
Guía rápida **PATRONES DE DISEÑO**



Los patrones de diseño no son otra cosa que plantillas para solucionar problemas de diseño recurrentes que podrás encontrar en cualquier aplicación o juego.

Estas plantillas aportan una forma estandarizada de solucionar los problemas que ya está contrastada y validada.

La aplicación de estos patrones aportan un lenguaje común con el que comunicarse entre programadores, esto es debido a que cada patrón intenta resolver un problema concreto y tiene asignado un nombre único. Cuando un programador experimentado escucha el nombre de un patrón ya sabe que problemática intenta resolver y cuál es su implementación.

La función de esta guía no es otra que facilitar la aplicación de estos patrones de diseño, sobretodo cuando estás en una fase de aprendizaje y aún no los dominas.

Mi recomendación es que tengas esta guía siempre a mano cuando estés programando, ya sea en papel o digital. Así cuando vayas a resolver un problema, según sea de creación, comportamiento o estructura, podrás acudir a la sección correspondiente y encontrar un resumen de todos los patrones que podrías aplicar y ver cuál se adapta más.

En cada una de las secciones encontrarás los patrones que la componen así como un pequeño resumen y un diagrama de una posible implementación. De ti depende profundizar y adaptar ese diagrama a tus necesidades.

Patrones de creación	4
Patrones estructurales	11
Patrones de comportamiento	19
Sobre mí	31

Patrones de creación

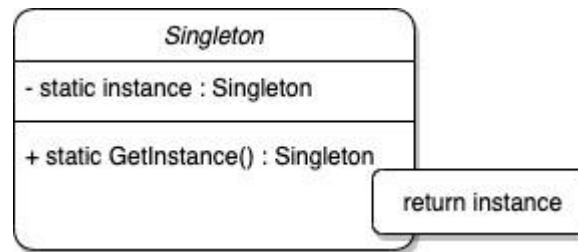
Estos patrones abstraen el proceso de creación de los objetos, lo que nos ayuda a separar como se comporta un objeto (o instancia) de cómo lo tenemos que construir.

Dicho de otra forma, estos patrones esconden los detalles de cómo vamos a crear cierto objeto, si lo haremos con un `new`, o utilizaremos un `Instantiate()` en Unity, o si una vez construido necesitamos llamar a varios métodos para pasarle sus colaboradores, etc.

Aquí encontramos los patrones: Singleton (4), Monostate (5), Factory Method (6), Abstract Factory (7), Builder (8), Prototype (9).

Singleton

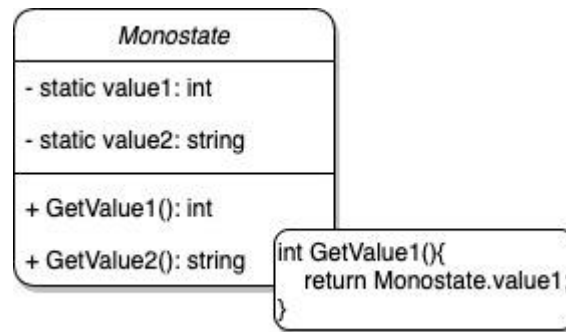
Se asegura de que solo exista una única instancia y proporciona un acceso global a esta.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-singleton-y-monostate/>

Monostate

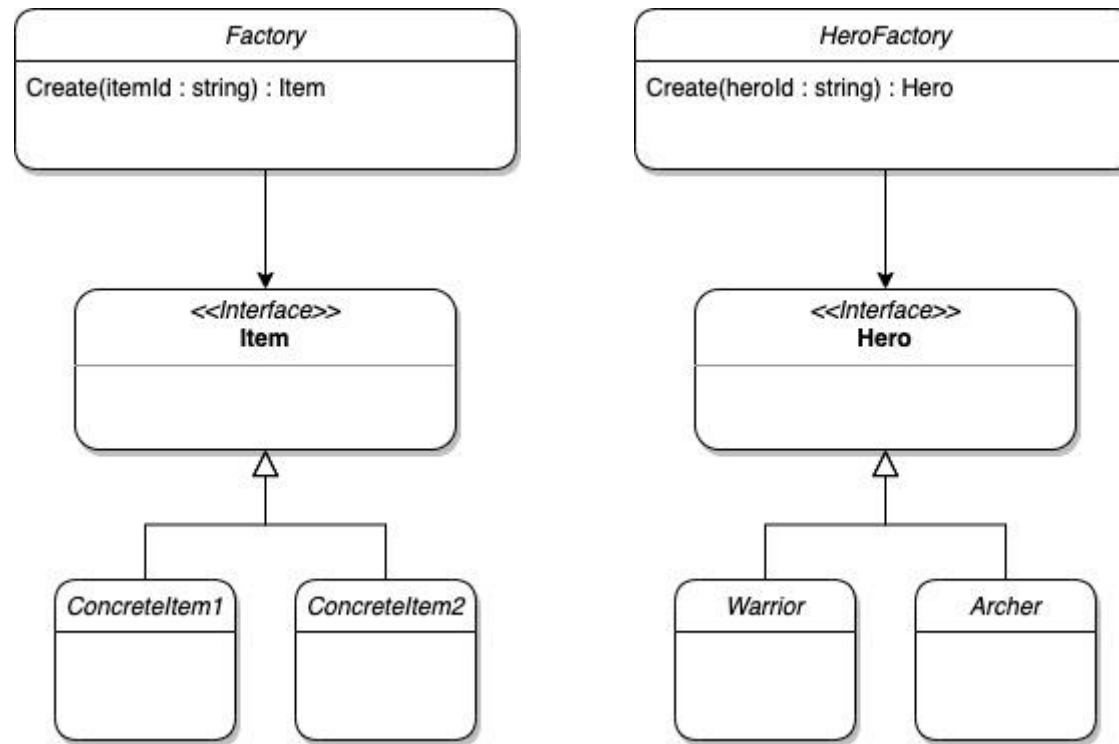
Se asegura de que solo exista un único estado. Todos los datos son estáticos, así que todas las instancias comparten el mismo estado.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-singleton-y-monostate/>

Factory Method

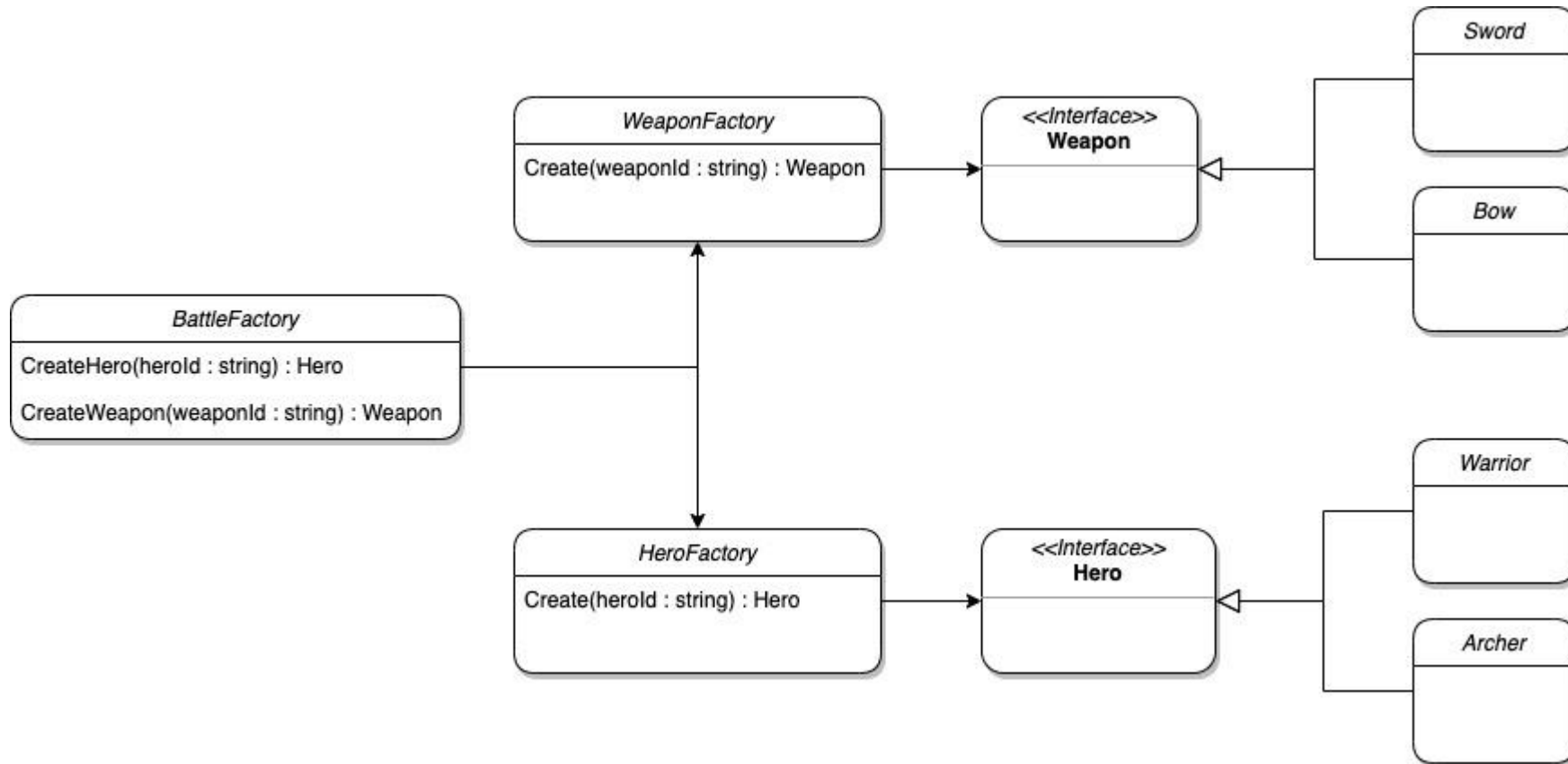
Abstrae como creamos objetos de un tipo concreto sin que nos tengamos que preocupar de los detalles o depender de la clase concreta.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-factory-method/>

Abstract Factory

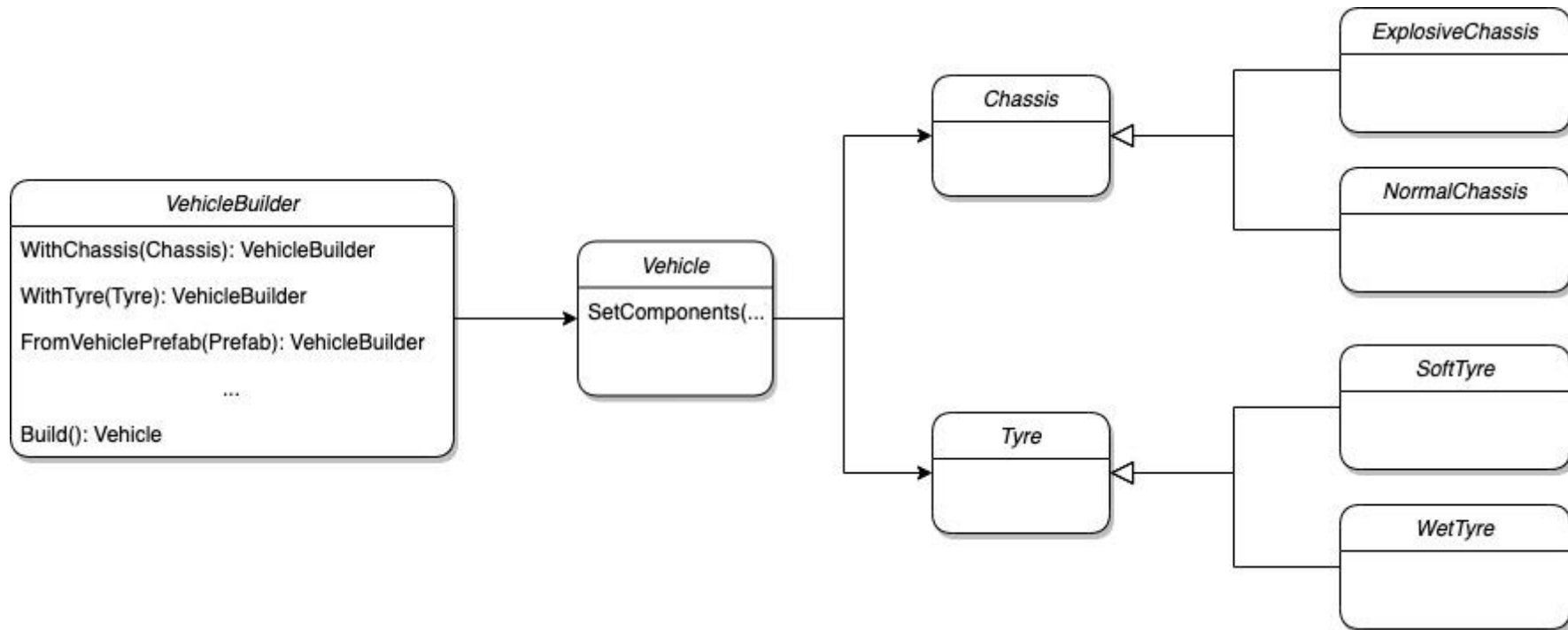
Proporciona la funcionalidad de poder crear una familia de objetos relacionados, o dependientes entre sí.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-abstract-factory/>

Builder

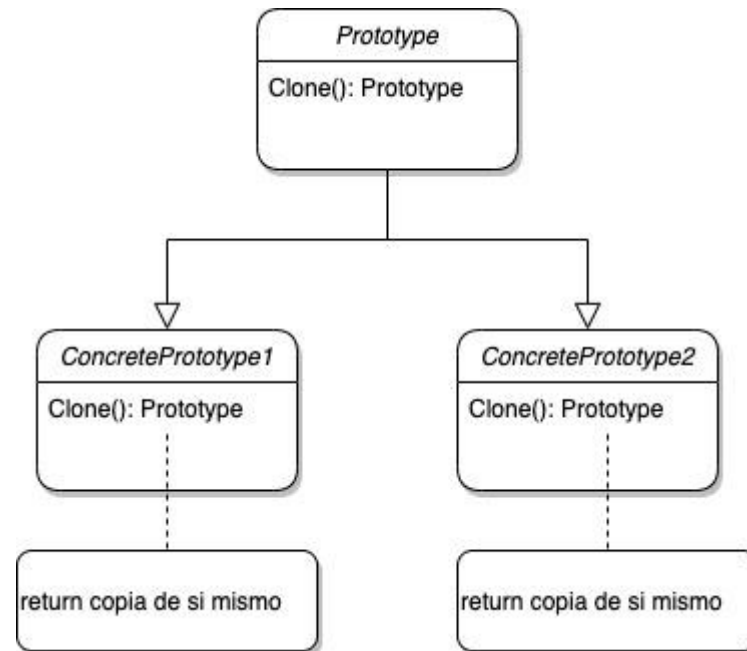
Nos permite crear un objeto complejo por partes. Le iremos proporcionando las distintas partes y cuando lo tenga todo le pediremos que nos entregue el objeto construido.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-builder/>

Prototype

Nos permite crear un objeto a partir de una instancia que actúa como arquetipo o molde.



Para profundizar: [https://es.wikipedia.org/wiki/Prototipo_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Prototipo_(patr%C3%B3n_de_dise%C3%B1o))

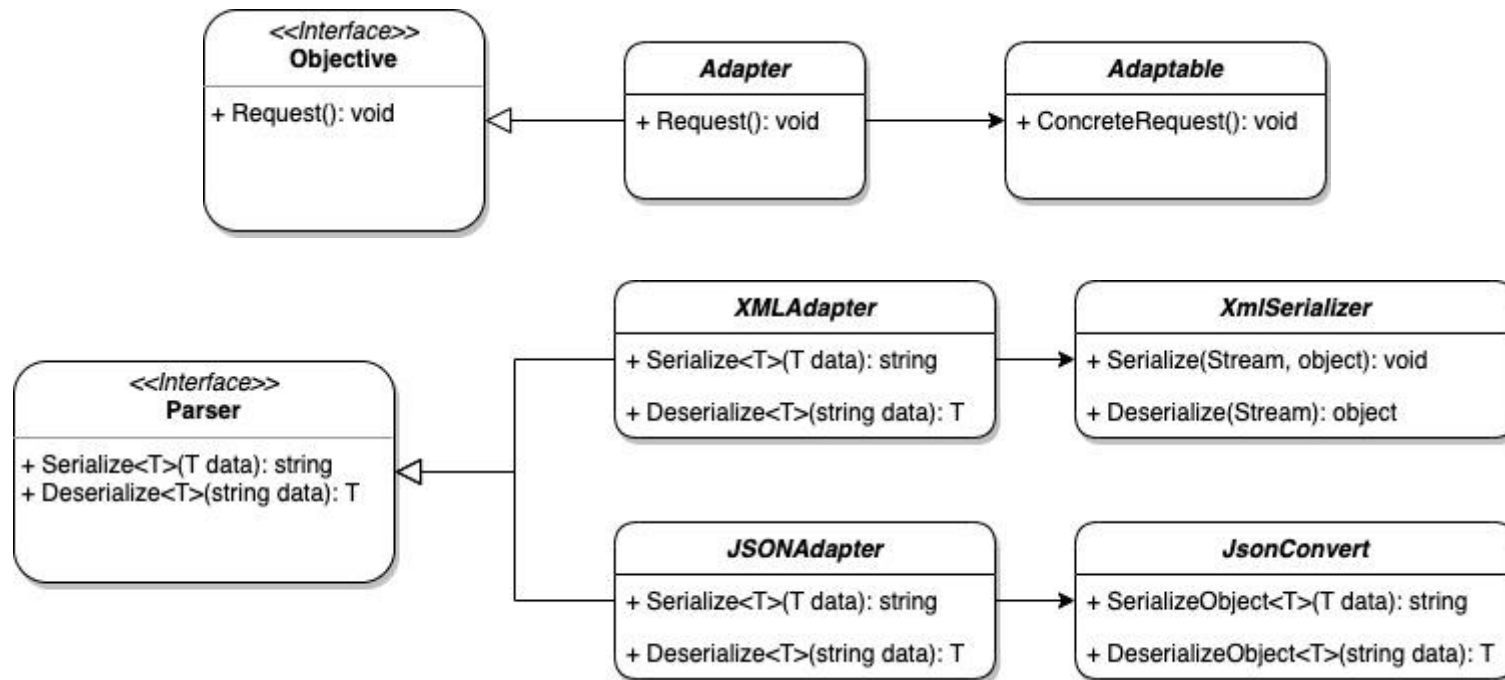
Patrones estructurales

Los patrones estructurales se encargan de resolver cómo se combinan las clases, ya sea mediante herencia o composición, para crear estructuras más complejas.

Aquí encontramos los patrones: Adapter (11), Decorator (12), Composite (13), Bridge (14), Facade (15), Flyweight (16) y Proxy (17).

Adapter

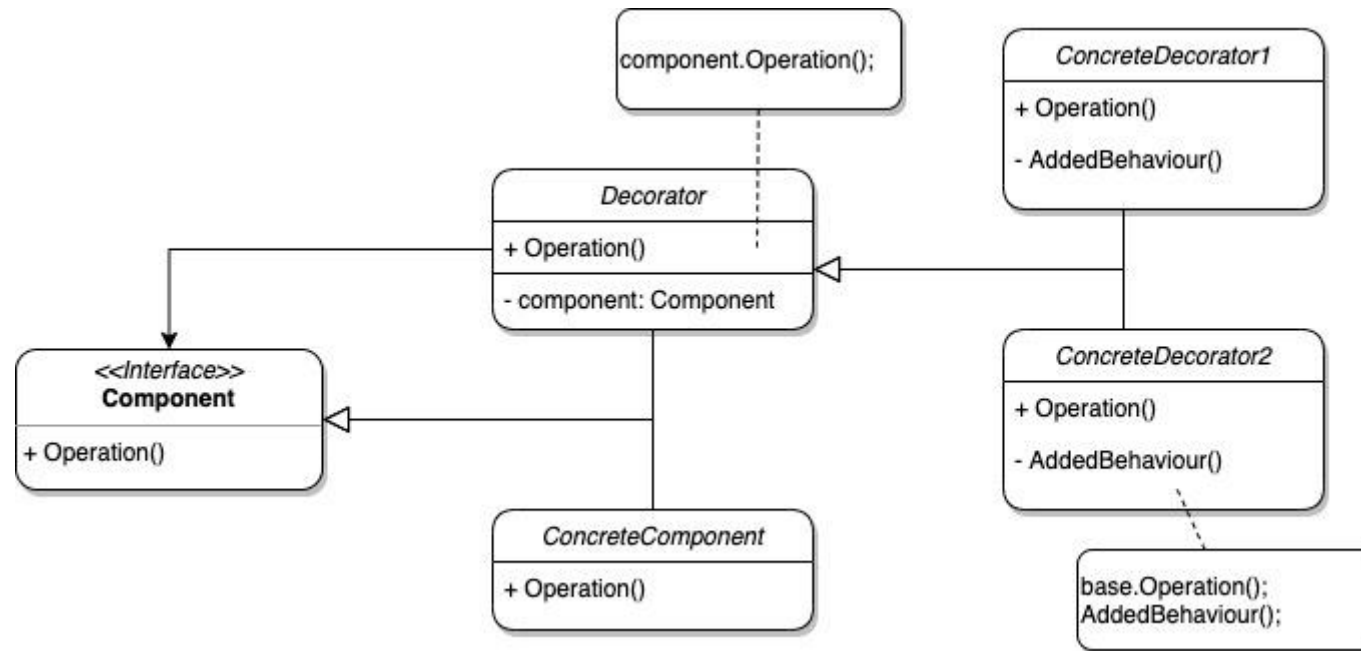
Nos abstrae de la clase concreta permitiéndonos más flexibilidad. Podemos adaptar una interfaz a otra que se adapte a nuestro sistema y convertir clases estáticas en clases que podemos instanciar.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-adapter/>

Decorator

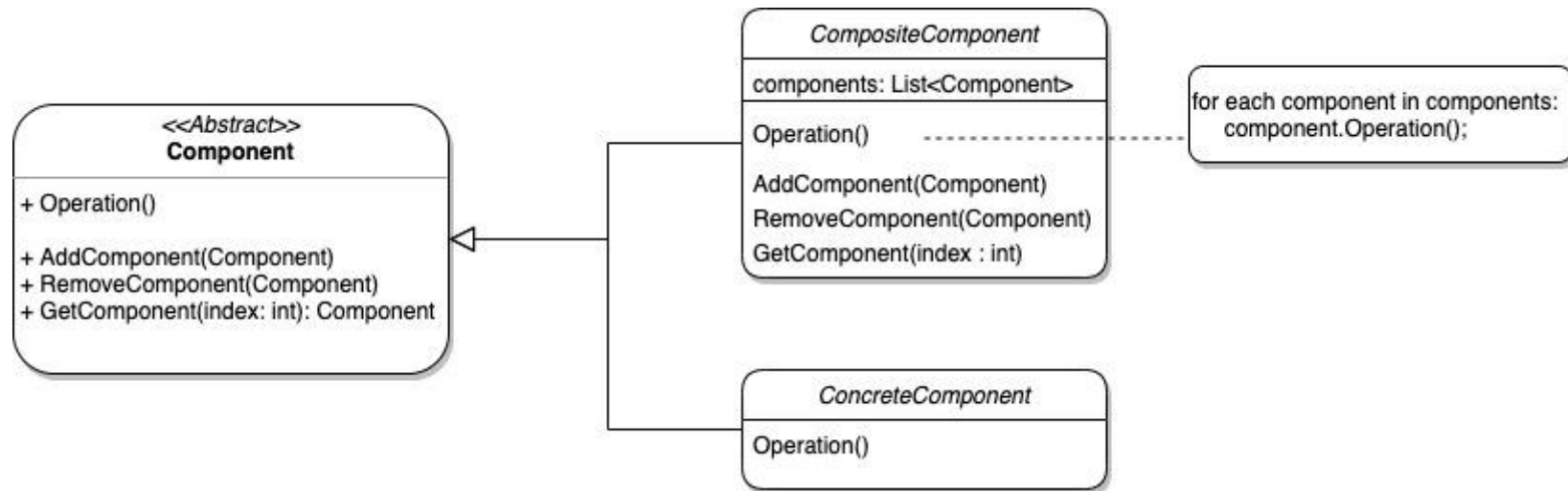
Es una forma fácil de añadir responsabilidades adicionales a un objeto de forma dinámica y sin tener que modificar ese objeto. Es una alternativa a la herencia.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-decorator/>

Composite

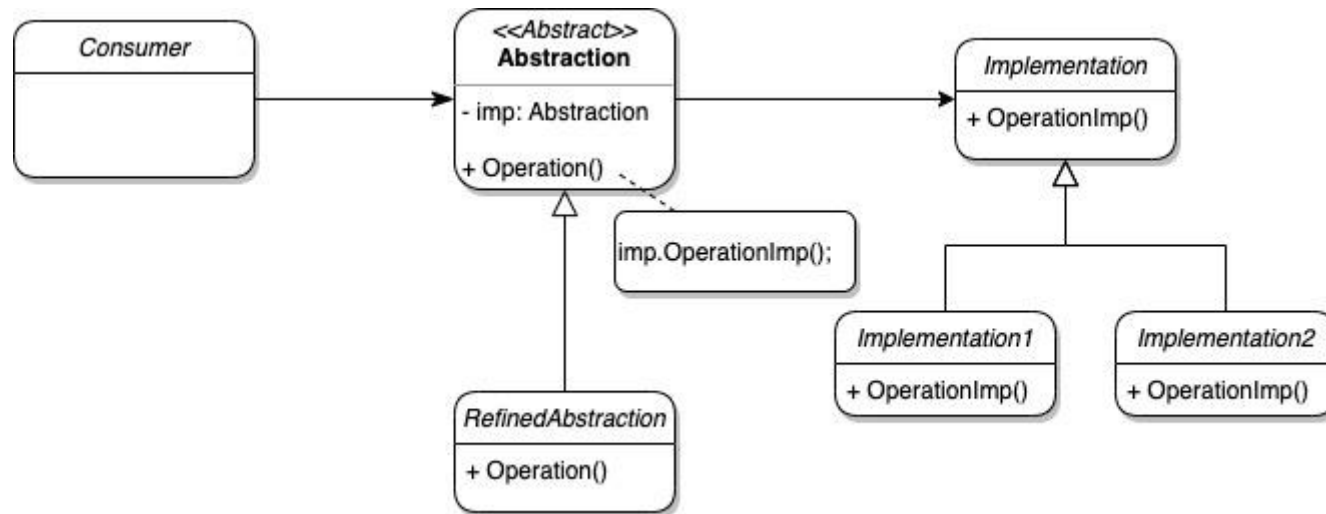
Su objetivo es que el consumidor no sepa si está hablando con un solo objeto, o si hay mil objetos detrás de este.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-composite/>

Bridge

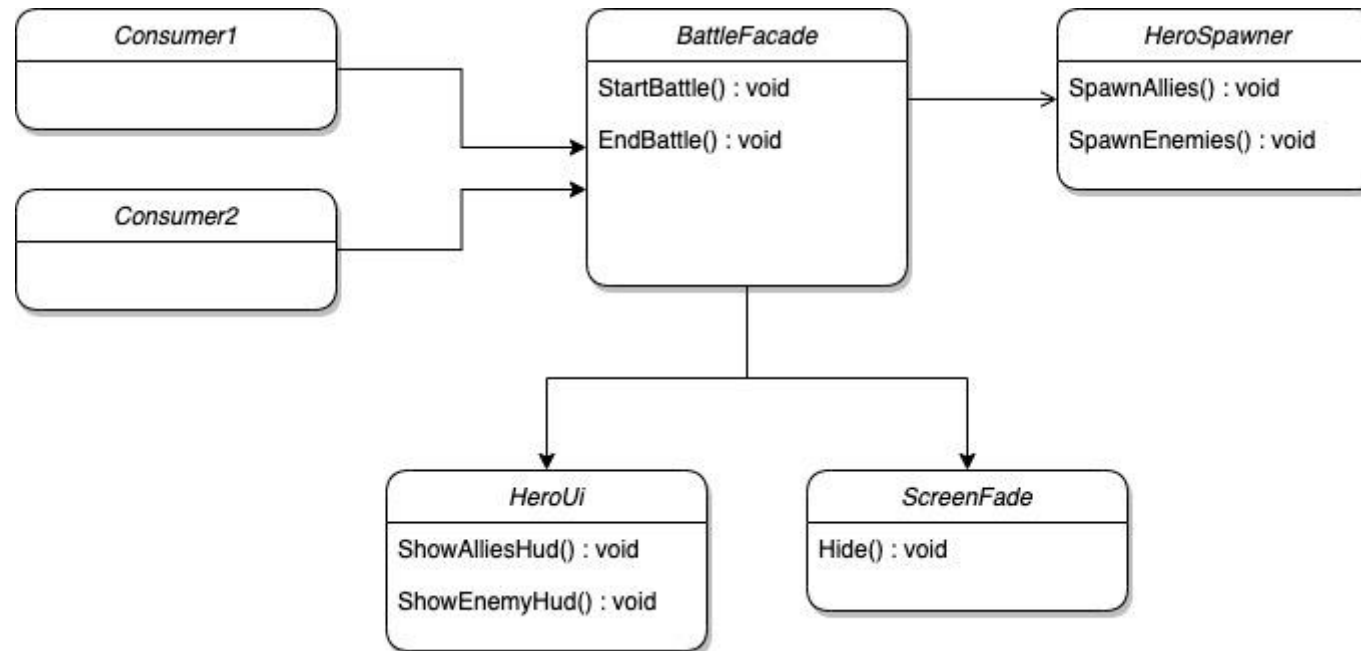
Desacopla una abstracción de su implementación concreta y lo hace gracias a la composición.



Para profundizar: [https://es.wikipedia.org/wiki/Bridge_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Bridge_(patr%C3%B3n_de_dise%C3%B1o))

Facade

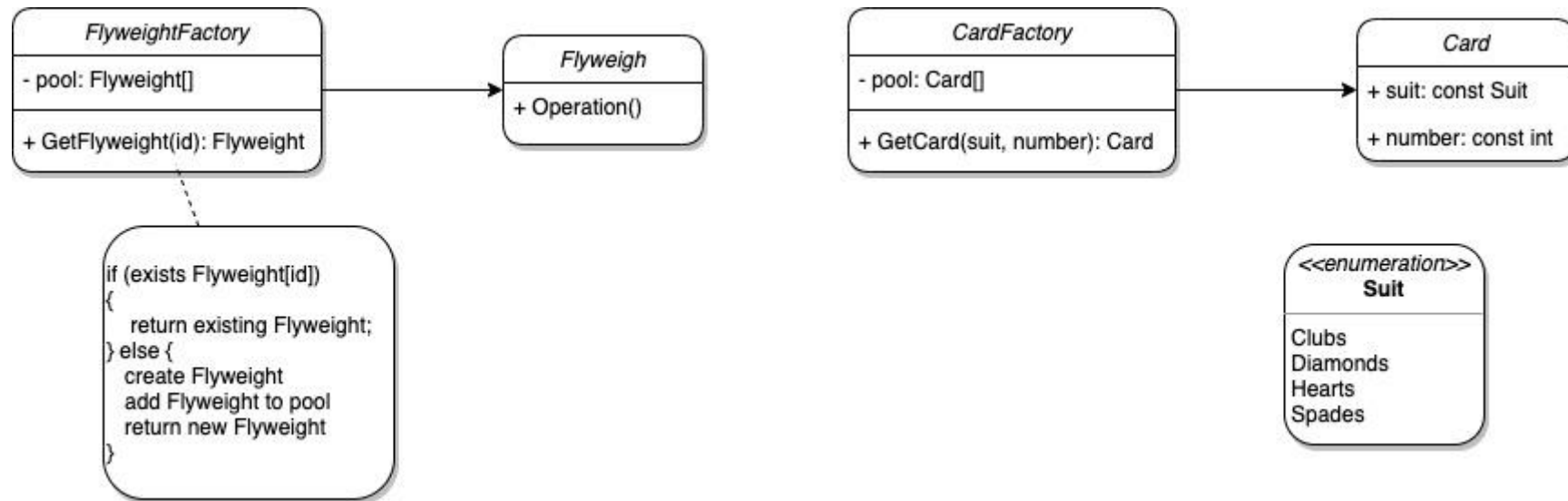
Hace de intermediario (de fachada) entre los consumidores y los distintos sistemas.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-facade/>

Flyweight

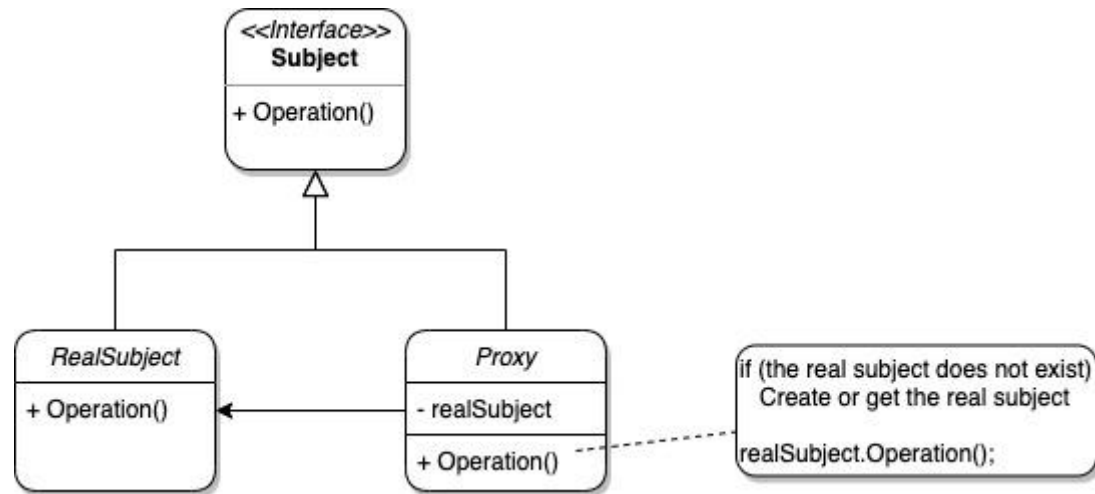
Permite reducir el número de objetos repetidos y compartirlos entre varios consumidores. Se asemeja a una pool pero no es lo mismo.



Para profundizar: [https://es.wikipedia.org/wiki/Flyweight_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Flyweight_(patr%C3%B3n_de_dise%C3%B1o))

Proxy

Proporciona un sustituto o intermediario de otro objeto.



Para profundizar: [https://es.wikipedia.org/wiki/Proxy_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Proxy_(patr%C3%B3n_de_dise%C3%B1o))

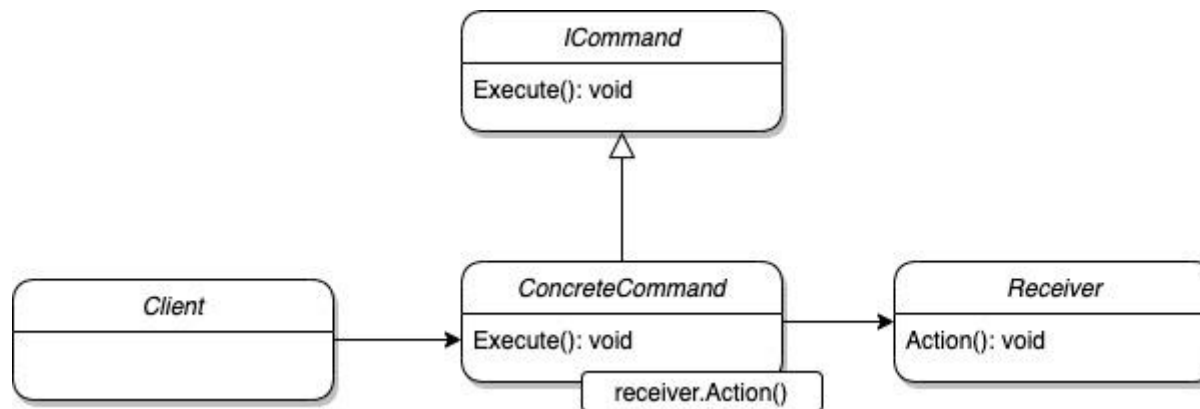
Patrones de comportamiento

Los patrones de comportamiento hablan de la lógica de nuestro sistema, algoritmos, responsabilidades de las clases y cómo se comunican entre sí.

Aquí encontraremos los patrones: Command (19), Mediator (20), Visitor (21), Observer (22), Template Method (23), State (24), Strategy (25), Memento (26), Iterator (27), Interpreter (28), Chain of Responsibility (29).

Command

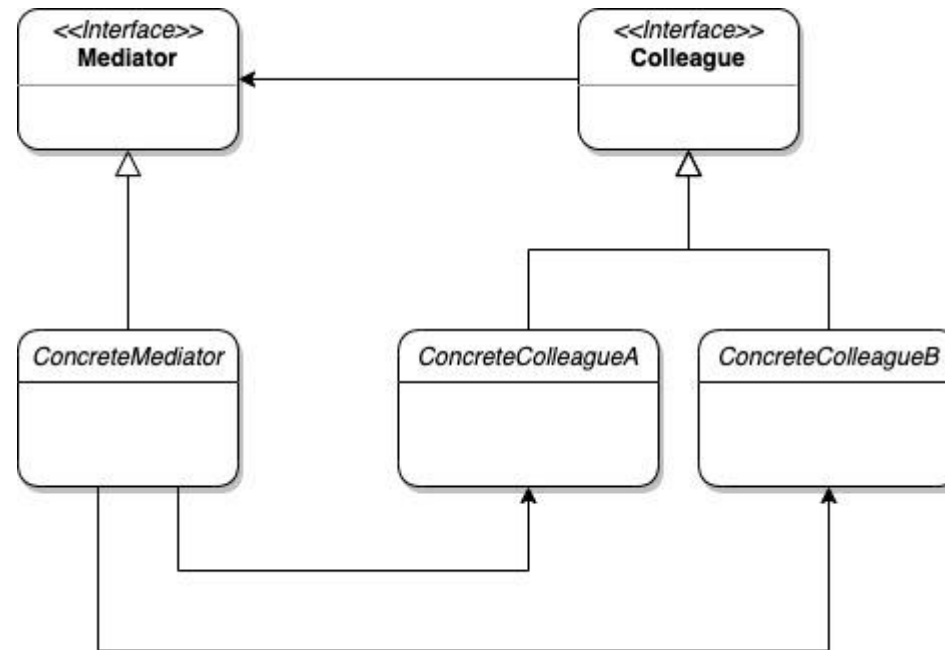
Este patrón encapsula una orden.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-command/>

Mediator

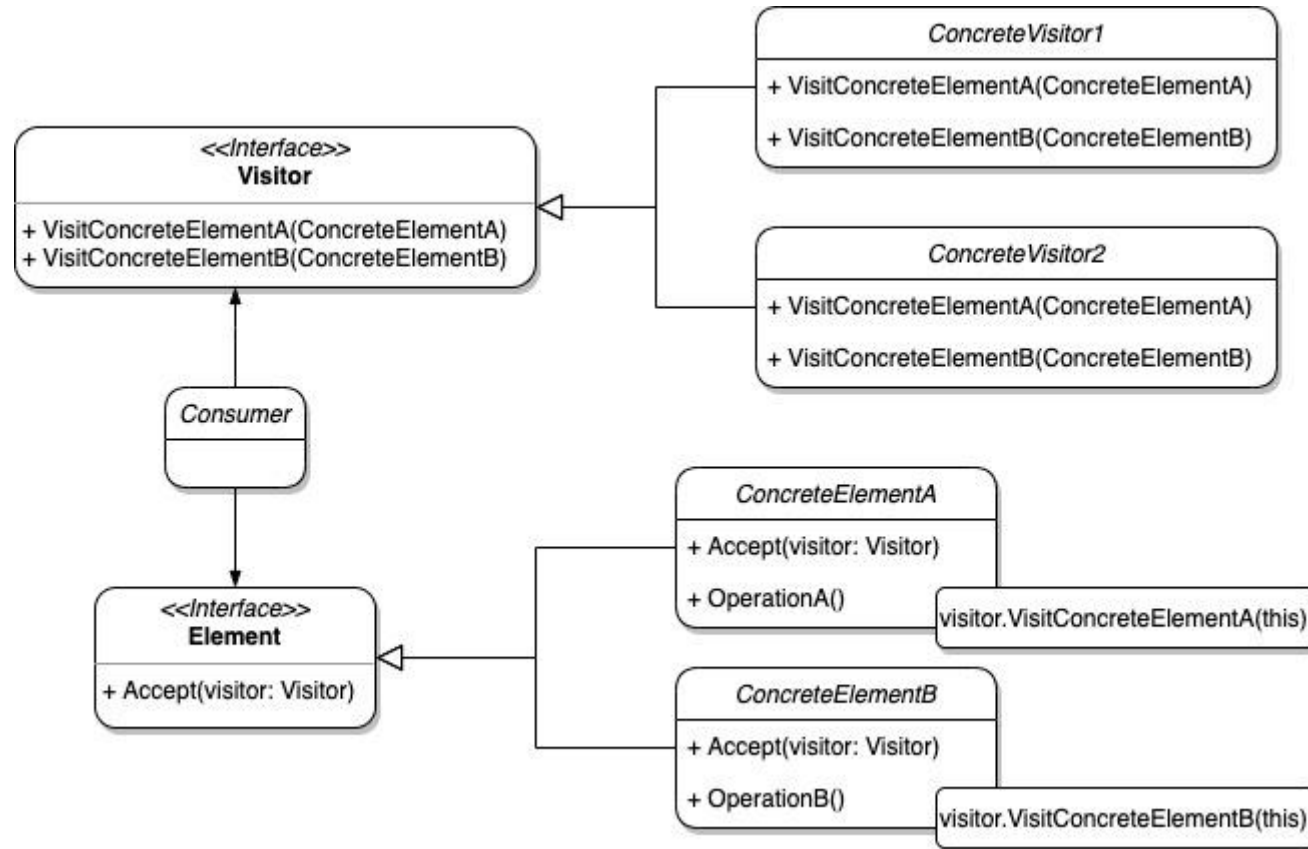
Encapsula cómo interactúan varios objetos entre sí, nos ayuda a tener un bajo acoplamiento.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-mediator/>

Visitor

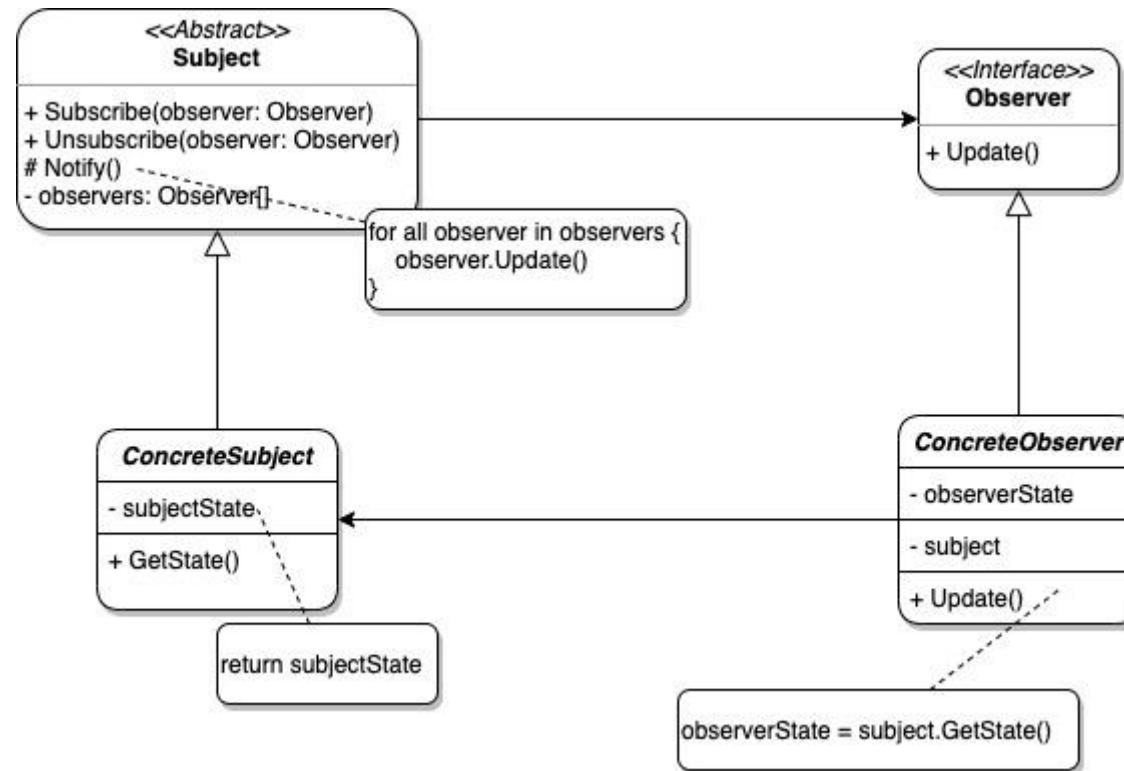
Permite añadir más operaciones a un sistema sin tener que cambiar las clases de los elementos con los que trabaja.



Para profundizar: [https://es.wikipedia.org/wiki/Visitor_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Visitor_(patr%C3%B3n_de_dise%C3%B1o))

Observer

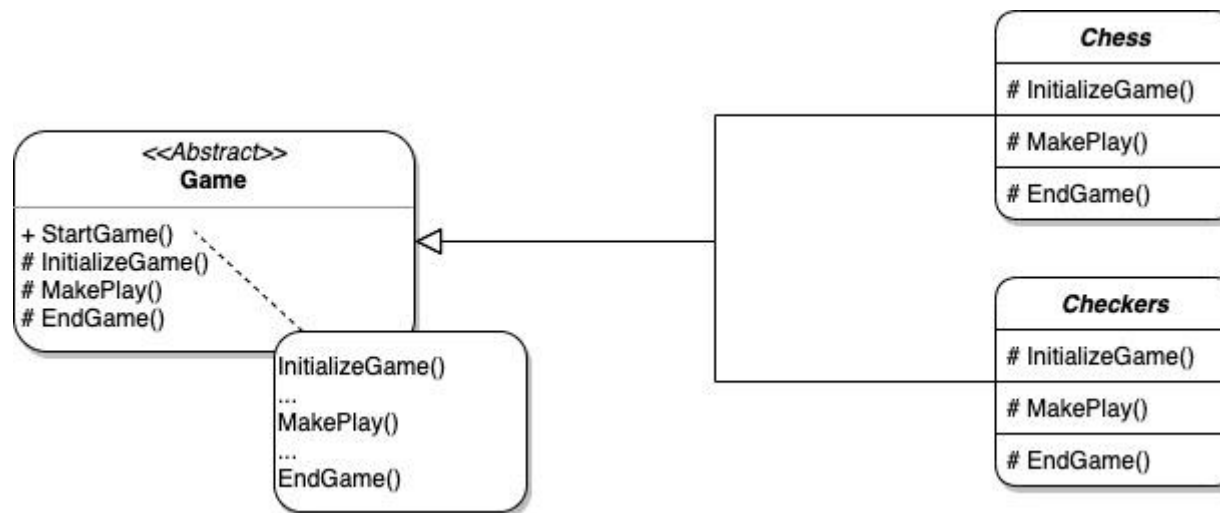
Se encarga de notificar a los interesados cuando un objeto cambia de estado y estos actualicen automáticamente los datos que dependan de ese objeto.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-observer/>

Template Method

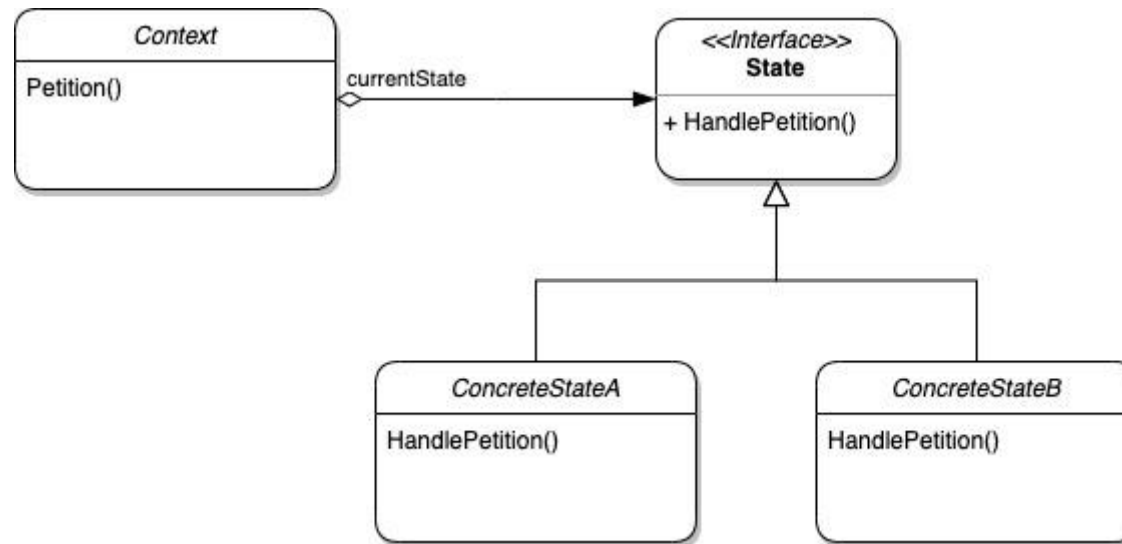
Define el esqueleto del algoritmo y delega en las subclases que redefinan ciertas partes para modificarlo.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-template-method/>

State

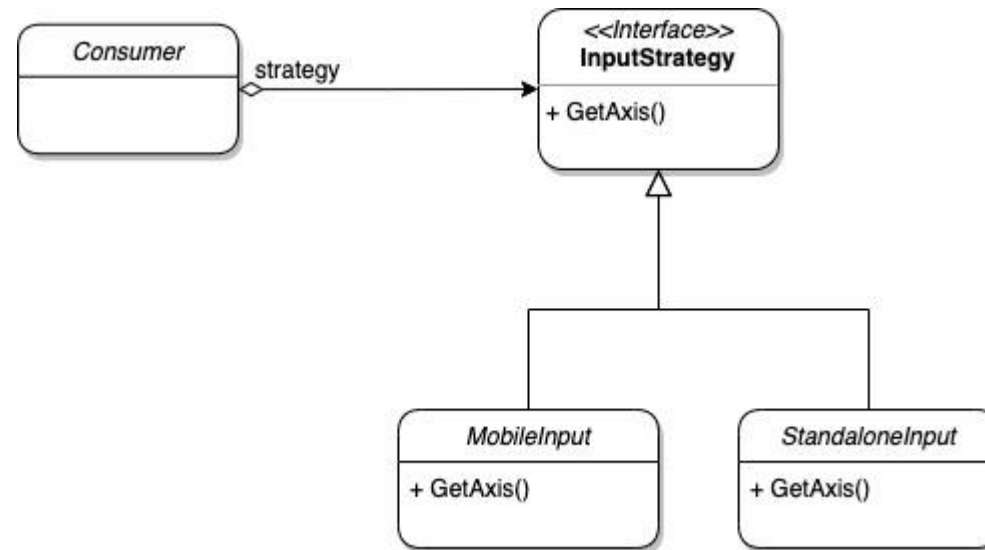
Permite de forma sencilla que un objeto cambie su comportamiento en función del estado en el que se encuentra.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-state-pattern/>

Strategy

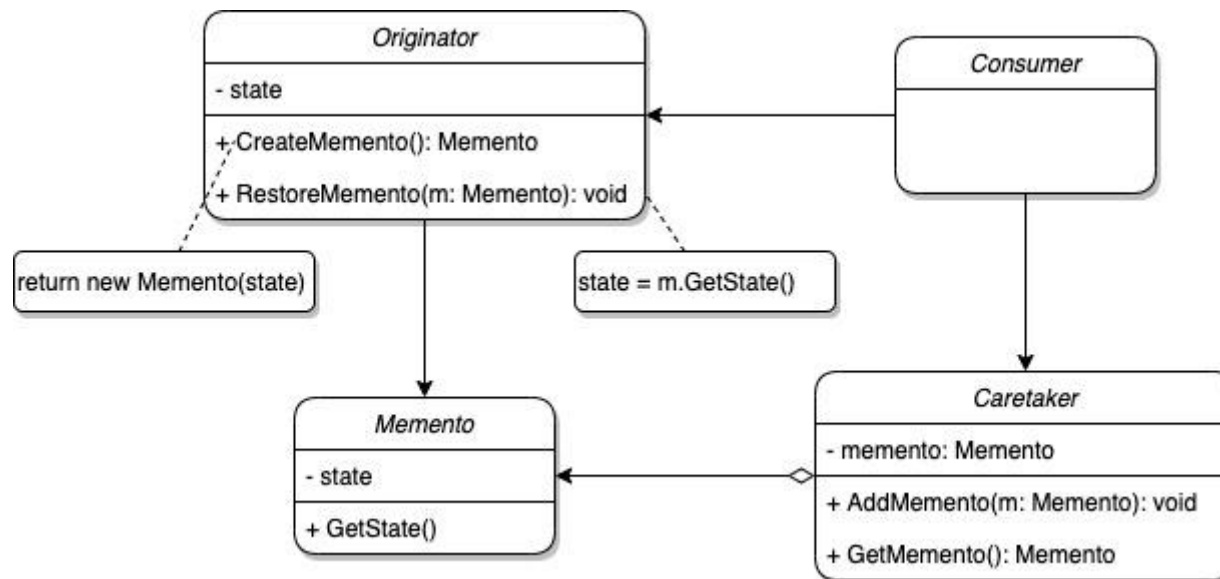
También conocido como inyección de dependencias. Define una familia de algoritmos y nos permite intercambiarlos sin que el consumidor se vea afectado.



Para profundizar: <https://thepowerups-learning.com/patrones-de-diseno-strategy/>

Memento

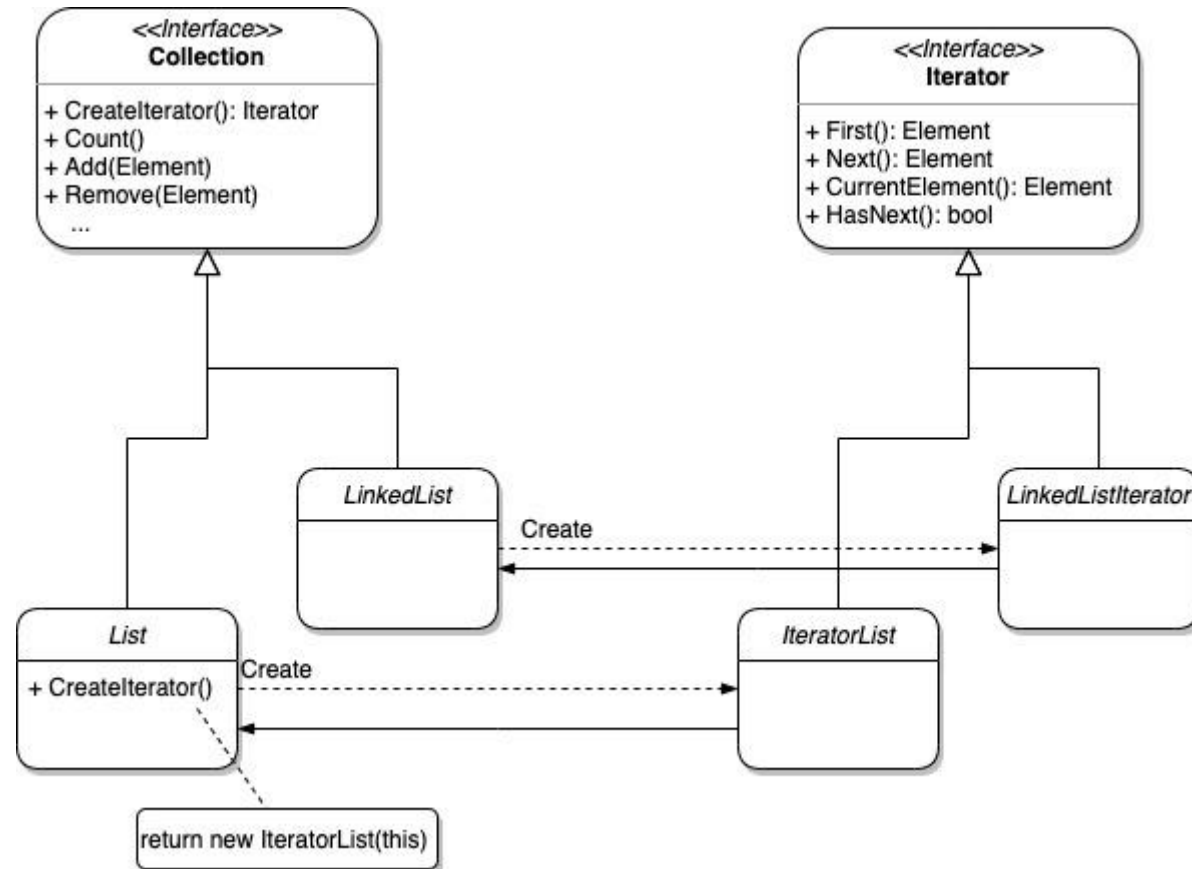
Almacena el estado de un objeto en un momento dado de forma que lo podamos restaurar a este estado de manera sencilla.



Para profundizar: [https://es.wikipedia.org/wiki/Memento_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Memento_(patr%C3%B3n_de_dise%C3%B1o))

Iterator

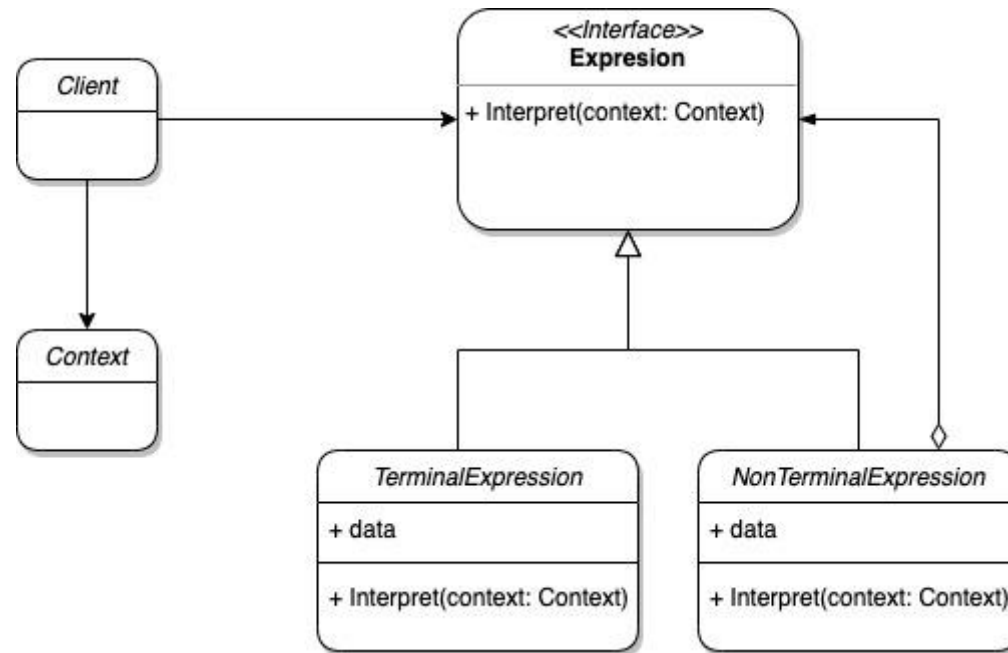
Nos proporciona los métodos necesarios para recorrer secuencialmente una colección de objetos.



Para profundizar: [https://es.wikipedia.org/wiki/Iterador_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Iterador_(patr%C3%B3n_de_dise%C3%B1o))

Interpreter

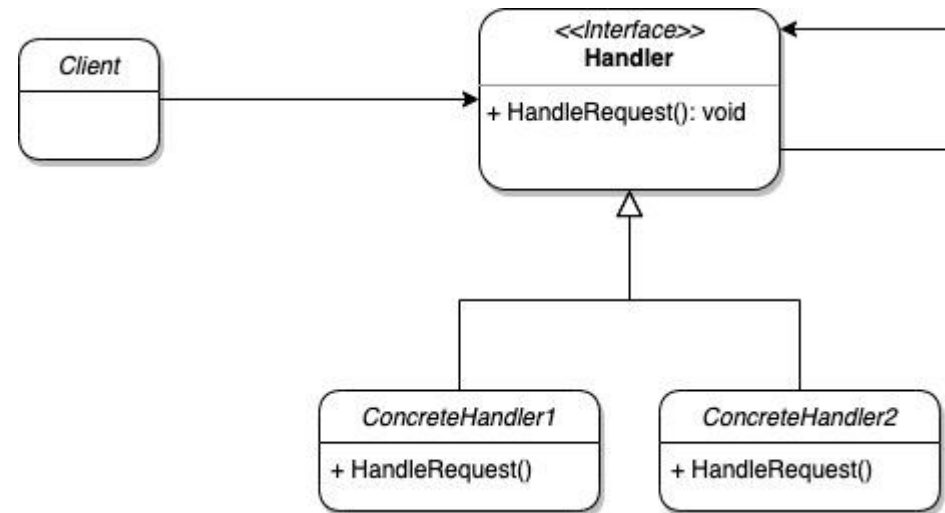
Dado un lenguaje, define una representación para su gramática junto con un intérprete que utiliza la representación para interpretar las instrucciones.



Para profundizar: [https://es.wikipedia.org/wiki/Interpreter_\(patr%C3%B3n_de_dise%C3%B1o\)](https://es.wikipedia.org/wiki/Interpreter_(patr%C3%B3n_de_dise%C3%B1o))

Chain of Responsibility

Evita acoplar el emisor de una petición con su receptor y le da la posibilidad a más de un objeto de responder a esa petición.



Para profundizar: https://es.wikipedia.org/wiki/Cadena_de_responsabilidad

Sobre mí



Soy Daniel Parra, desarrollador de videojuegos desde hace 10 años y un apasionado de la programación.

Llevo toda mi carrera profesional desarrollando videojuegos para distintas plataformas y con distintos motores. He desarrollado juegos para WiiWare, PS4, Steam/PC y dispositivos móviles, y he trabajado con Unity, Unreal y motores propios de la empresa desarrollados en C++.

Soy un fanático del código limpio, los principios SOLID, patrones de diseño, TDD y arquitectura. Creo que uno de los problemas de que los proyectos se dilaten en el tiempo y acaben costando más dinero del estimado, es en gran medida por la falta de aplicar buenas prácticas y pensar en el largo plazo del proyecto.

En el caso de que te haya gustado esta guía, te animo a que sigas profundizando en los patrones de diseño, por lo que te recomiendo mi curso “Patrones de diseño para VIDEOJUEGOS”.

Más información:

<https://curso-patrones-para-videojuegos.thepowerups-learning.com/>

Guía rápida de patrones de diseño

[THEPOWERUPS-LEARNING.COM](https://thepowerups-learning.com)