SOLID

SOLID es un acrónimo mnemotécnico, son 5 principios para hacer el los diseños de POO mas entendibles, flexibles y mantenibles.

- **Single Responsibility principle:** todas las clases deben tener una sola responsabilidad. No debe haber mas de una razón por la cual la clase debe cambiar
- **Open-Closed principle:** las entidades deben ser abiertas a su extension y cerradas a su modificación.
- **Liskov substitution principle:** las funciones que usan punteros o referencias a la clase base deben poder usar objetos de la clase derivada sin saberlo.
- **Interface Segregation Principle:** clientes no pueden ser forzados a depender de interfaces que no usen.
- **Dependency inversion principle:** depende de abstracciones, no concreciones.

Sintomas de Diseño Podrido

Rigidez: es difícil de cambiar. Cada cambio causa una cascada de cambios en modulos dependientes a este.

Fragilidad: es la tendencia a que el software se rompa en todos los lugares donde es cambiado. A veces los cambios ocurren en areas que no están relacionadas.

Inmovilidad: es la incapacidad de reutilizar software del proyecto, para otros proyectos o para este mismo.

Viscosidad: viscosidad del diseño: cuando las formas de hacer algo no preservan el diseño. Viscosidad del environment: cuando el entorno de desarrollo es lento e ineficiente.

OCP (Open Closed Principle)

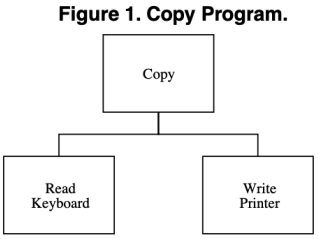
Los modulos deben ser abiertos a la extension y cerrados a la modificación. Todas las técnicas para lograr esto se basan en las abstracciones.

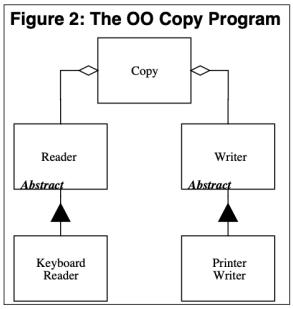
Técnicas:

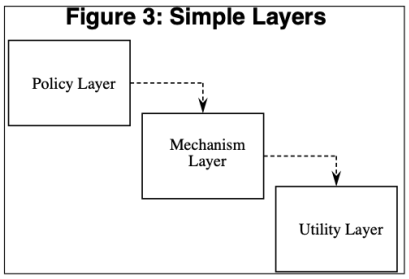
Dynamic Polymorphism: por ejemplo si se tiene una función LogOn que tiene un if/else preguntando cada tipo de modem que es para poder hacer algo diferente, es mejor tener una abstracción Modem con el método en cuestión.

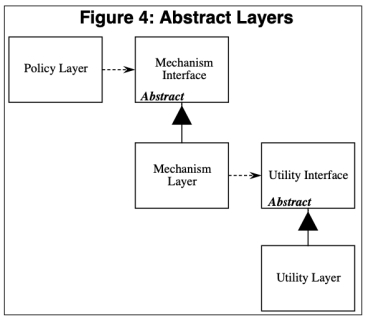
DIP (Dependency Inversion Principle)

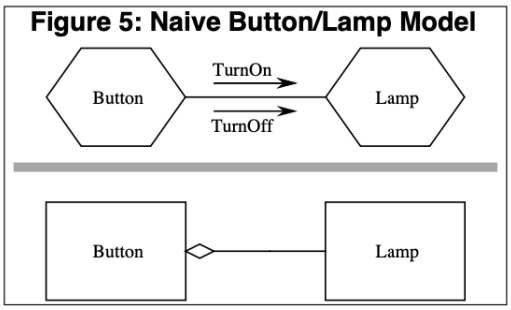
Es la union entre OCP y LSP. Los modulos de alto nivel no deben depender de los modulos de bajo nivel, sino que ambos deben depender de abstracciones. Las clases de nivel superior no deben depender de las clases de nivel superior, sino que ambas deben depender de interfaces o abstracciones.
Promueve el desarrollo de código flexible y desacoplado (esto porque las clases no dependen directamente de otras sino de una abstracción). Consigue una mayor modularidad, reutilización de código y facilidad de introducir nuevas implementaciones sin tener que cambiar todo el código.

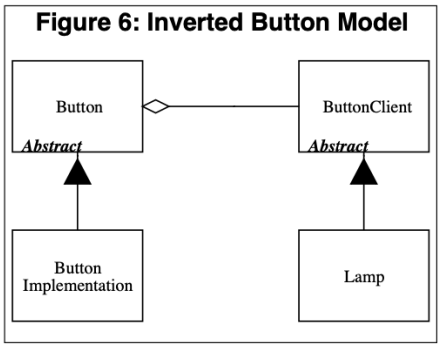




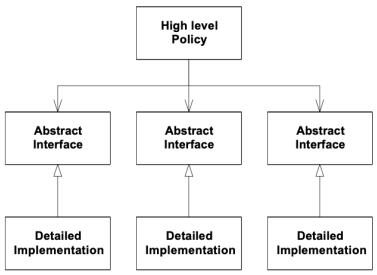








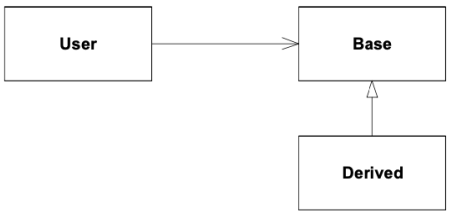
Ningún objeto puede depender de una clase concreta, todos deben depender de interfaces o abstracciones.



LSP (Liskov Substitution Principle)

Las instancias de una clase derivada pueden ser sustituidas por instancias de su clase base sin alterar la corrección del programa. Si la clase A es subclase de la clase B entonces los objetos de la clase B pueden ser reemplazados por objetos de la clase A sin afectar el funcionamiento del programa.

Por ejemplo si User tiene una función que toma como argumento un tipo de Base, entonces se puede pasar una instancia de Derived a la función.



Por ejemplo: Circle viola el contrato de Ellipse al no ser reemplazable, por ende viola LSP.

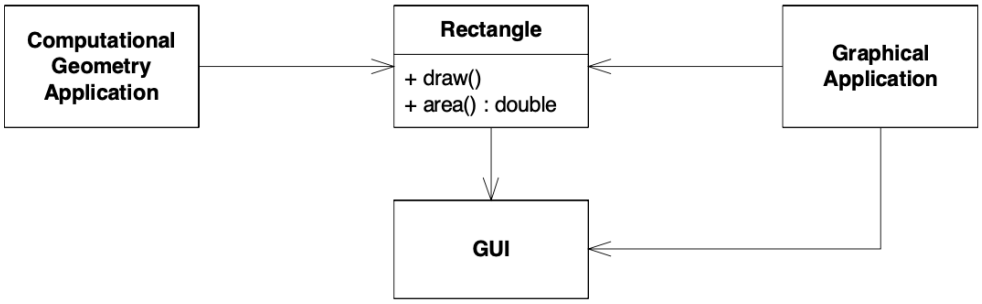
ISP (Interface Segregation Principle)

Establece que los modulos no deben depender de interfaces que no utilizan. En lugar de tener una interfaz monolítica que tenga muchos métodos, se deben crear interfaces mas especificas y cohesivas, adaptadas a las necesidades de cada modulo. Al tener interfaces mas pequeñas y especializadas, se promueve un acoplamiento mas débil entre los modulos y las implementaciones. Mejora la modularidad y la cohesion.

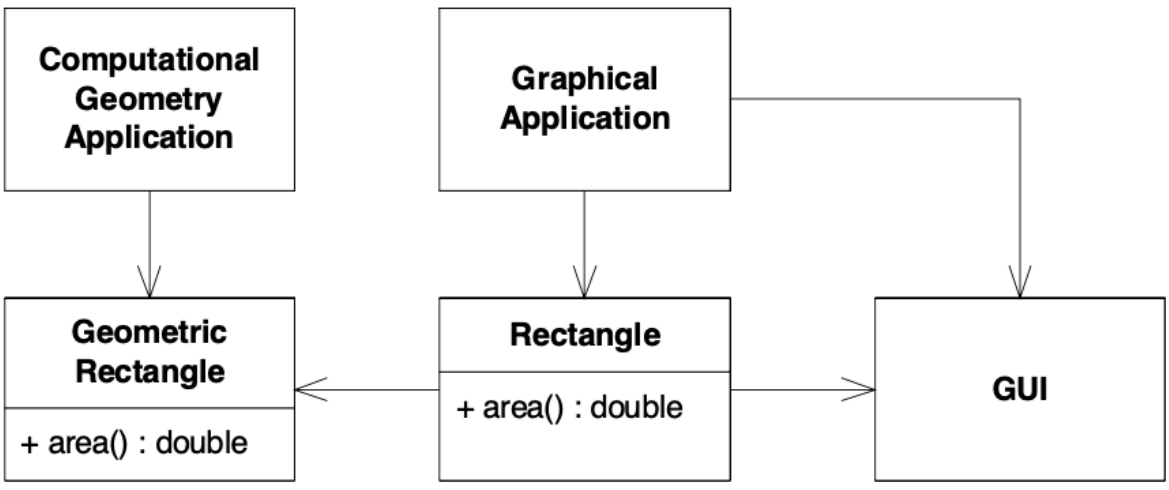
SRP (Single Responsibility Principle)

No debe haber mas de una razón para que la clase cambie.

Si una clase asume mas de una responsabilidad, entonces va a tener mas de una razón para cambiar. Las responsabilidades se acoplan, lo que conlleva a diseños frágiles.



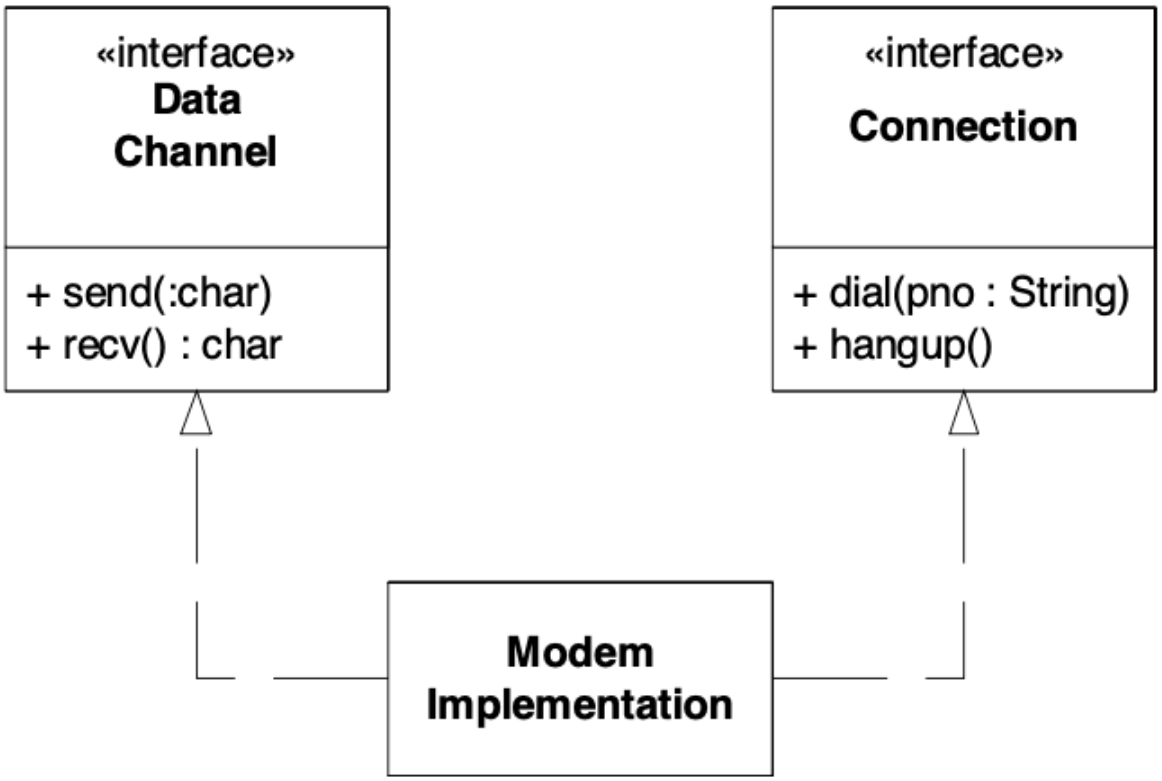
Dos aplicaciones diferentes usan la clase rectángulo, una hace geometría y la otra muestra de forma gráfica. Viola SRP porque tiene 2 responsabilidades. 1 proveer el modelo matemático del rectángulo, 2 renderizar el rectángulo. Si hay un cambio en GraphicalApplication causa que el rectángulo cambie su razón, esto conlleva rebuild, retest, redeploy también de ComputationalGeometryApplication.



Listing 9-1

Modem.java -- SRP Violation

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```



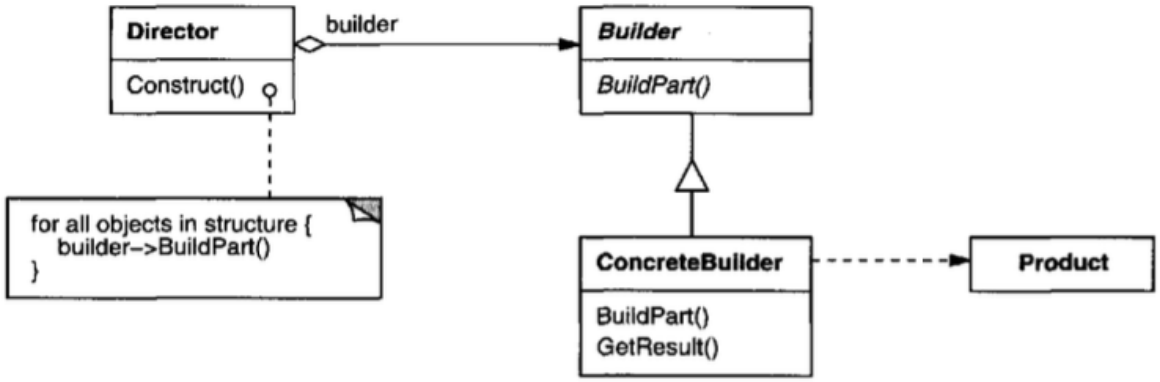
Patrones Creacionales

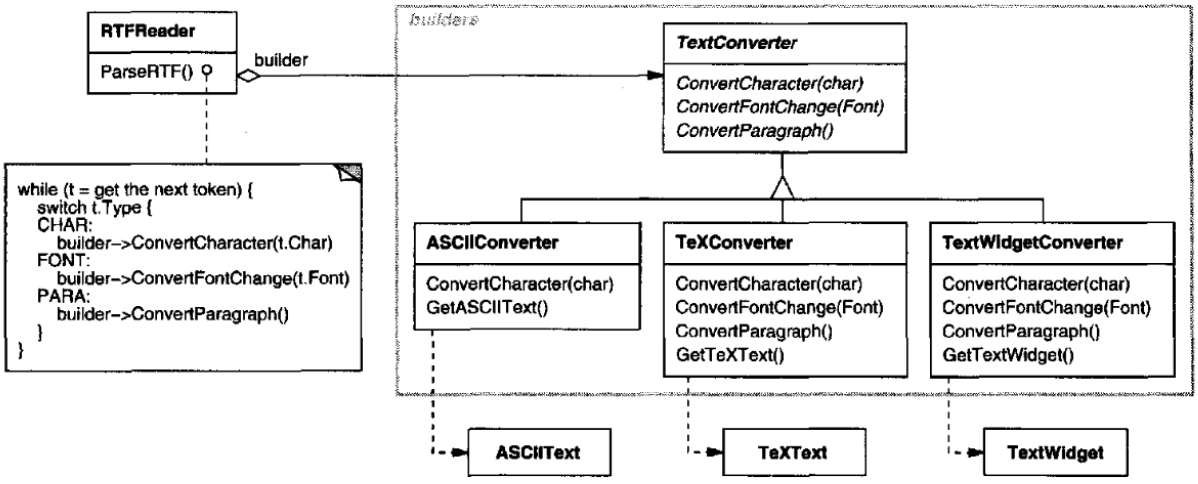
Builder

Su objetivo es simplificar la creación de objetos complejos al separar el proceso de construcción de la representación final del objeto. Usar el patrón Builder cuando:

- El algoritmo para crear un objeto complejo debe ser independiente de las partes y de como se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto construido.

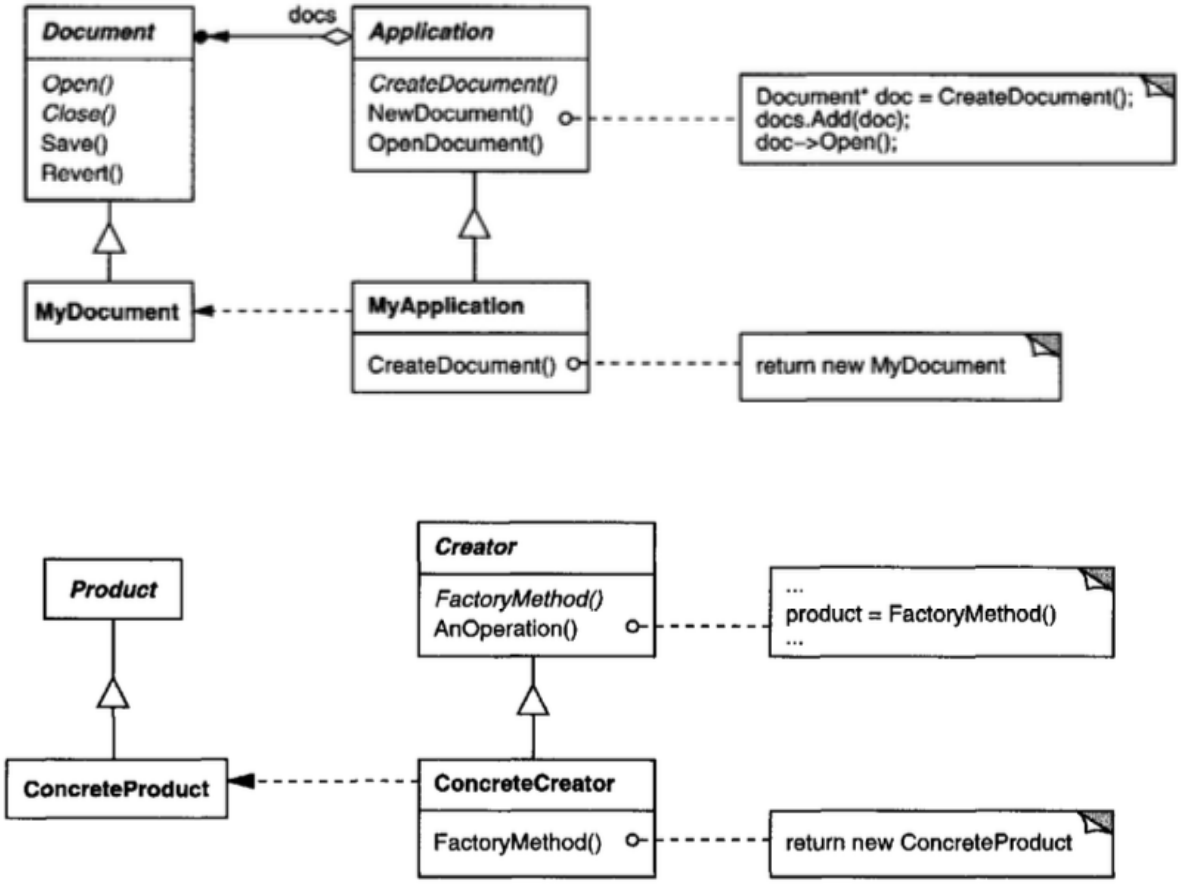
Builder: especifica una interfaz abstracta para crear partes del producto.
ConcreteBuilder: construyen y ensamblan partes del producto implementando Builder interfaz.
Director: construye el objeto utilizando Builder interfaz.
Product: representa el objeto bajo construcción.





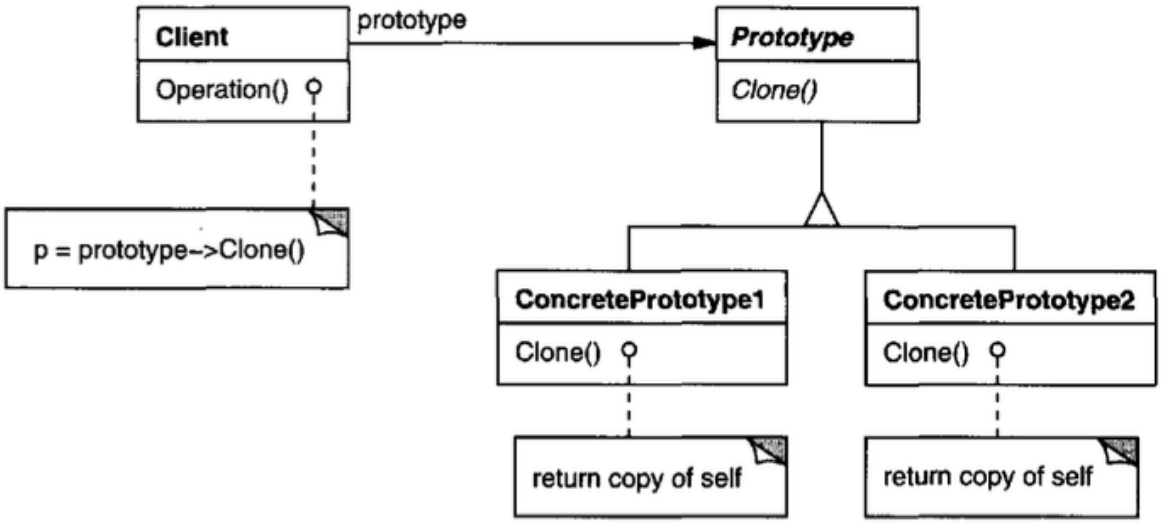
Factory

Su objetivo es encapsular la creación de objetos y proporcionar una interfaz común para la creación de diferentes tipos de objetos sin especificar su clase concreta. Se usa cuando tenemos un conjunto de clases relacionada que comparten una interfaz común, y queremos que la creación e objetos de estas clases sea flexible y desacoplada del código del cliente.



Product: define la interfaz de los objetos que crea el método de fabrica.
ConcreteProduct: implementa la interfaz producto.
Creator: declara el metodo de fabrica, que retorna un objeto de tipo producto.
ConcreteCreator: sobreescribe la instancia del método de fabrica y retorna el ConcreteProduct.

Elimina la necesidad de enlazar clases especificas el código solo interactúa con la interfaz Product.
La desventaja es que el cliente tiene que subclasificar la clase Creator solo paran particular ConcreteProduct.

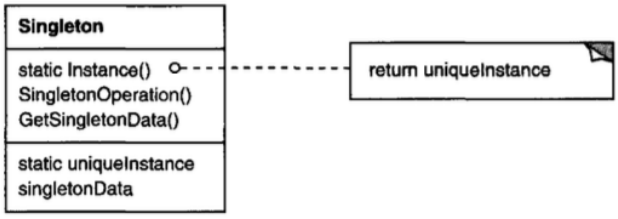


Prototype

Su objetivo es crear nuevos objetos duplicando un objeto existente llamado prototipo en lugar de crearlos desde 0.

Prototype: declara la interfaz para ser clonada.
ConcretePrototype: implementa la operación para la clonación.
Client: crea un nuevo objeto pidiendo al prototipo que se clone a si mismo.

Singleton

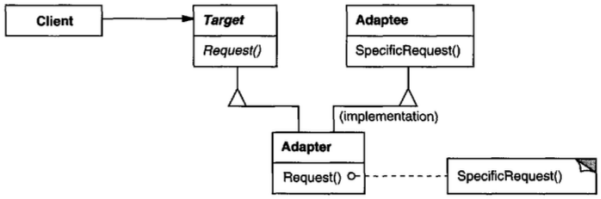


Patrones Estructurales

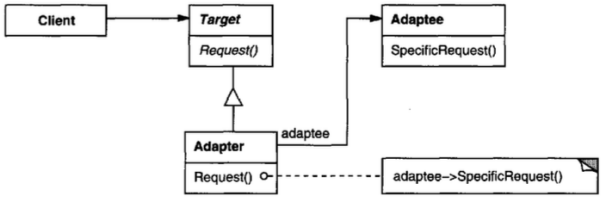
Adapter

Su objetivo es permitir que dos interfaces incompatibles trabajen juntas mediante la creación de un adaptador que actúa como intermediario entre ambas interfaces. Cuando tenemos dos clases o componentes con interfaces diferentes y queremos que interactúen entre si.

A class adapter uses multiple inheritance to adapt one interface to another:



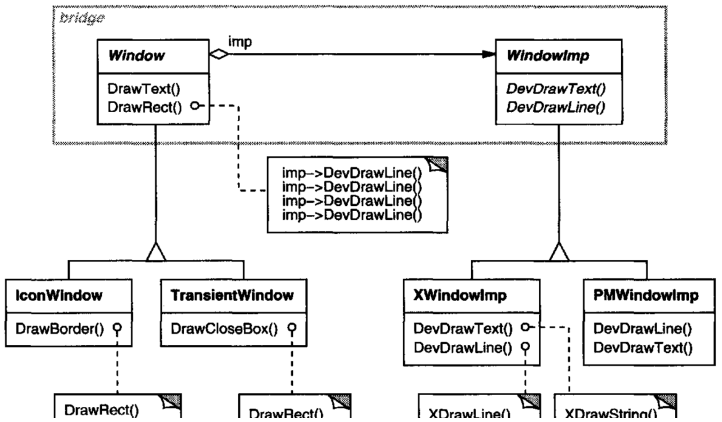
An object adapter relies on object composition:

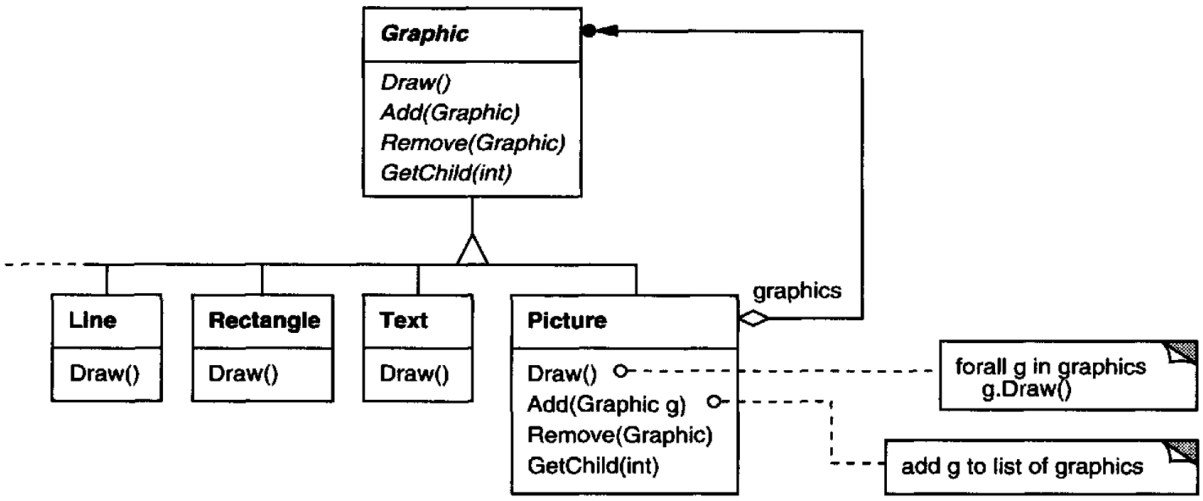


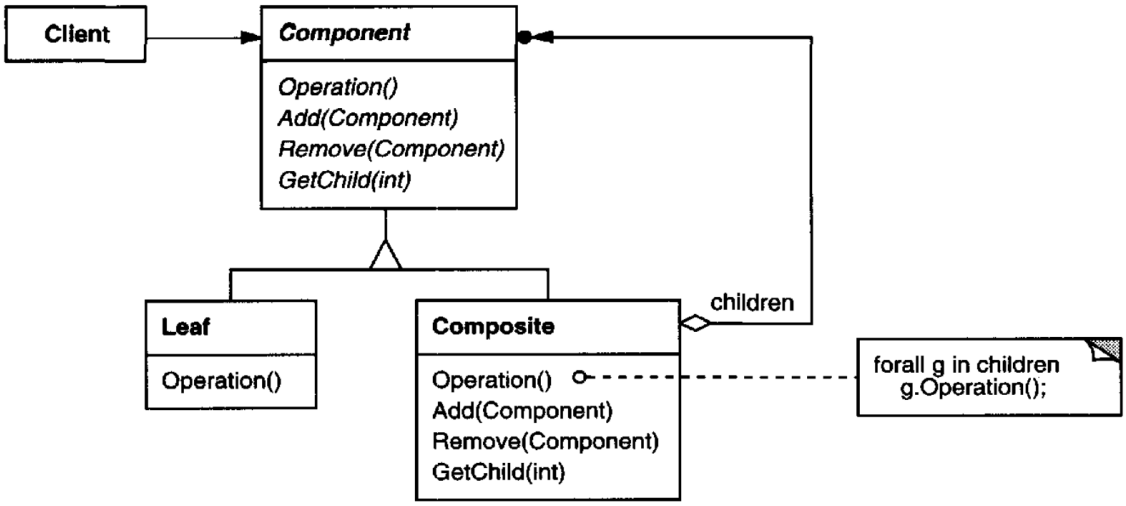
Target: define la interfaz especifica del dominio que utiliza el cliente.
Client: colabora con los objetos que se ajustan a la interfaz Target.
Adaptee: define una existente que necesita adaptación
Adapter: adapta la interfaz Adaptee a Target

Bridge

Desacoplar una abstracción de su implementación para que ambos puedan variar de forma diferente.







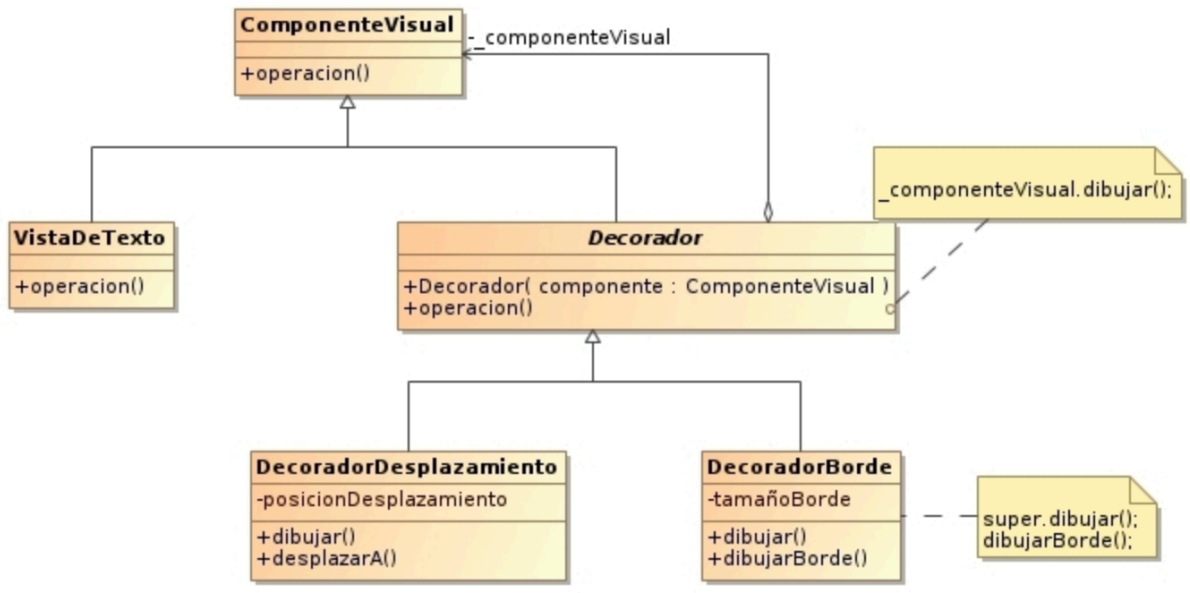
Composite

Componga objetos en estructura de arboles para representar jerarquías parciales completas. Permite a los clientes tratar objetos individuales y composiciones de objetos de forma uniforme. El patrón Composite se utiliza cuando se desea representar una estructura de árbol en la que los objetos individuales y las agrupaciones de objetos tienen la misma interfaz. Esto permite tratar a los objetos individuales y a los grupos de objetos de manera uniforme, lo que facilita la manipulación y la recursividad en la estructura.

Component: declara la interfaz de los objetos en la composición.
Leaf: representa objetos hoja en la composición. Leaf no tiene hijo.
Composite: define el comportamiento de los componentes que tienen hijo.
Client: manipula objetos de la composición mediante la interfaz Component.

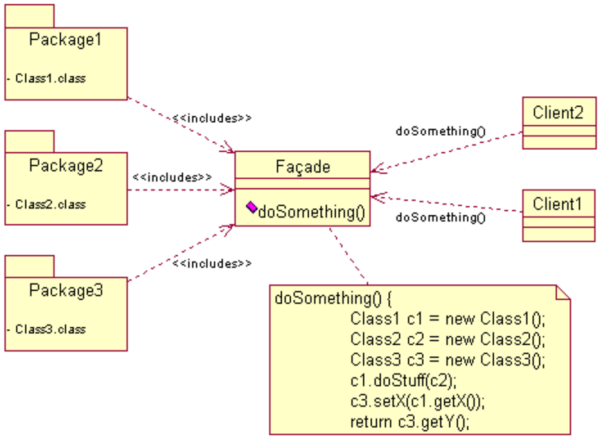
Decorator

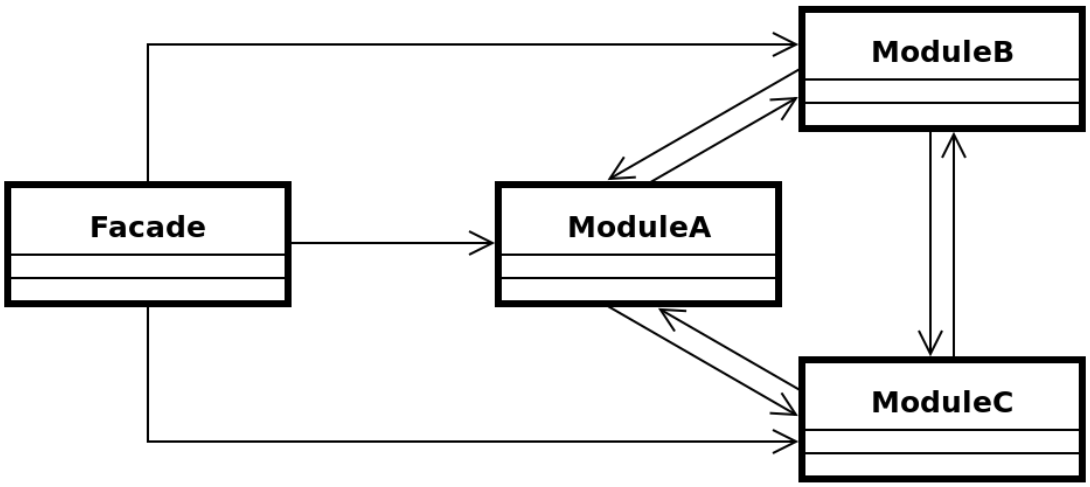
Responde a la necesidad de añadir dinámicamente funcionalidad a un objeto. Esto no spermite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino que la implementan y la asocian a la primera.



Facade

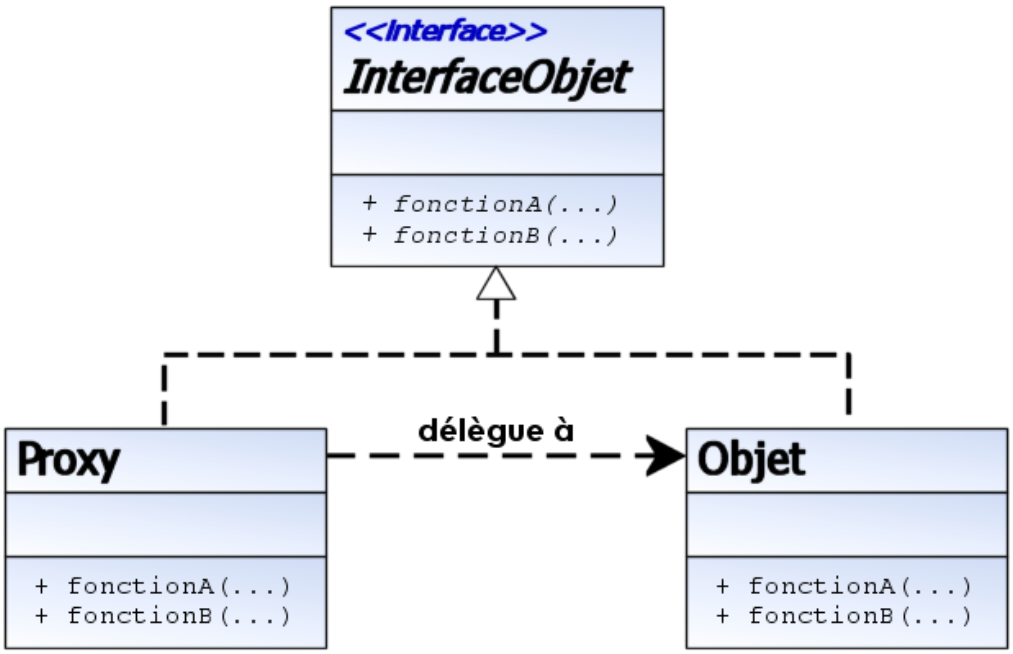
Viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división de subsistemas, minimizando las comunicaciones y dependencias entre estos. Se aplica el patron de Fachada cuando se necesite proporcionar una interfaz simple para un sistema complejo o cuando se quiera estructurar varios subsistemas en capas, las fachadas serian el punto de entrada a cada nivel.





Proxy

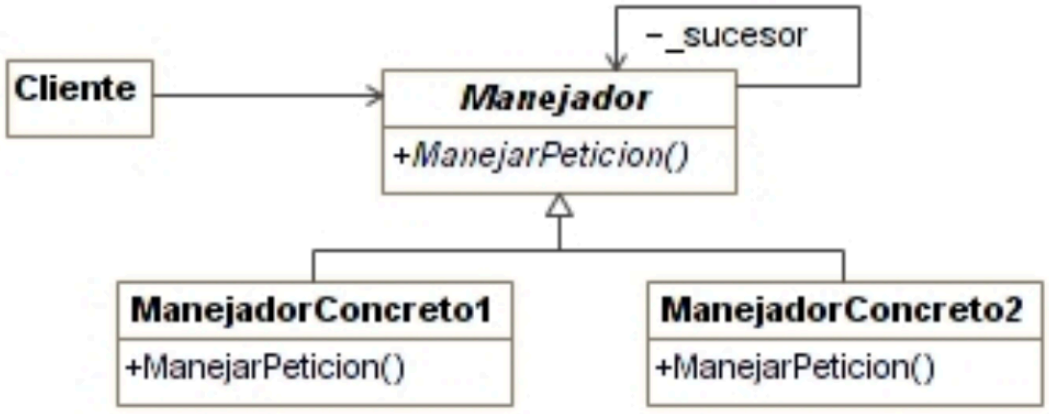
Tiene como propósito proporcionar un intermediario de un objeto para controlar su acceso. Es una clase que funciona como interfaz para otra cosa. El uso del proxy puede ser simplemente reenvío al objeto real o también puede proporcionar funcionalidad adicional: almacenamiento en cache cuando las operaciones en el objeto real llevan muchos recursos, o comprobación de condiciones previas antes de que se invoquen las operaciones en el objeto real.



Patrones de Comportamiento

Chain of Responsibility

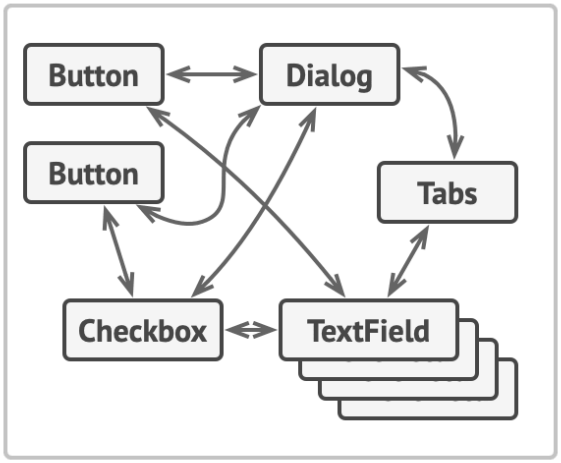
Se encadenan los receptores y pasa la petición a través de la cadena hasta que un objeto la toma.



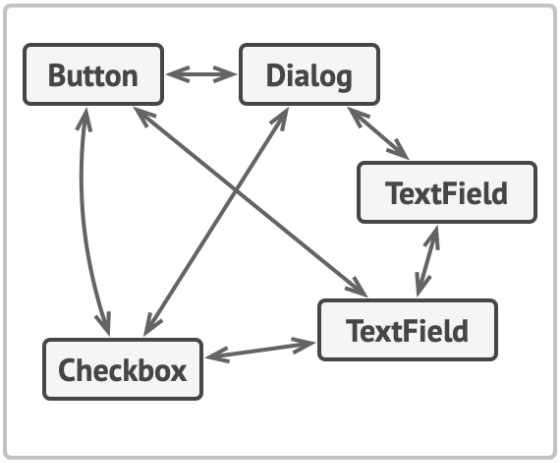
Mediator

Una clase que actua como mediadora en un conjunto de objetos que cumplen un determinado comportamiento.

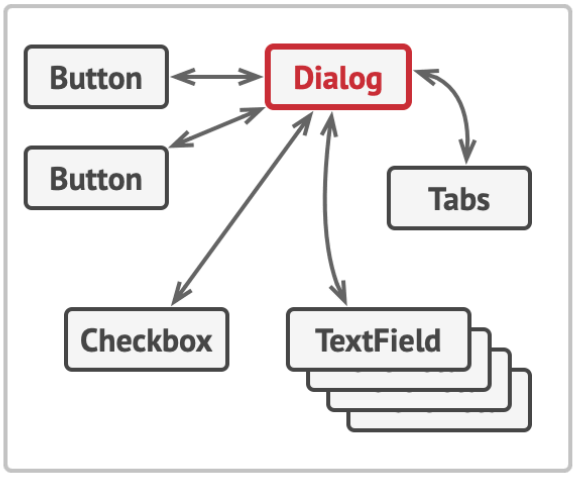
Diálogo de Perfil



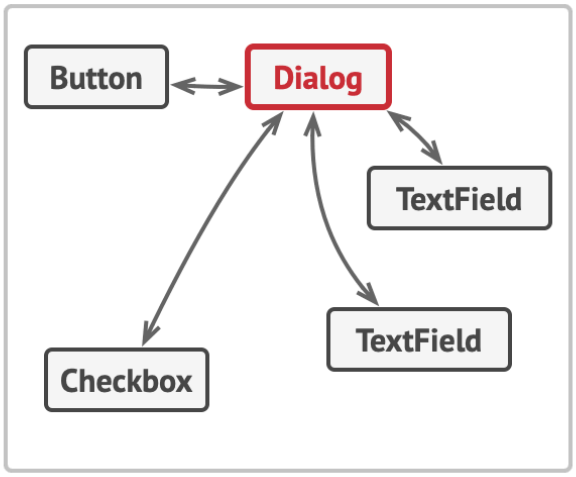
Diálogo de Inicio de Sesión



Diálogo de Perfil



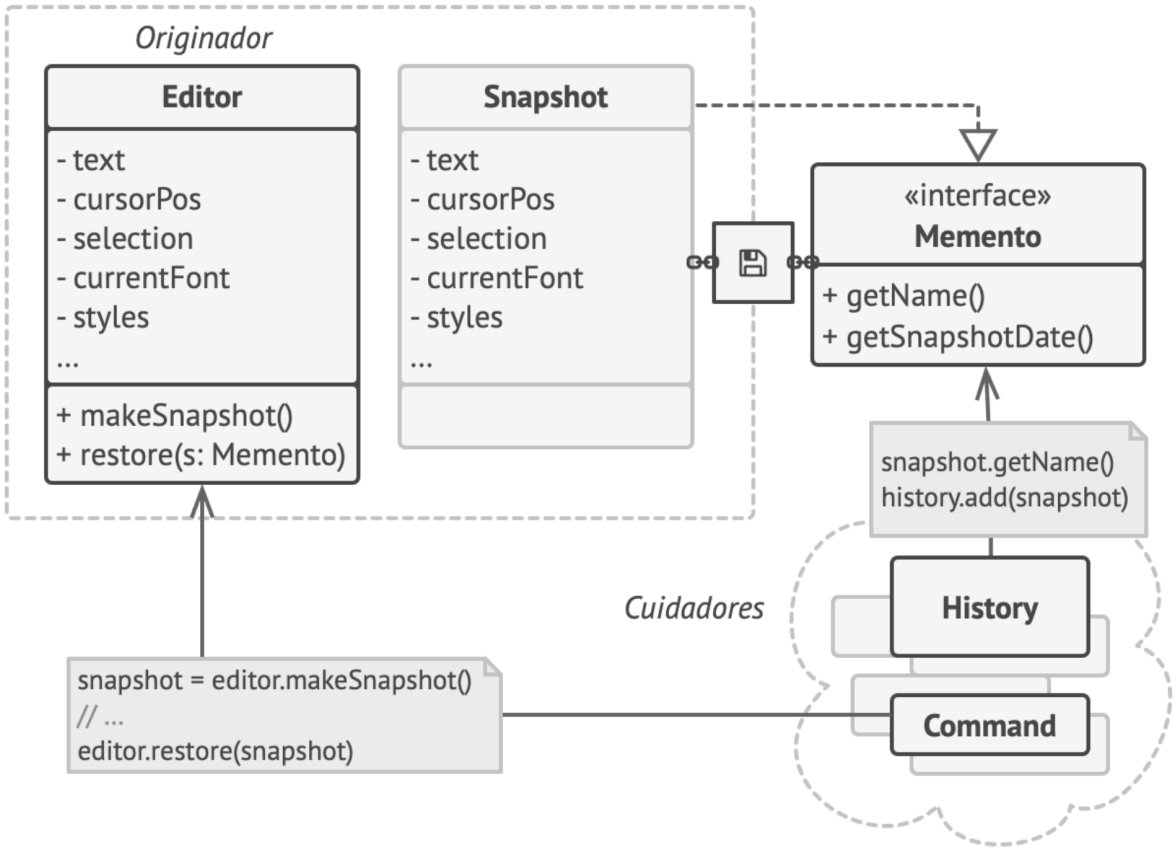
Diálogo de Inicio de Sesión



Memento

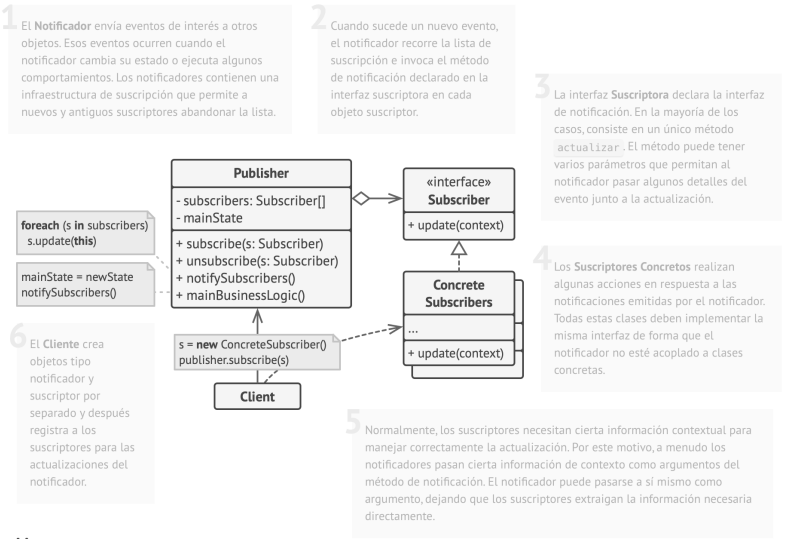
Te permite guardar y restaurar el estado previo de un objeto sin revelar detalles de implementación.

El patron sugiere almacenar la copia del estado del objeto en un objeto llamado “Memento”. Los objetos no pueden ser accedido por ningún objeto excepto el que lo produjo.



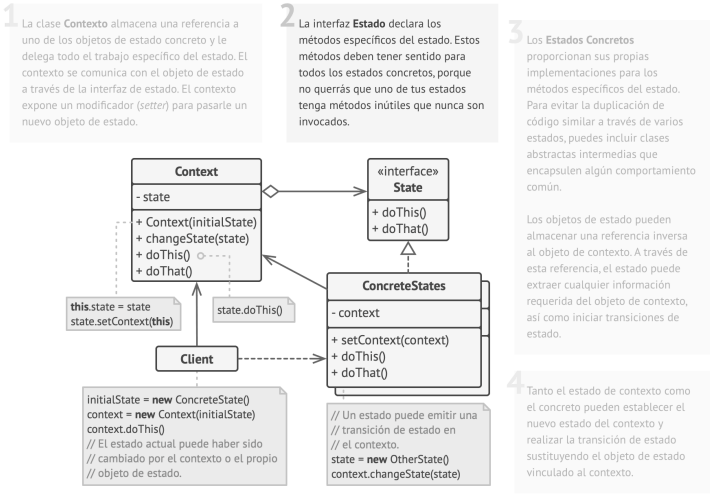
Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



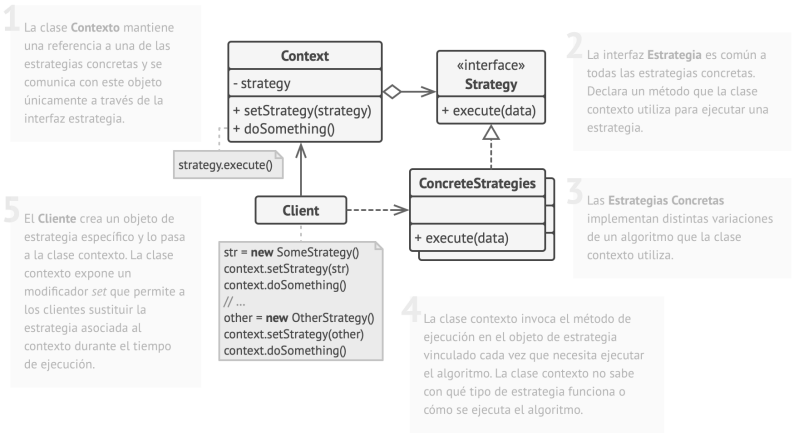
State

Permite al objeto cambiar su comportamiento cuando su estado interno cambia.



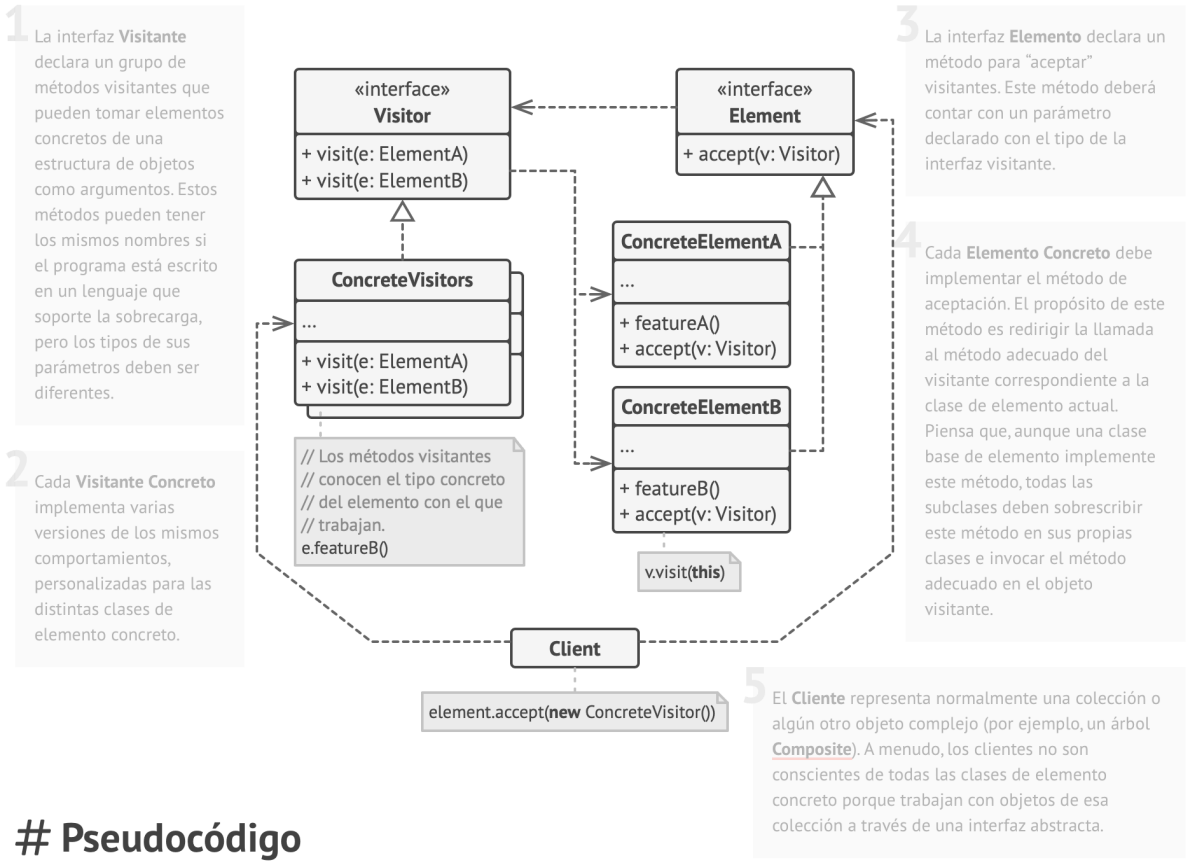
Strategy

Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.



Visitor

Permite separar algoritmos de los objetos que los operan.



Pseudocódigo

Anti Patrones

Los patrones nos ofrecen una forma de resolver un problema típico, los antipatrones muestran formas de enfrentarse a problemas con consecuencias negativas conocidas. Los antipatrones se basan en la idea de que puede resultar más fácil detectar a priori fallos en el desarrollo del proyecto que elegir el camino correcto o, lo que es lo mismo, descartar las alternativas incorrectas nos puede ayudar a la elección de la mejor alternativa.
El estudio de los antipatrones es muy útil porque sirve para no escoger malos caminos en el desarrollo de sistemas, teniendo para ello una base documental y así evitar usar simplemente la intuición. Además proporciona una denominación común a problemas que facilita la comunicación entre diferentes desarrolladores.

The Blob

Se encuentra en diseños donde una clase monopoliza el procesamiento y otras clases simplemente encapsulan información. Este antipatron es caracterizado por una gran clase compleja y alrededor simples clases que guardan datos.

Lava Flow (Código muerto)

Sucede cuando un software es entregado antes de ser completado o antes de ser completamente probado y al ser expuesto es imposible cambiar sus características.

Golden Hammer

Proyecto que se ensalza de manera exagerada, se predice que resolverá multiples problemas que ni siquiera esta hecha para resolver.

Spaghetti Code

Software que tiene una estructura de control de flujo compleja e incomprensible.