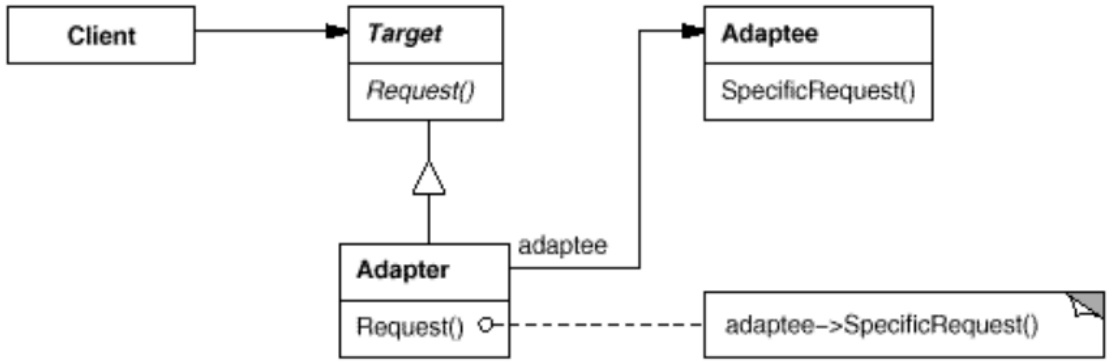
PATRONES DE DISEÑO

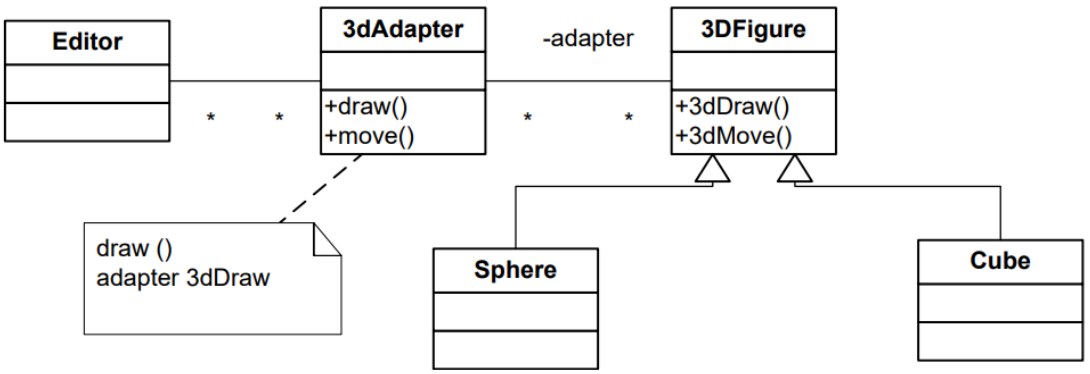
ADAPTER

. Intención: “Convertir” la interfaz de una clase en otra que el cliente espera. EL Adapter permite que ciertas clases trabajen en conjunto cuando no podrían por tener interfaces incompatibles.
. Aplicabilidad: Use el adapter cuando ud quiere usar una clase existente y su interfaz no es compatible con lo que precisa.
. Estructura:



. Participantes:

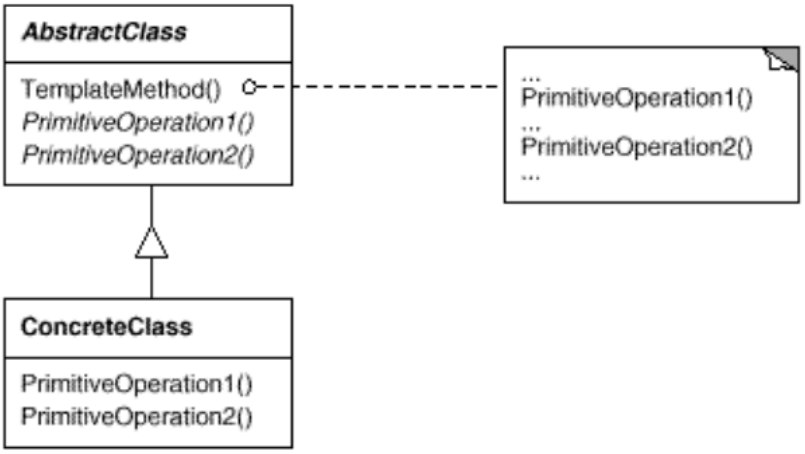
Target (Figura): Define la interfaz específica que usa el cliente.
Client (Editor): Colabora con objetos que satisfacen la interfaz de Target.
Adaptee (3DFigure): Define una interfaz que precisa ser adaptada.
Adapter (3DAdapter): Adapta la interfaz del Adaptee a la interfaz de Target.



TEMPLATE METHOD

. Intención: Definir el esqueleto de un algoritmo en un método, difiriendo algunos pasos a las subclases. El template method permite que las subclases redefinan ciertos aspectos de un algoritmo sin cambiar su estructura.
. Aplicabilidad: Para implementar las partes invariantes de un algoritmo una vez y dejas que las sub-clases implementen los aspectos que varían.

. Estructura:



Abstraer cada "parte" del código de las sub-clases que difiere -Expresarlo como un método más primitivo -Definir el algoritmo en la clase abstracta y expresar la variabilidad en los métodos primitivos en cada sub-clase.

COMPOSITE

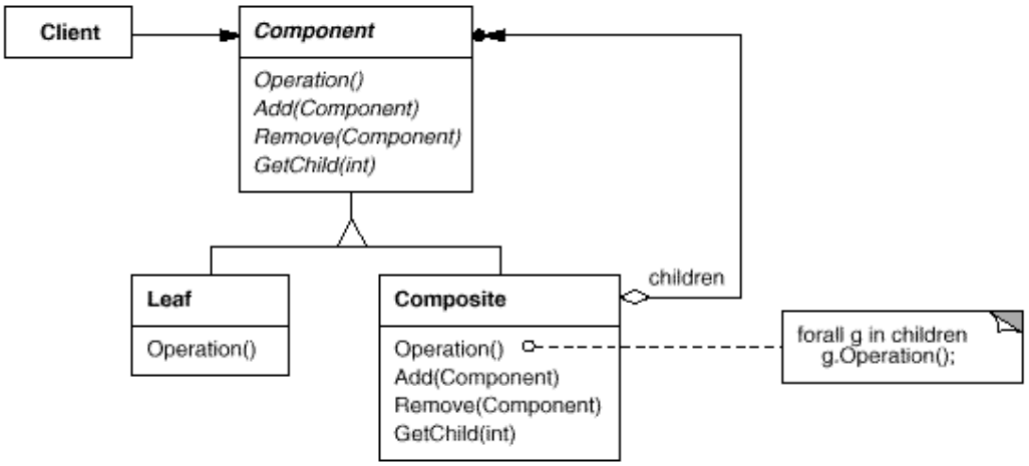
. Propósito: Componer objetos en estructuras de árbol para representar jerarquías parte-todo. El Composite permite que los clientes traten a los objetos atómicos y a sus composiciones uniformemente.

. Aplicabilidad: Use el patrón Composite cuando:

Quiere representar jerarquías parte-todo de objetos.

Quiere que los objetos "clientes" puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes trataran a los objetos atómicos y compuestos uniformemente.

. Estructura:



. Participantes:

Component (Figure):

Declara la interfaz para los objetos de la composicion.

Implementa comportamientos default para la interfaz común a todas las clases.

Declara la interfaz para definir y acceder “hijos”.
(opcional) define una interfaz para para acceder el “padre” de un componente en la estructura recursiva y la implementa si es apropiado.

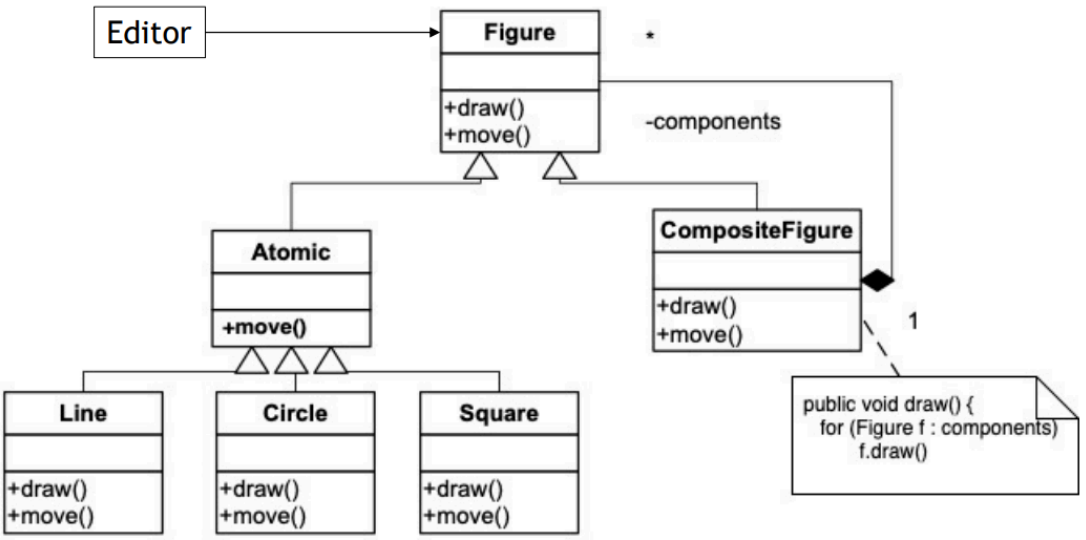
Leaf (Rectangle, Line, Text, etc.):

Representa arboles “hojas” en la composición. Las hojas no tienen “hijos”.
Define el comportamiento de objetos primitivos en la composición.

Composite (CompositeFigure):

Define el comportamiento para componentes con “hijos”.
Contiene las referencias a los “hijos”.
Implementa operaciones para manejar “hijos”.

. Ejemplo:



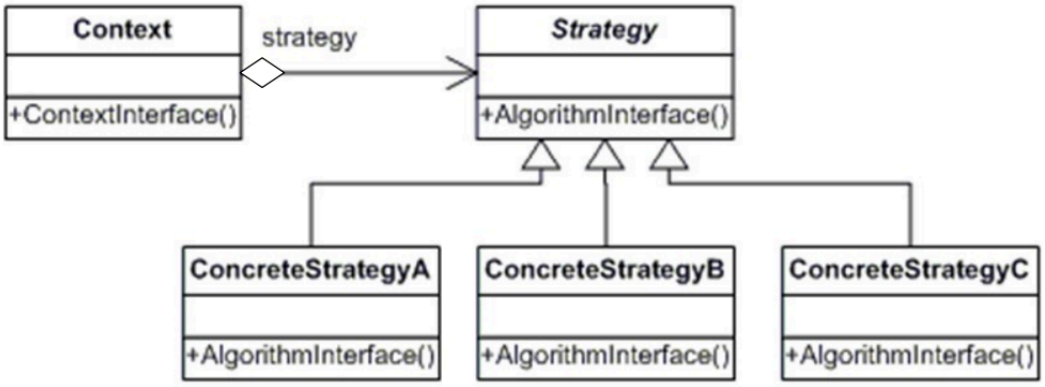
STRATEGY

. Propósito: definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables. El Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan.
Desacoplar un algoritmo del objeto que lo utiliza.
Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica.
Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.
. Aplicabilidad: Existen muchos algoritmos para llevar a cabo una tarea. No es deseable codificarlos todos en una clase y seleccionar cual utilizar por medio de sentencias condicionales. Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas y difíciles de mantener. Es necesario cambiar el algoritmo en forma dinámica.

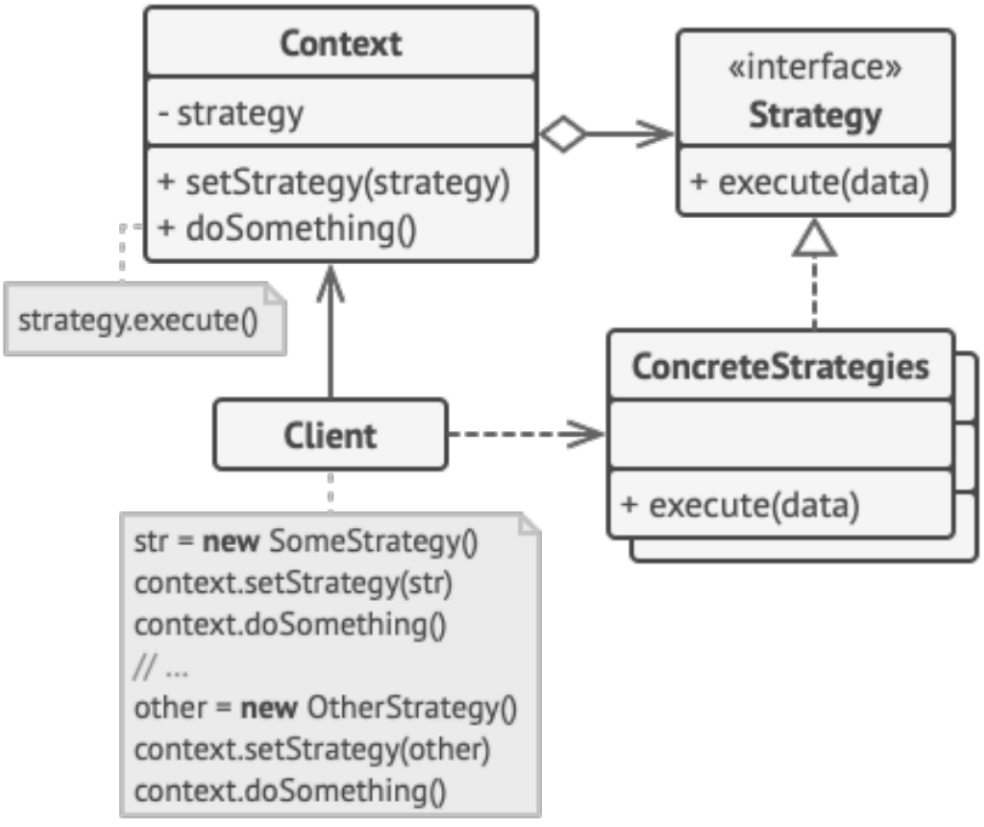
. Solución-Estructura:

1- Definir una familia de algoritmos, encapsular cada uno en un objeto y hacerlos intercambiables.

2- Son los clientes del contexto los que generalmente crean las estrategias.



. Ejemplo:



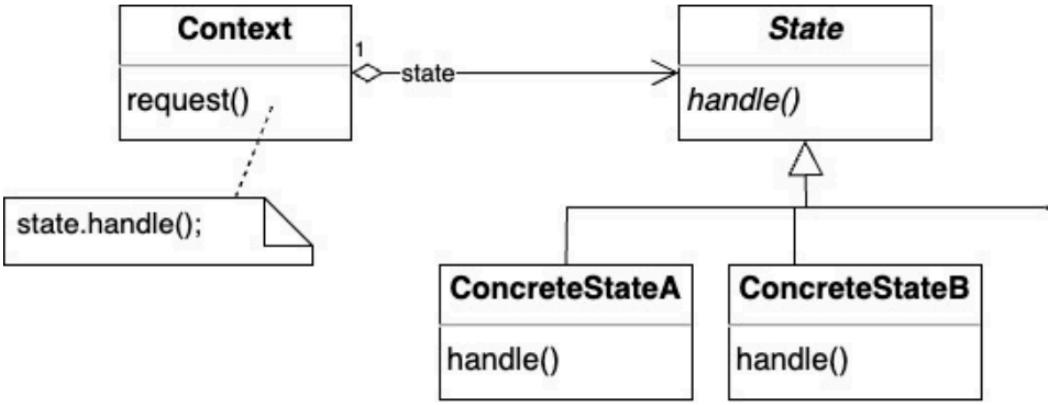
STATE

. Propósito: Modificar el comportamiento de un objeto cuando su estado interno se modifica. Externamente parecería que la clase del objeto ha cambiado.

. Aplicabilidad: Usamos el patrón State cuando:

El comportamiento de un objeto depende del estado en el que se encuentre.
Los métodos tienen sentencias condicionales complejas que dependen del estado. Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional. El patrón State reemplaza el condicional por clases.

. Estructura:



. Participantes:

• Context (Alarm): Define la interfaz que conocen los clientes y mantiene una instancia de alguna clase de ConcreteState que define el estado corriente.
• State (AlarmState): Define la interfaz para encapsular el comportamiento de los estados de Context.
• ConcreteState subclases (Active, Inactive,Sleeping): Cada subclase implementa el comportamiento respecto al estado específico.

STATE VS. STRATEGY

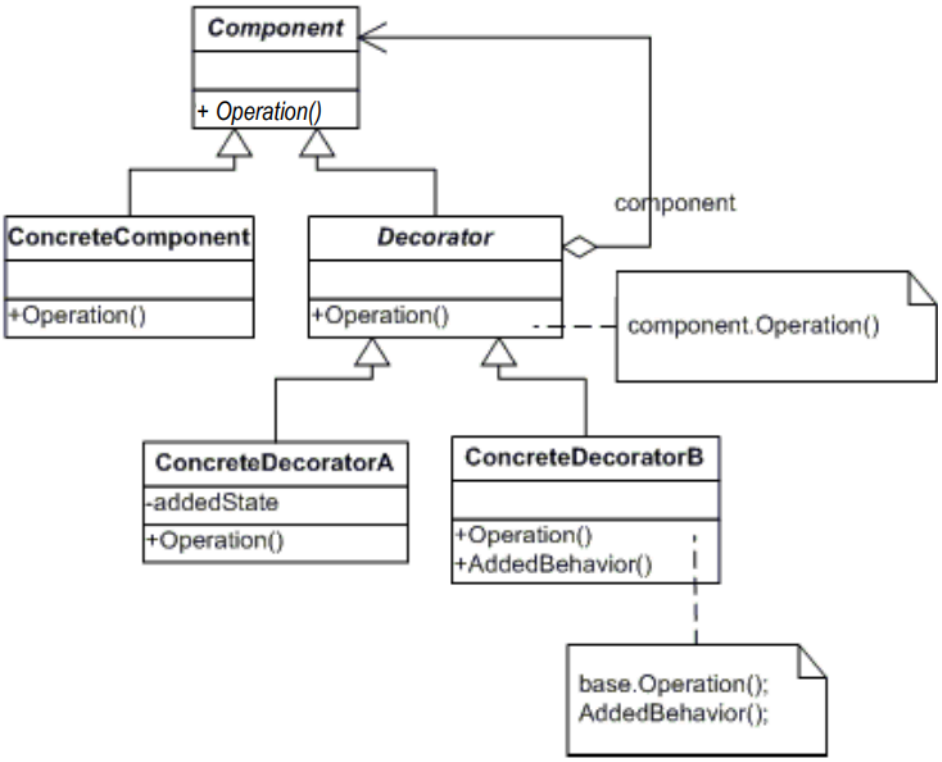
. El patrón STATE es útil para una clase que debe realizar transiciones entre estados fácilmente.
En State, los diferentes estados son internos al contexto, no los eligen las clases clientes sino que la transición se realiza entre los estados mismos.
El estado es privado del objeto, ningún otro objeto sabe de él.

. El patrón STRATEGY es útil para permitir que una clase delegue la ejecución de un algoritmo a una instancia de una familia de estrategias.
En Strategy, las diferentes estrategias son conocidas desde afuera del contexto, por las clases clientes del contexto.
El contexto del Strategy debe contener un mensaje público para cambiar el ConcreteStrategy, ya que es seteado por el cliente.

DECORATOR

. Objetivo: Agregar o quitar comportamiento a un objeto dinámicamente y en forma transparente.

. Estructura:



ADAPTER VS. DECORATOR

. Adapter: **convierte** la interface del objeto para el cliente. No se anidan.
. Decorator: **preserva** la interface del objeto para el cliente. Pueden y suelen anidarse.

STRATEGY VS. DECORATOR

. Se parecen en cuanto a su propósito ya que ambos permiten que un objeto cambie su funcionalidad dinámicamente.
. Se diferencian en cuanto a su estructura: el Strategy cambia el algoritmo por dentro del objeto, el Decorator lo hace por fuera.

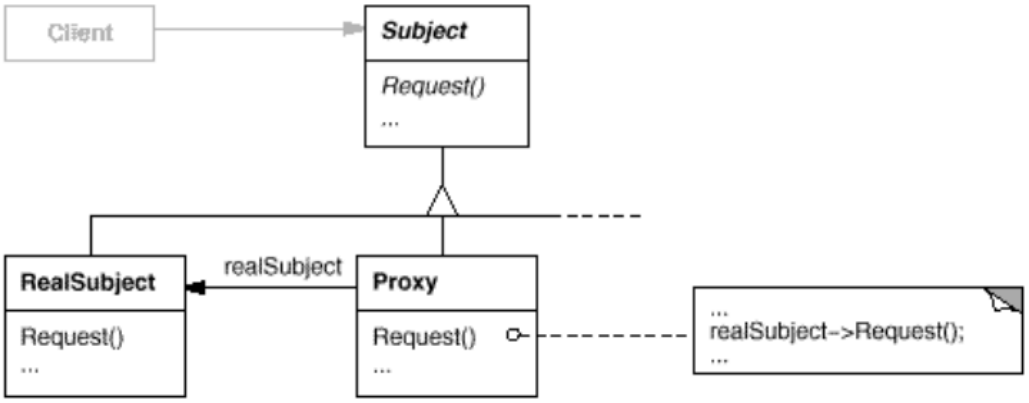
PROXY

. Propósito: proporcionar un intermediario de un objeto para controlar su acceso.
. Aplicabilidad: cuando se necesita una referencia a un objeto más flexible o sofisticado.

- Virtual proxy: demorar la construcción de un objeto hasta que sea realmente necesario, cuando sea poco eficiente acceder al objeto real.
- Protection proxy: Restringir el acceso a un objeto por seguridad.

• Remote proxy: representar un objeto remoto en el espacio de memoria local. Es la forma de implementar objetos distribuídos. Estos proxies se ocupan de la comunicación con el objeto remoto, y de serializar/deserializar los mensajes y resultados.

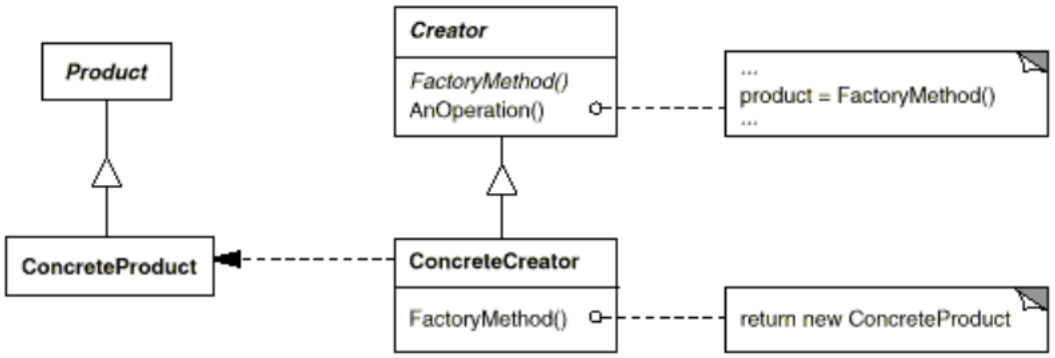
. Estructura:



FACTORY METHOD

. Intención: Define una “interface” para la creación de objetos, mientras permite que subclases decidan qué clase se debe instanciar.

. Estructura:



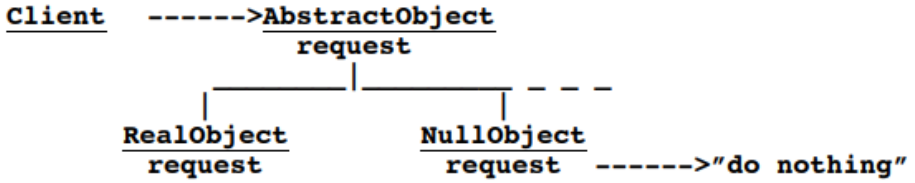
. Participantes (Roles):

- Product: Define la interface de los objetos creados por el “factory method”.
- Concrete Product: Implementa la interface definida por el Product.
- Creator: Declara el “factory method” (abstracto o con comportamiento default).
- ConcreteCreator: Implementa el “factory method”.

Define una “interface” de 1 método crearPoliza(), para construir diferentes cosas relacionadas (en un dominio). Cada versión del método tiene que estar en una clase creadoras (polimorfismo).

NULL OBJECT

. Objetivo: Proporciona un sustituto para otro objeto que comparte la misma interfaz pero no hace nada. El NullObject encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores.
Elimina todos los condicionales que verifican si la referencia a un objeto es NULL. Hace explícito elementos del dominio que hacen “nada”.
. Estructura:

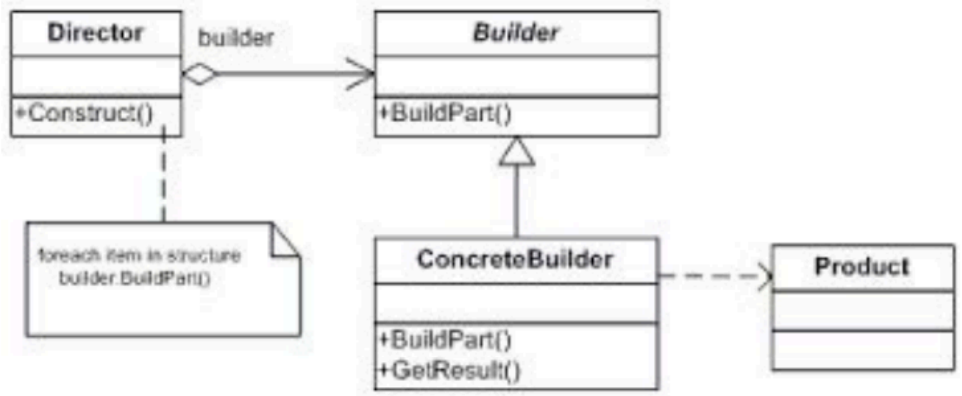


BUILDER

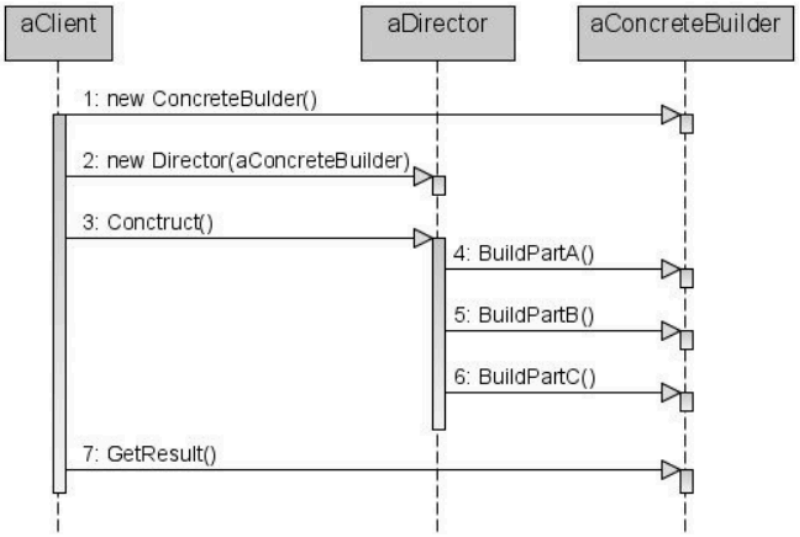
. Intención: separa la construcción de un objeto complejo de su representación (implementación) de tal manera que el mismo proceso puede construir diferentes representaciones (implementaciones).
Configurar mismo objeto de distintas formas (armar por partes).
. Participantes:

- Builder: especifica una interface abstracta para crear partes de un Producto.
- Concrete Builder: construye y ensambla partes del producto. Guarda referencia al producto en construcción.
- Director: conoce los pasos para construir el objeto. Utiliza el Builder para construir las partes que va ensamblando.
- Product: es el objeto complejo a ser construido.

. Estructura:



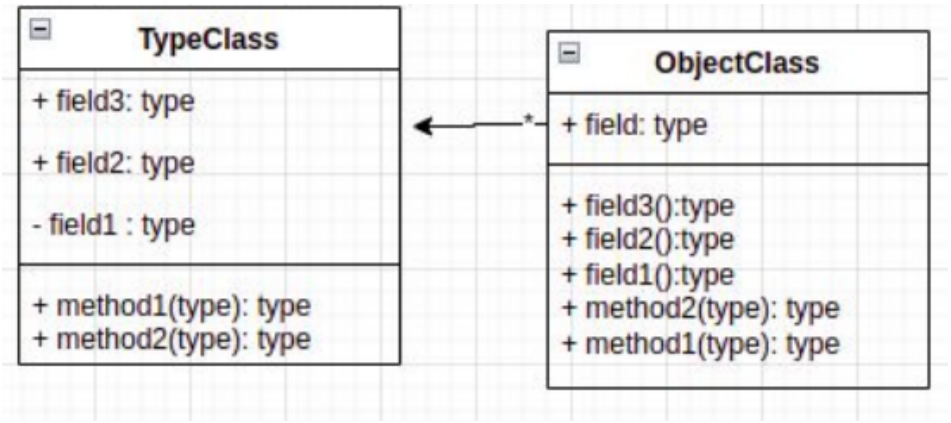
. Diagrama de secuencia:



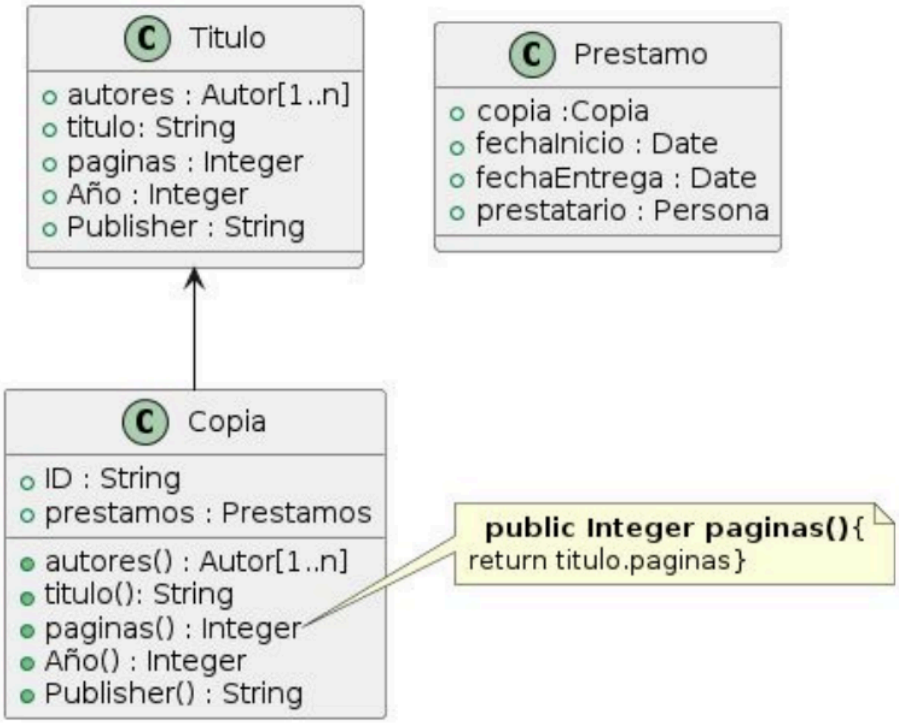
TYPE OBJECT

. Intención: “Desacoplar las instancias de sus clases para que esas clases se puedan implementar como instancias de un clase”. Traducción: Identificar elementos del dominio que actúan como clases/metadata de otros elementos del dominio.

. Estructura:



. Ejemplo: En una biblioteca el manejo de libros con sus copias…



TEST DOUBLE

. Objetivo: Crear un objeto que es una maqueta (polimórfica) del objeto o módulo requerido. Utilizar la maqueta según se necesite.

. Rangos de implementacion:

- Test Stub: recibe los mensajes, no hace nada.
- Test Spy: guarda registro de mensajes recibidos.
- Mock Object: comprueba la validez de los mensajes recibidos.
- Fake Object: simula el comportamiento en tiempo y forma.

REFACTORING

CODE SMELLS

. Duplicate Code
. Large Class
. Long Method
. Data Class
. Feature Envy
. Long Parameter List
. Switch Statements
. Reinventar la rueda
. Temporary Field

REFACTORING CATALOGUE

. Composición de método:
Extract Method, Replace Temp with Query, Split Temporary Variable, Substitute Algorithm.

. Mover aspectos entre objetos:
Move Method, Move Field, Extract class.

. Organización de datos:
Self Encapsulate Field, Encapsulate Field / Collection, Replace Data Value with Object, Replace Array with Object.

. Simplificación de expresiones condicionales:
Replace Conditional with Polymorphism.

. Simplificación en la invocación de métodos:
Rename Method.

. Manipulación de la generalización:
Pull Up/Push Down.

. Form Template Method:
1) Encontrar el método que es similar en todas las subclases y extraer sus partes en: métodos idénticos (misma signatura y cuerpo en las subclases) o métodos únicos (distinta signatura y cuerpo).
2) Aplicar “Pull Up Method” para los métodos idénticos.
3) Aplicar “Rename Method” sobre los métodos únicos hasta que el método similar quede con cuerpo idéntico en las subclases.
4) Compilar y testear después de cada “rename”.
5) Aplicar “Rename Method” sobre los métodos similares de las subclases (esqueleto).
6) Aplicar “Pull Up Method” sobre los métodos similares.
7) Definir métodos abstractos en la superclase por cada método único de las subclases.

. Replace Conditional Logic with Strategy:

1) Crear una clase Strategy.
2) Aplicar “Move Method” para mover el cálculo con los condicionales del contexto al strategy.
    1) Definir una v.i. en el contexto para referenciar al strategy y un setter (generalmente el constructor del contexto).
    2) Dejar un método en el contexto que delegue.
    3) Elegir los parámetros necesarios para pasar al strategy (el contexto entero? Sólo algunas variables? Y en qué momento?).
    4) Compilar y testear.
3) Aplicar “Extract Parameter” en el código del contexto que inicializa un strategy concreto, para permitir a los clientes setear el strategy. - Compilar y testear.
4) Aplicar “Replace Conditional with Polymorphism” en el método del Strategy.
5) Compilar y testear con distintas combinaciones de estrategias y contextos.

. Replace Altern-State Conditional with State:

1. Aplicar “Replace Type-Code with Class” para crear una clase que será la superclase del State a partir de la v.i. que mantiene el estado.
2. Aplicar “Extract Subclass” [F] para crear una subclase del State por cada uno de los estados de la clase contexto.
3. Por cada método de la clase contexto con condicionales que cambiar el valor del estado, aplicar “Move Method” hacia la superclase de State.
4. Por cada estado concreto, aplicar “Push down method” para mover de la superclase a esa subclase los métodos que producen una transición desde ese estado. Sacar la lógica de comprobación que ya no hace falta.
5. Dejarlos estos métodos como abstractos en la superclase o como métodos por defecto.

. Replace Embellishment with Decorator:

1. Identificar la superclase (or interface) del objeto a decorar (clase Component del patrón). Si no existe, crearla.
2. Aplicar Replace Conditional Logic with Polymorphism (crea decorator como subclase del decorado).
Alcanza? Si no sigo
3. Aplicar Replace Inheritance with Delegation (decorator delega en decorado como clase “hermana”).
4. Aplicar Extract Parameter en decorator para asignar decorado.

FRAMEWORK

FROZENSPOTS VS. HOTSPOTS

. Frozenspots: aspectos que no podemos cambiar.
. Hotspots: El framework ofrece puntos de extensión que nos permiten introducir variantes y así construir aplicaciones diferentes.

CAJA BLANCA VS. CAJA NEGRA

Los puntos de extensión pueden implementarse en base a herencia o en base a composición:
. A los frameworks que utilizan herencia en sus puntos de extensión, les llamamos de Caja Blanca (Whitebox).
. A los que utilizan composición les llamamos de Caja Negra (blackbox).

VENTAJAS Y DESVENTAJAS HERENCIA - COMPOSICIÓN

Las ventajas que ofrece la herencia son que la subclasificación resulta sencilla y que los métodos de los pasos en este caso se ejecutan dentro de la misma clase. La desventaja es la gran cantidad de código duplicado.
Por otra parte, las ventajas que ofrece la composición son la reducción en cantidad de clases y código, la simplicidad de agregar un nuevo componente y la posibilidad de modificar dinámicamente los componentes de los objetos sin tener que instalar uno nuevo.
La desventaja se comprende en el armado de los objetos y en que en este caso el objeto Robot debe pasarse a sí mismo como parámetro a otros objetos en cada paso.

DATOS IMPORTANTES HERENCIA

. La plantilla se implementa en una clase abstracta y los ganchos en sus subclases (como el clásico patrón Método Plantilla).
. Es más simple para casos con pocas alternativas y/o pocas combinaciones
. Al implementar los métodos gancho puedo utilizar las variables de instancia y todo el comportamiento heredado de la clase abstracta
. Si hay muchas variantes o combinaciones, empiezo a tener muchas clases y duplicación de código

DATOS IMPORTANTES COMPOSICIÓN

. Un objeto implementa la plantilla y delega la implementación de los ganchos en sus partes.
. Evita la duplicación de código y el creciente número de clases cuando existen muchas alternativas y combinaciones posibles
. Al implementar los métodos gancho debo pasar como parámetro todo lo que necesiten - no tienen acceso a las variables de instancia del objeto
. Puedo cambiar el comportamiento en tiempo de ejecución sin mayor dificultad.

INVERSIÓN DE CONTROL

La inversión de control es la característica principal de la arquitectura run-time de un framework. Esta característica permite que los pasos canónicos de procesamiento de la aplicación (comunes a todas las instancias) sean especializados por objetos tipo manejadores de eventos a los que invoca el framework como parte de su mecanismo reactivo de despacho.