentes entidades pueden lanzar eventos que deben ser tratados en el momento en el que se producen. Por ejemplo, la pulsación de un botón en el joystick de un jugador es un evento que debe ejecutar el código pertinente para que la acción tenga efecto en el juego.

El patrón ***'Reactor'*** se compone de **eventos**, **manejadores de eventos**, **recursos** y el **reactor** propiamente dicho:

- **Eventos** - los eventos externos que pueden ocurrir sobre los recursos **(Handles)**. Normalmente su ocurrencia es asíncrona y siempre está relacionada a un recurso determinado.
- **Recursos (Handlers)** - se refiere a los objetos sobre los que ocurren los eventos.
- **Manejadores de Eventos** - Asociados a los recursos y a los eventos que se producen en ellos, se encuentran los manejadores de eventos **(EventHandler)** que reciben una invocación a través del método **handle()** con la información del evento que se ha producido.
- **Reactor** - se trata de la clase que encapsula todo el comportamiento relativo a la desmultiplexación de los eventos en manejadores de eventos **(dispatching)**. Cuando ocurre cierto evento, se busca los manejadores asociados y se les invoca el método **handle()**.

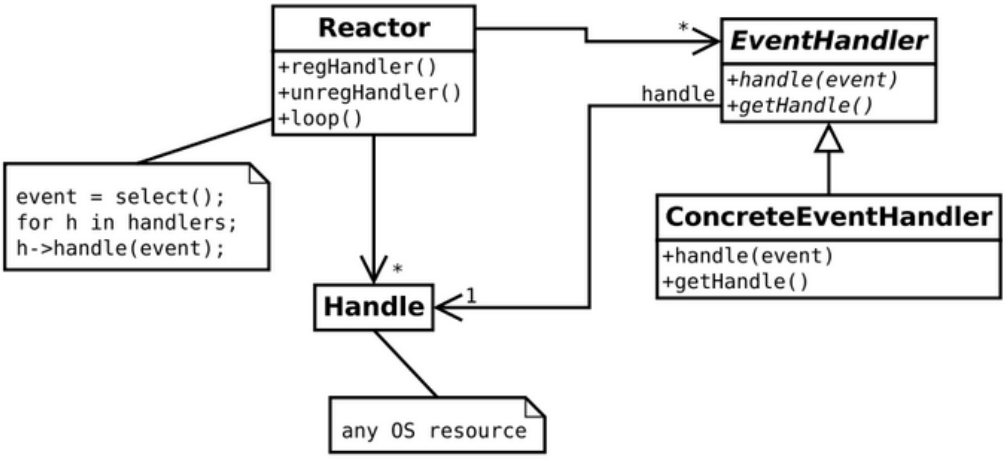


Figura 4.18: Diagrama de clases del patrón Reactor

Caso práctico

```java
interface EventHandler {
    void handle(Event event);
}

class ConcreteEventHandler implements EventHandler {
    @Override
    public void handle(Event event) {
        event.getInfo();
    }
}

class Event {
    void getInfo() {
        System.out.println("Event occurs");
    }
}

public class Reactor {
    private static List<EventHandler> registeredHandlers = new ArrayList<>();

    private static void registerHandler(EventHandler eventHandler) {
        registeredHandlers.add(eventHandler);
    }

    public static void main(String[] args) {
        // Handler registration
        EventHandler eventHandler = new ConcreteEventHandler();
        registerHandler(eventHandler);
        while (true) {
            // Simulate event
            if (new Random().nextInt(10000000) == 50) {
                Event event = new Event();

                for (EventHandler handler : registeredHandlers) {
                    handler.handle(event);
                }
            }
        }
    }
}
```

Comportamiento

1. Los manejadores se registran usando el método `regHandler()` del **reactor**. De esta forma, el Reactor puede configurarse para esperar los eventos del recurso que el manejador espera. El manejador puede dejar de recibir notificaciones con `unregHandler()`.

2. A continuación, el **reactor** entra en el bucle infinito `loop()`, en el que se espera la ocurrencia de eventos.

3. Utilizando alguna llamada al sistema, como puede ser `select()`, el **reactor** espera a que se produzca algún evento sobre los recursos monitorizados.

4. Cuando ocurre, busca los manejadores asociados a ese recurso e invoca el método `handle()` de cada manejador con el evento que ha ocurrido como parámetro.

5. El manejador recibe la invocación y ejecuta todo el código asociado al evento.

Consideraciones

Aunque los eventos puede ocurrir de forma concurrente, el **reactor** serializa las llamadas a los manejadores. Por lo tanto, la ejecución de los manejadores de eventos ocurre de forma **secuencial**.

Para evitar que el sistema quede inoperable, los manejadores de eventos no pueden consumir mucho tiempo. En general, cuanto mayor sea la frecuencia en que ocurren los eventos, menos tiempo deben consumir los manejadores.

Desde un punto de vista general, el ***'Reactor Pattern'*** tiene un comportamiento muy parecido al ***'Observer Pattern'***. Sin embargo, este patrón está pensado para las relaciones **1 a 1** mientras que en el caso del ***'Observer Pattern'*** se producen relaciones **1 a n**.

Referencias

- https://www.baeldung.com/concurrency-principles-patterns
- https://es.wikipedia.org/wiki/Reactor_%28patr%C3%B3n_de_dise%C3%B1o%29

"MVC Pattern"

Concepto

El patrón ***'MVC'*** separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones.

Para ello el patrón ***'MVC'*** propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

Caso práctico

- **Vista** - se trata de la interfaz de usuario que interactúa con el usuario y recibe las órdenes (pulsar un botón, introducir texto, etc…). También recibe órdenes desde el controlador para mostrar información o realizar un cambio en la interfaz.

- **Controlador** - el controlador recibe órdenes utilizando, habitualmente, manejadores o *'callbacks'* y traduce esa acción al dominio del modelo de la aplicación. La acción puede ser crear una nueva instancia de un objeto determinado, actualizar estados, pedir operaciones al modelo, etc…

- **Modelo** - el modelo de la aplicación recibe las acciones a realizar por el usuario, pero ya independientes del tipo de interfaz utilizado porque se utilizan, únicamente, estructuras propias del dominio del modelo y llamadas desde el controlador.

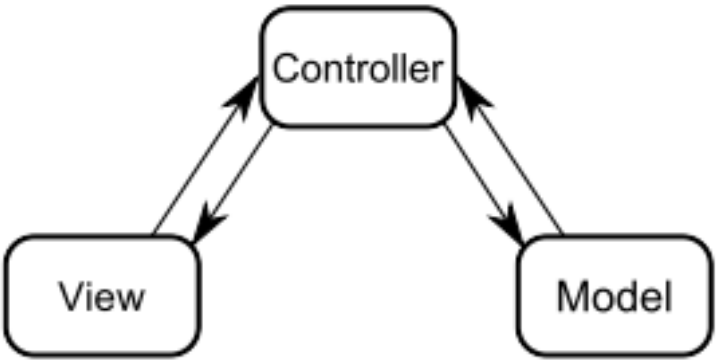


Figura 4.9: Estructura del patrón MVC.

Normalmente, la mayoría de las acciones que realiza el controlador sobre el modelo son operaciones de consulta de su estado para que pueda ser convenientemente representado por la vista.

El patrón ***'MVC'*** no es un patrón con una separación rígida. Es posible encontrar implementaciones en las que, por ejemplo, el modelo notifique directamente a las interfaces de forma asíncrona eventos producidos en sus estructuras y que deber ser representados en la vista (siempre y cuando exista una aceptable independencia entre capas). Para ello es de gran utilidad el patrón ***'Observer'***.

El patrón ***'MVC'*** se utiliza en un gran número de entornos de ventanas y sobretodo en entornos web.

Sin embargo, es la estructura más utilizada en los videojuegos: la interfaz gráfica utilizando gráficos 2D/3D (vista), bucle de eventos (controlador) y las estructuras de datos internas (modelo).

Referencias

- https://www.baeldung.com/mvc-vs-mvp-pattern
- https://www.digitalocean.com/community/tutorials/what-is-mvc

---

Licencia