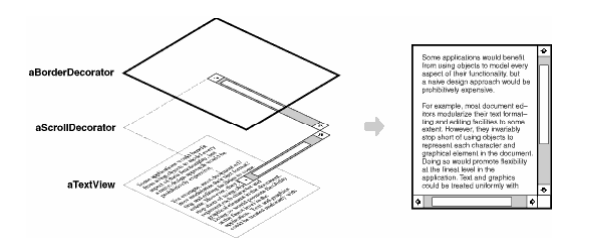
# (En este documento pone tu info)

# Decorator

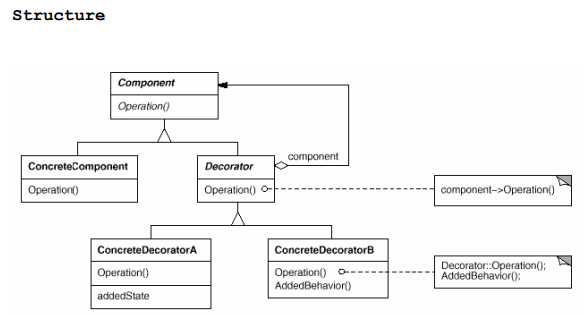
La intención de este patrón de diseño es enganchar diferentes responsabilidades adicionales a un objeto dinámico. Los decoradores proveen una manera alternativa de subclases para extender una funcionalidad.

A veces queremos agregar responsabilidades a objetos individuales, no a la clase entera. Si bien esto se podría hacer con herencia, hay un punto en el que esto ya no es flexible

Un acercamiento más flexible, es encerrar el componente en otro objeto que añada el borde. El objeto que encierra es llamado **decorador**. El decorador conforme a la interfaz del componente decora así que su presencia es transparente al componente del cliente.

Por ejemplo, supongamos que tengamos un objeto TextView que muestra el texto en una ventana. El textview no tiene scroll bars por defecto, porque no necesariamente siempre las necesitaremos. Cuando lo hacemos, podemos usar un ScrollDecorator para añadirlo. Supongamos que tambien queremos añadir un negro borde grueso alrededor del TextView. Podemos usar un BorderDecorator para añadir esto también.

Aplicabilidad

* Para agregar responsabilidades a objetos individuales dinámica y transparentemente, eso es, sin afectar a otros objetos.
* para responsabilidades que pueden ser quitadas
* Cuando la extensión de una subclase es impráctica.

Consecuencias

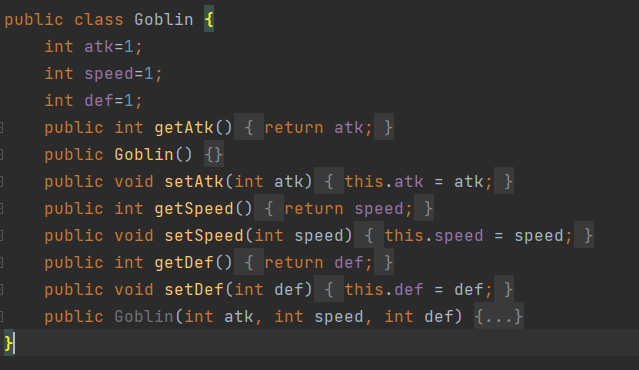
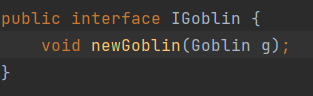
1. Mayor flexibilidad que una herencia estática y los Decorator también hacen más fácil agregar una propiedad dos veces.
2. Evita las clases cargadas de características en los niveles más altos de la jerarquía. Los Decorator ofrecen un acercamiento paga-asi-como-vas para añadir responsabilidades.
3. Un decorador y su componente no son idénticos. un decorador actúa como un encapsulamiento transparente. Pero desde un punto de vista de identidad de objeto, un componente de decoración no es idéntico al componente mismo. Por eso no deberías confiar en la identidad del objeto al usar el decorador.
4. Muchos objetos pequeños. Un diseño que usa decoradores normalmente resulta en sistemas compuestos por objetos pequeños que todos se parecen. Los objetos difieren únicamente en la manera en la que ellos están interconectados, no en su clase o en su valor de sus variables. Aunque estos sistemas son fáciles de modificar para aquellos quienes los entienden, pueden ser difíciles de leer y debuggear.

Implementación.

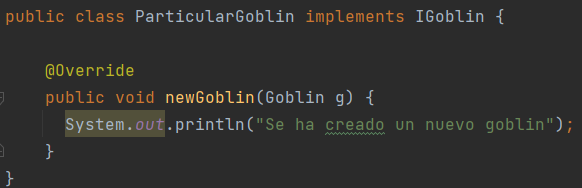
1. Conformidad de la interfaz: Una interfaz de objeto Decorator deberá ajustarse a la interfaz del componente que la decora. Las clases ConcreteDecorator deberán entonces heredar de una clase común.
2. Omitiendo la clase abstracta Decorator. No hay necesidad de definir una clase abstracta Decorator cuando tu solo debes añadir una responsabilidad. Usualmente ese es el caso cuando estás lidiando con la jerarquía de clases existente en lugar de cuando estás diseñando una nueva. En ese caso, puedes fusionar la responsabilidad Decorators para reenviar solicitudes al componente en el ConcreteDecorator.
3. Mantener los componentes de clases con peso ligero. Para asegurar una interfaz conforme, componentes y decoradores deberán descender de una clase Component común. Es importante mantener esta clase común ligera; eso es, que deberá enfocarse en definir e interfaz. no en almacenar datos. La definición de la representación de datos podrá ser diferida a las subclases; de otra manera la complejidad de la clase Component podrá hacer a los decoradores demasiado pesados de usar en cantidad. Poniendo muchas de las funcionalidades en componentes también incremente la probabilidad de concretar subclases que pagarán las características que ellos no necesitaran.

Demo

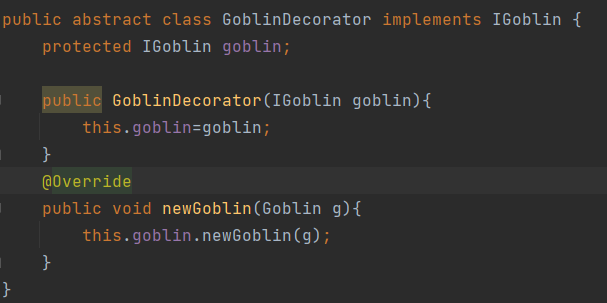
Interfaz y Clase base



Hijo de clase



Decorators



# Patrón de Estado

State es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiará su clase.

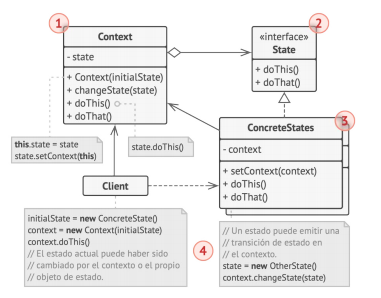
La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número finito de estados. Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados. Estas normas de cambio llamadas transiciones también son finitas y predeterminadas.

A diferencia de procesar estos cambios de estado con una estructura de toma de decisiones, habrá un punto en el que este sea muy complejo, y será necesario aplicar un patrón que nos ayude a poner un orden y estructurar el proyecto de una manera más comprensible.

El problema tiende a empeorar con la evolución del proyecto. Es bastante difícil predecir todos los estados y transiciones posibles en la etapa de diseño. Por ello, una máquina de estados esbelta, creada con un grupo limitado de condicionales, puede crecer hasta convertirse en un abotargado desastre con el tiempo.

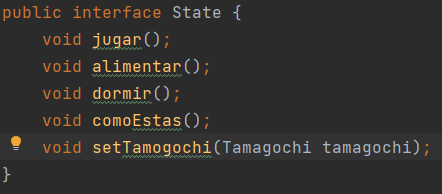
El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases. En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado contexto, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto

**Estructura**

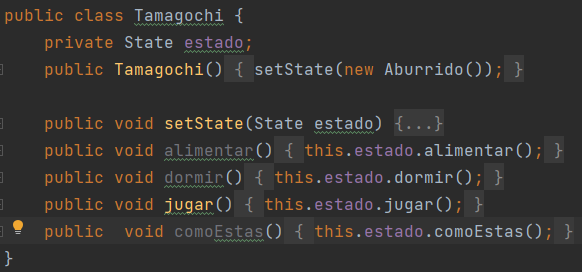


1. La clase Contexto almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (setter) para pasarle un nuevo objeto de estado.
2. La interfaz Estado declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.
3. Los Estados Concretos proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, pueden incluir clases abstractas intermedias que encapsulan algún comportamiento común. Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.
4. Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto

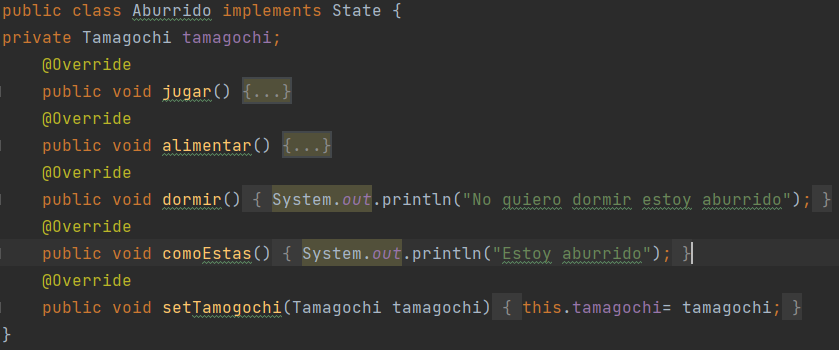
Demo

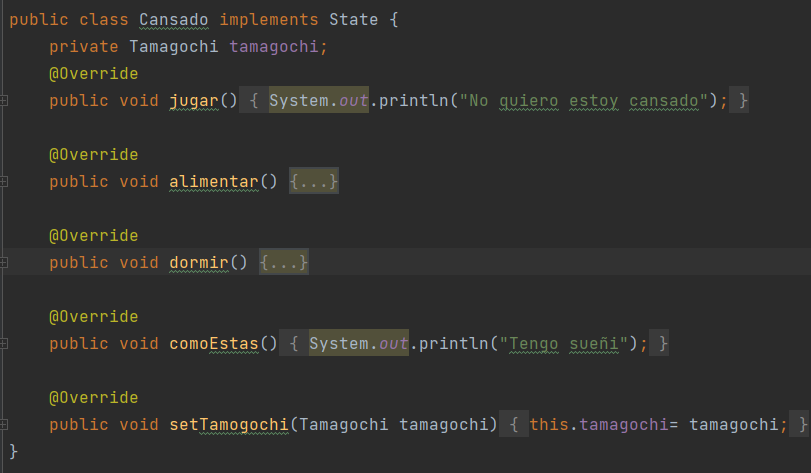


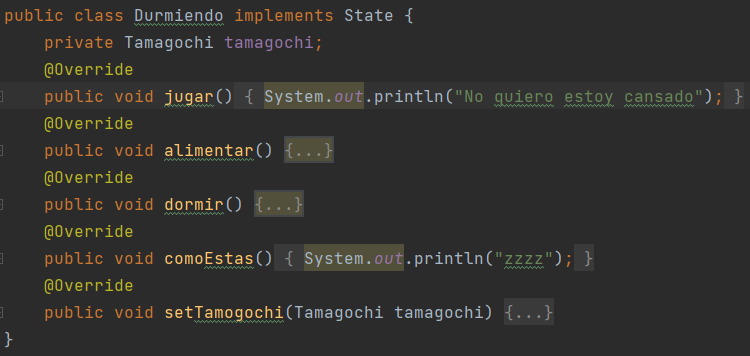
Interfaz con los diferentes estados que nos interesa tratar

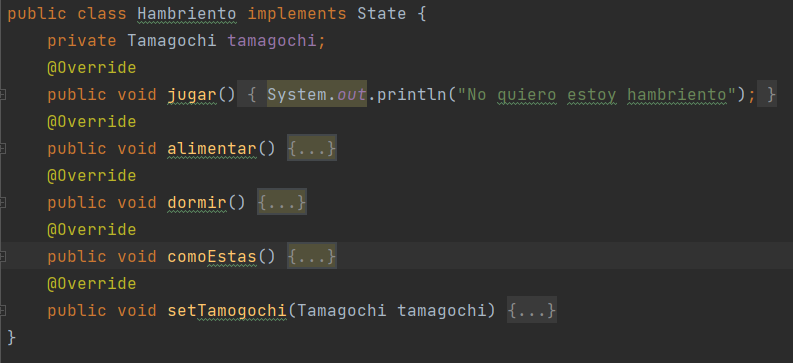


Clase base







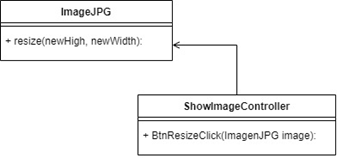


Los diferentes estados del proyecto

# Protected Variation / Variaciones protegidas (GRASP)

Como el nombre lo indica este patrón nos invita a que diseñemos un sistema protegido o pensado para futuras actualizaciones o modificaciones que el cliente requiera. Es común que se soliciten pequeños o grandes cambios y en el peor de los casos podría darse la necesidad de reestructurar el sistema por completo. ***Protected variation*** es el principio fundamental para protegerse frente al cambio.

Este principio recomienda se haga un análisis previo con el fin de encontrar “que” es susceptible a cambios para envolverlo en una interfaz y utilizando polimorfismo para tener la posibilidad de crear varias implementaciones; haciendo esto podemos cumplir con el requerimiento actual, pero también abierto a futuras actualizaciones sin que quede ligado a nuestro sistema. Al producirse variaciones o cambios, estos nos repercutirán lo mínimo posible.

A continuación, se presentarán unos diagramas para clarificar mejor el uso de este patrón.

Tenemos una clase ImagenJPG con un método que redimensiona una imagen y un controlador que usa esta asocia dicha clase en una función.

Podrías preguntarte *¿qué problema puede surgir?*, veamos lo de esta forma, *¿qué pasaría si en lugar de pasar una imagen en formato jpg decidimos pasar un png?*

Sencillo el programa daría error, pero inicialmente funcionaba. Realizar cambios con el agregar soporte a nuevos formatos de imagen podría ser un problema ya que el sistema no estaba pensado para el cambio.

Veamos la solución aplicando ***Protected variation***.



Como podemos ver aquí se aplica el principio, y se crea una interfaz image la cual será firmada por todos los tipos de imágenes que requiera nuestro sistema; dando la posibilidad de expandirlo conforme la necesidad sin que se dañe el sistema, ya que este en lugar de tomar una clase en sí, toma el contrato donde podrán ser seteadas (como argumento) las clases que firmaron dicho contrato.

Prototype (Creational)

La intención de este patrón es poder crear distintos objetos de un mismo tipo con los valores ya establecidos, usándose para ello una instancia prototípica que pueda clonarse a sí misma, sin que el código dependa de sus clases.

Un sistema debe ser capaz de crear objetos sin conocer su clase exacta, como son creados, o que datos representan, también cuando las clases serán instanciadas durante el tiempo de ejecución y cuando las instancias tienen unas cuantas combinaciones distintas y resulta más conveniente instalar prototipos y clonarlos en vez de crearlos manualmente con una ligera diferencia.

***La estructura básica de un prototipo es la siguiente:***

**Interfaz**: que define la operación de clonado. Será implementada por todos los objetos que puedan ser clonados. *En ocasiones es implementado como una clase abstracta.* **Client**: Crea nuevos objetos Mediante la clonación de los mismos. **ConcretePrototype1**: Clase que puede ser instanciada mediante la clonación de un prototipo. Implementa la interfaz Prototype. **ConcretePrototype2**: Otra clase con las mismas características que **ConcretePrototype1**.

Clonar un objeto puede tener aspectos positivos ya que es mucho más rápido que crearlo, un programa puede añadir y borrar dinámicamente objetos prototipo en tiempo de ejecución y el cliente no debe conocer los detalles de cómo construir los objetos prototipo.

El siguiente diagrama muestra cómo se estructura la clonación de un personaje en un videojuego tomando como base a objetos anteriormente creados, existen 3 tipos de personajes (elf, orc, y goblin) cada uno de ellos heredan una clase personaje la cual es abstracta.

Las clases Elf, Orc y Gobling agregan su funcionalidad adicional requerida e implementan el método clonar de su superclase (personaje).

A continuación, parte del código creado en base al diagrama anterior para clarificar las ideas.

|  |
| --- |
|  |

Ilustración 1- Abstract class

|  |  |
| --- | --- |
| Ilustración 2- Goblin class that extends Character   |  | | --- | |  | |

|  |  |
| --- | --- |
| |  | | --- | | Ilustración 3- Main class | |

# 

# COMMAND PATTERN

El *Command Pattern* es una solución probada que aborda problemas recurrentes de componentes de *invoker* y *receiver* estrechamente acoplados en las aplicaciones. Este patrón establece que las solicitudes deben encapsularse como objetos que, como cualquier otro objeto, pueden almacenarse y pasarse por la aplicación. Las solicitudes encapsuladas como objetos se conocen como *commands*.

Este patrón permite guardar listas de código que se ejecutan en otro momento o múltiples veces. Una de las ventajas de este patrón, es que permite la extensión de nuevos *commands* con facilidad sin cambiar el código existente.

Un objeto de *command* tiene un método *execute()* y un método *undo()* que puede ser opcional. El método *execute()* ejecuta una operación en el recibidor, mientras que el método *undo()* invierte la operación actual. La implementación de la operación es llevada a cabo por el *receiver*. El invocador solo se configura con un objeto de command e invoca un command llamando el método *execute()*. El invocador no sabe lo que el método *execute()* hará.

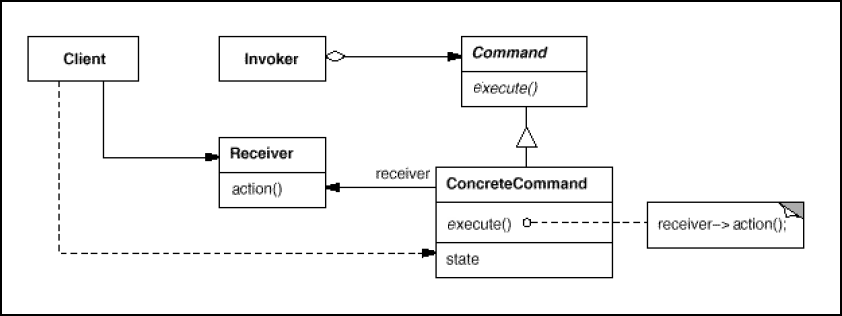
**Command:** Es una interfaz para ejecutar una acción.

**ConcreteCommand:** Clases que implementan Command y definen los métodos execute() y undo() para comunicarse con los receivers para realizar una acción y deshacerla, respectivamente.

**Invoker:** Pide al Command que lleve a cabo la acción.

**Receiver:** Realiza la acción según el comando que recibe.

**Client:** Crea un objeto ConcreteCommand y configura su receiver.



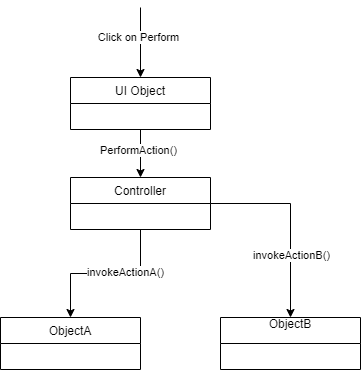
La consecuencia más grande de este patrón de diseño, es el hecho de que podemos terminar con muchas clases pequeñas.

# CONTROLLER PATTERN

Este patrón se ocupa de cómo delegar la solicitud desde el UI a un objeto en la lógica del programa. El controller podría ser considerado como el *“middle-man”*, quien se encarga de interpretar las acciones en el UI y sabe como conectar esas acciones a comportamientos en el sistema.

Este patrón permite separar el UI con la lógica del programa, y que no dependan uno del otro. Una manera fácil de interpretar este patrón es imaginándonos que el controller es el timón de un auto y este se encarga de conectar las intenciones del conductor con las acciones del auto. Si nosotros decidimos cambiar el motor del auto, eso no cambia como nosotros manejamos dicho auto. De igual manera, cambiar el timón no requiere que los componentes internos del carro sean modificados.

Cuando una petición viene de la UI, el controller pattern nos ayuda a determinar cuál es el primer objeto que recibe el mensaje de la UI. Su función es delegar el trabajo de otras clases y coordinar la actividad en general.



Una de las consecuencias de usar este patrón de diseño, es que el controlador puede volverse muy pesado o cargado. Pero en dado caso solo es de crear más controladores y repartir las responsabilidades.

**Repo:**

<https://github.com/MichaelBosworth2/PatternDesigns/tree/main/src/main/java>