# Orientación a Objetos 2

## Resumen patrones de diseño

# Adapter

## Intent

"Convertir" la interfaz de una clase en otra que el cliente espera

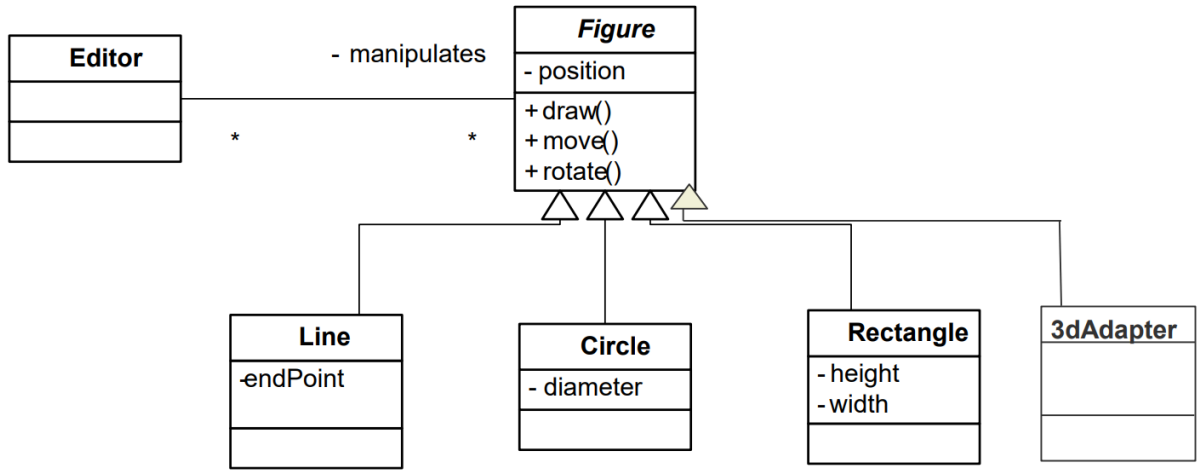
El Adapter permite que ciertas clases trabajen en conjunto cuando no podrian por tener interfaces incompatibles

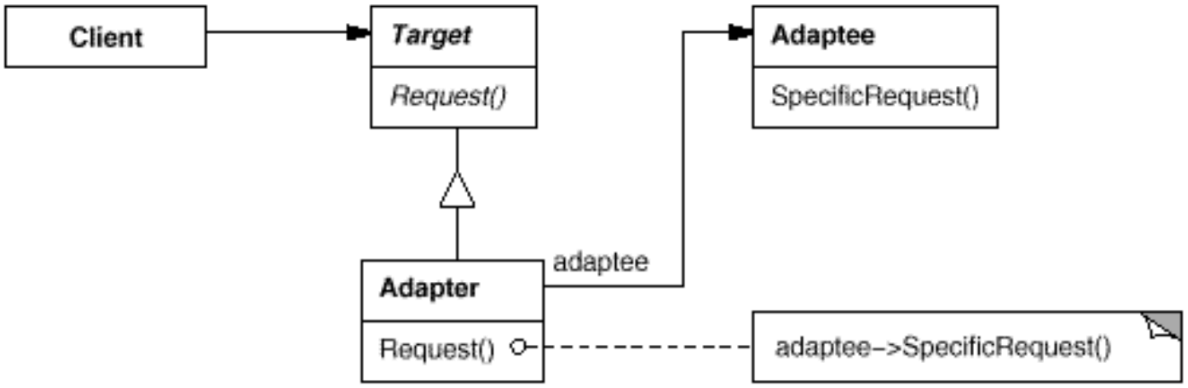
## Applicability

Usar Adapter cuando:

- Quiere usar una clase existente y su interfaz no es compatible con lo que precisa

## Structure






## Participants

- Target (Figura):
    - Define la interfaz especìfica 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

## Intent

Definir el esqueleto de un algoritmo en un método, difiriendo algunos pasos a las subclases

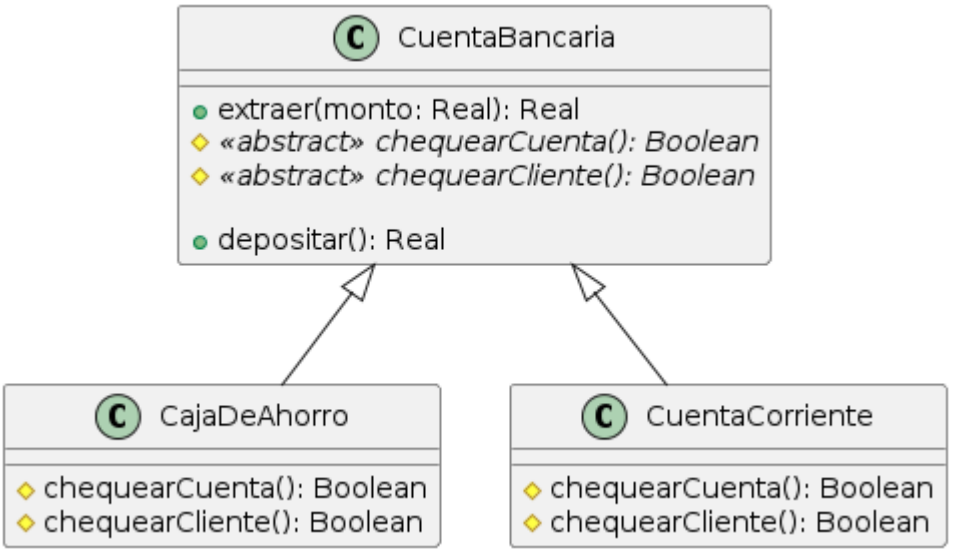
El template method permite que las subclases re definas ciertos aspectos de un algoritmo sin cambiar su estructura

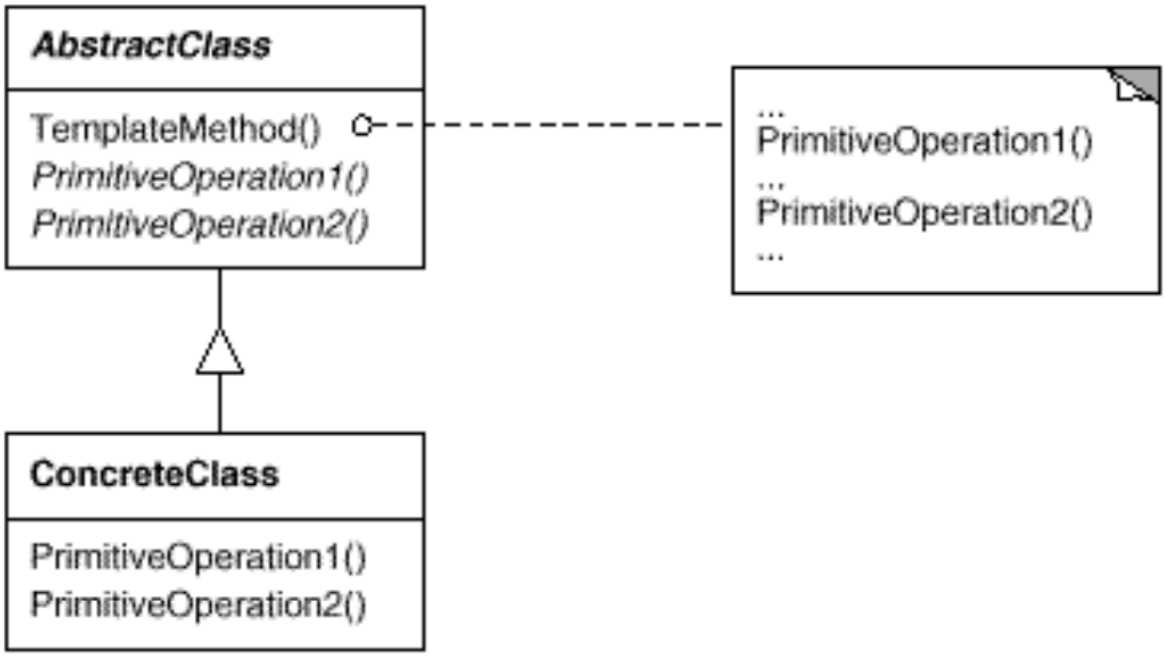
## Applicability

Usar Template Method para:

- Implementar las partes invariantes de un algoritmo una vez y dejar que las sub-clases implementen los aspectos que varían

## Structure






## Participants

- AbstractClass (CuentaBancaria):
    - Define el template method
    - Implementa las partes invariantes del mismo
    - Define métodos abstractos para las partes que deben ser redefinidas
- ConcreteClass (CajaDeAhorro/CuentaCorriente):
    - Implementa los métodos abstractos como sea necesario

# Composite

## Intent

Componer objetos en estructuras de árbol para representar jerarquías parte-todo

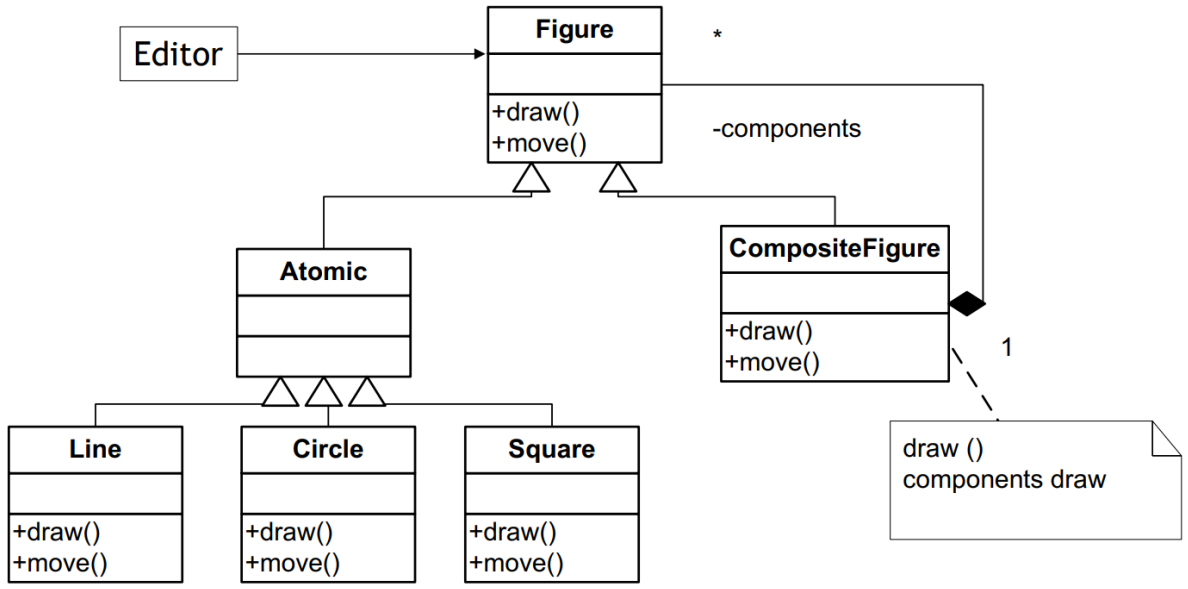
El Composite permite que los clientes traten a los objetos atómicos y a sus composiciones uniformemente

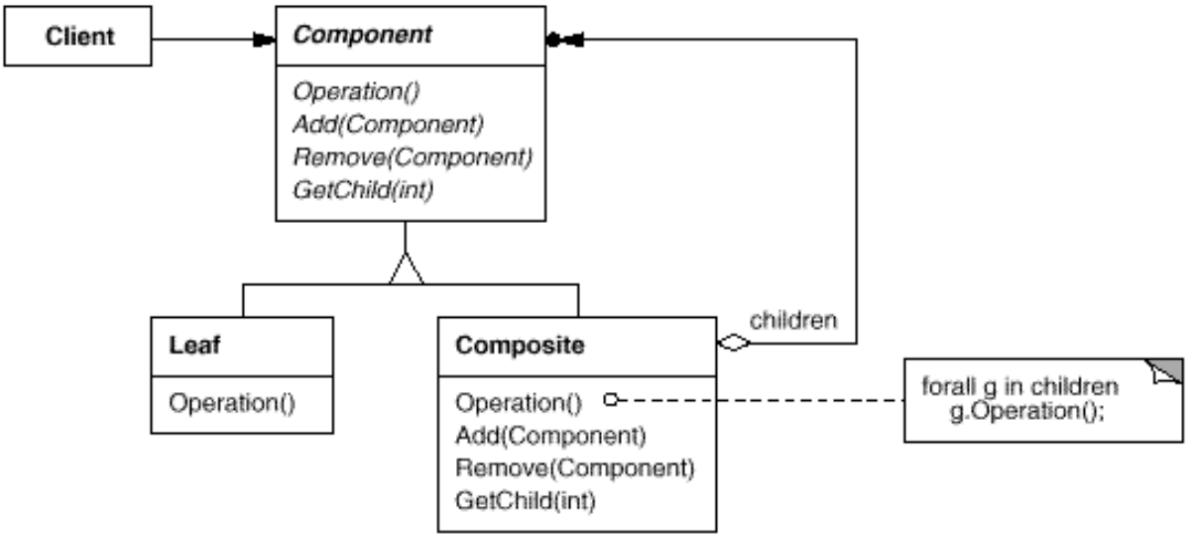
## Applicability

Usar Composite cuando:

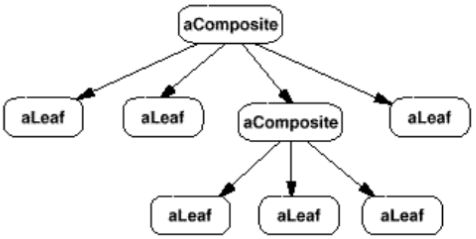
- Quiere representar jerarquías parte-todo de objetos
- Quiere que los objetos “clientes” puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes tratarán a los objetos atómicos y compuestos uniformemente

## Structure






**Una instancia típica se ve así**




## Participants

- Component (Figure):
  - Declara la interfaz para los objetos de la composición
  - 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 árboles “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”

## Consequences

- Positivas:
    - Define jerarquías de clases consistentes de objetos primitivos y compuestos. Los objetos primitivos pueden componerse en objetos complejos, los que a su vez pueden componerse recursivamente
    - Simplifica los objetos cliente. Los clientes usualmente no saben (y no deberían preocuparse) acerca de si están manejando un compuesto o un simple
    - Hace más fácil el agregado de nuevos tipos de componentes porque los clientes no tienen que cambiar cuando aparecen nuevas clases componentes
- Negativa:
    - No permite restringir las estructuras de composición (cuando ciertos compuestos pueden armarse solo con cierto tipo de atómicos)

## Implementation

Cuestiones a tener en cuenta:

- Referencias explícitas al padre (cuidado en mantener consistencia)
- Maximizar el protocolo de la clase/interfaz Component
- Transparencia vs. seguridad
- Orden de los hijos
- Caching
- Borrado de componentes
- Diferentes estructuras para guardar componentes

# Strategy

## Intent

Desacoplar un algoritmo del objeto que lo utiliza

Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica

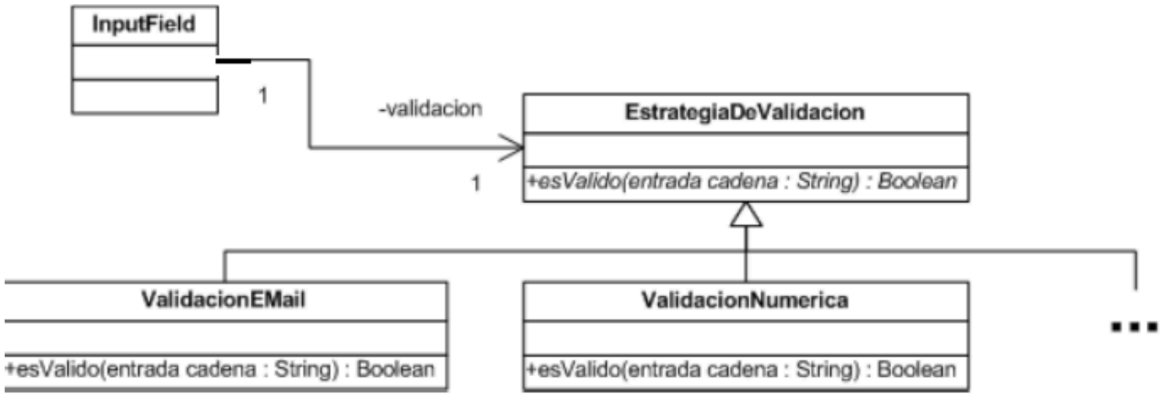
Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada

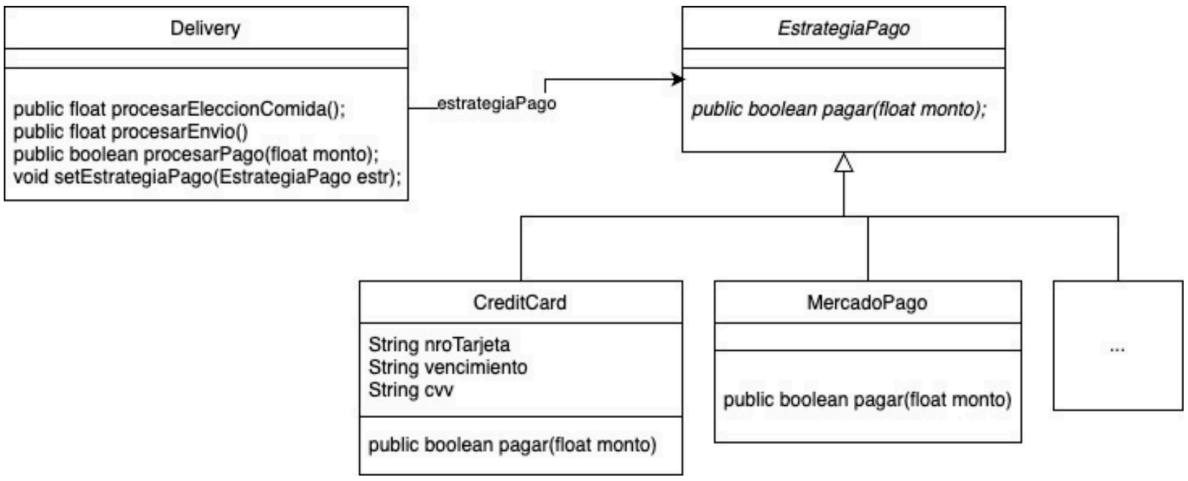
## Applicability

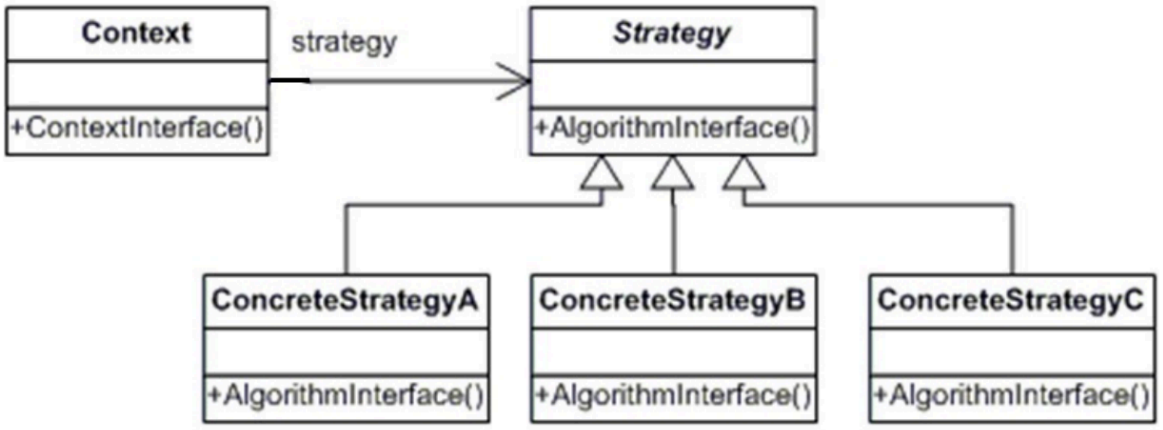
Usar Strategy cuando:

- Existen muchos algoritmos para llevar a cabo una tarea
- No es deseable codificarlos todos en una clase y seleccionar cuál 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

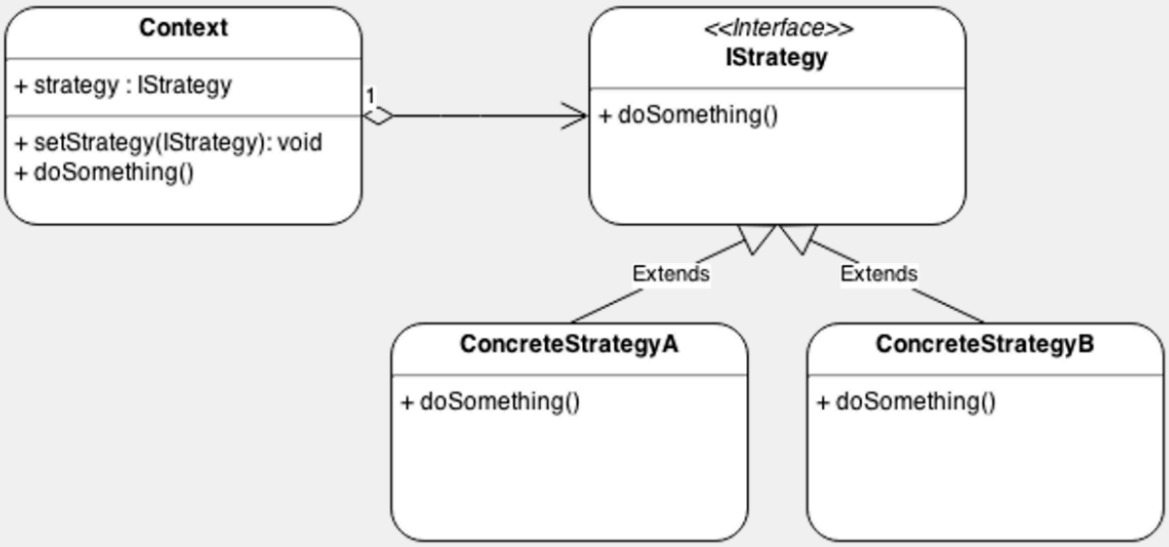
## Structure






## Strategy pattern – Class diagram



## Participants

- Context (Delivery/InputField):
    - Mantiene una instancia de tipo Strategy
    - Se configura con un objeto ConcreteStrategy
- Strategy (EstrategiaDePago/EstrategiaDeValidacion):
    - Define la interfaz para interactuar con las ConcreteStrategy
- ConcreteStrategy(CreditCard/MercadoPago/ValidacionEMail/ValidacionNumerica):
    - Implementa los métodos de Strategy con la lógica correspondiente

## Consequences

- Positivas:
    - Alternativa a subclasificar el contexto, para permitir que se pueda cambiar dinámicamente
    - Desacopla al contexto de los detalles de implementación de las estrategias
    - Se eliminan los condicionales
- Negativas:
    - Overhead en la comunicación entre contexto y estrategias
    - Los clientes deben conocer las diferentes estrategias para poder elegir

## Implementation

Cuestiones a tener en cuenta:

- El contexto debe tener en su protocolo métodos que permitan cambiar la estrategia
- Parámetros entre el contexto y la estrategia

# State

## Intent

Modificar el comportamiento de un objeto cuando su estado interno se modifica

Externamente parecería que la clase del objeto ha cambiado

## Applicability

Usar 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

## Details

Desacoplar el estado interno del objeto en una jerarquía de clases

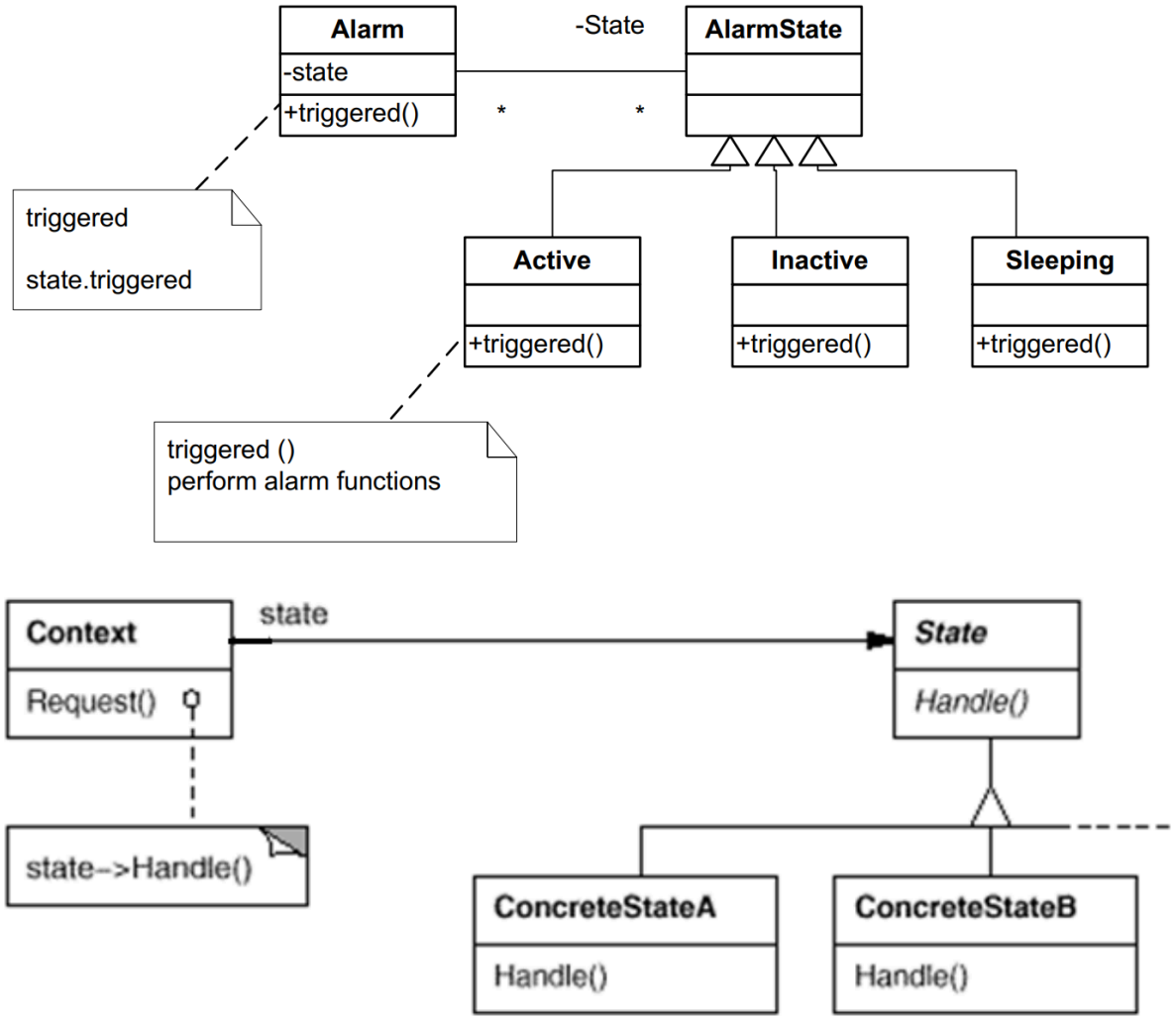
Cada clase de la jerarquía representa un estado concreto en el que puede estar el objeto

Todos los mensajes del objeto que dependan de su estado interno son delegados a las clases concretas de la jerarquía (polimorfismo)

Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional

El patrón State reemplaza el condicional por clases (es un uso inteligente del polimorfismo)

# Structure




# Participants

- Context (Alarm):
  - Define la interfaz que conocen los clientes
  - 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 (Active, Inactive,Sleeping):
  - Cada subclase implementa el comportamiento respecto al estado específico

## Consequences

- Positivas:
    - Localiza el comportamiento relacionado con cada estado
    - Las transiciones entre estados son explícitas
    - En el caso que los estados no tengan variables de instancia pueden ser compartidos
- Negativa:
    - En general hay bastante acoplamiento entre las subclases de State porque la transición de estados se hace entre ellas, por lo que deben conocerse entre sí

# Decorator

## Intent

Agregar comportamiento a un objeto dinámicamente y en forma transparente

## Motivation (Problem)

Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia

El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”

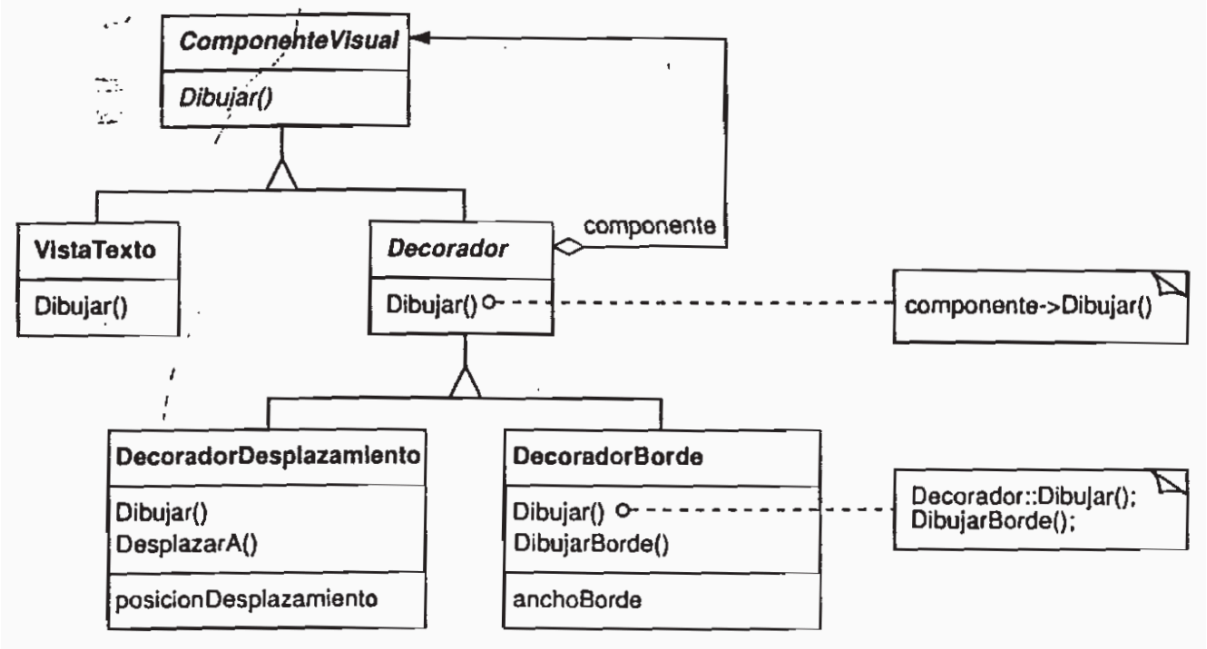
El problema que tiene la herencia es que se decide estáticamente

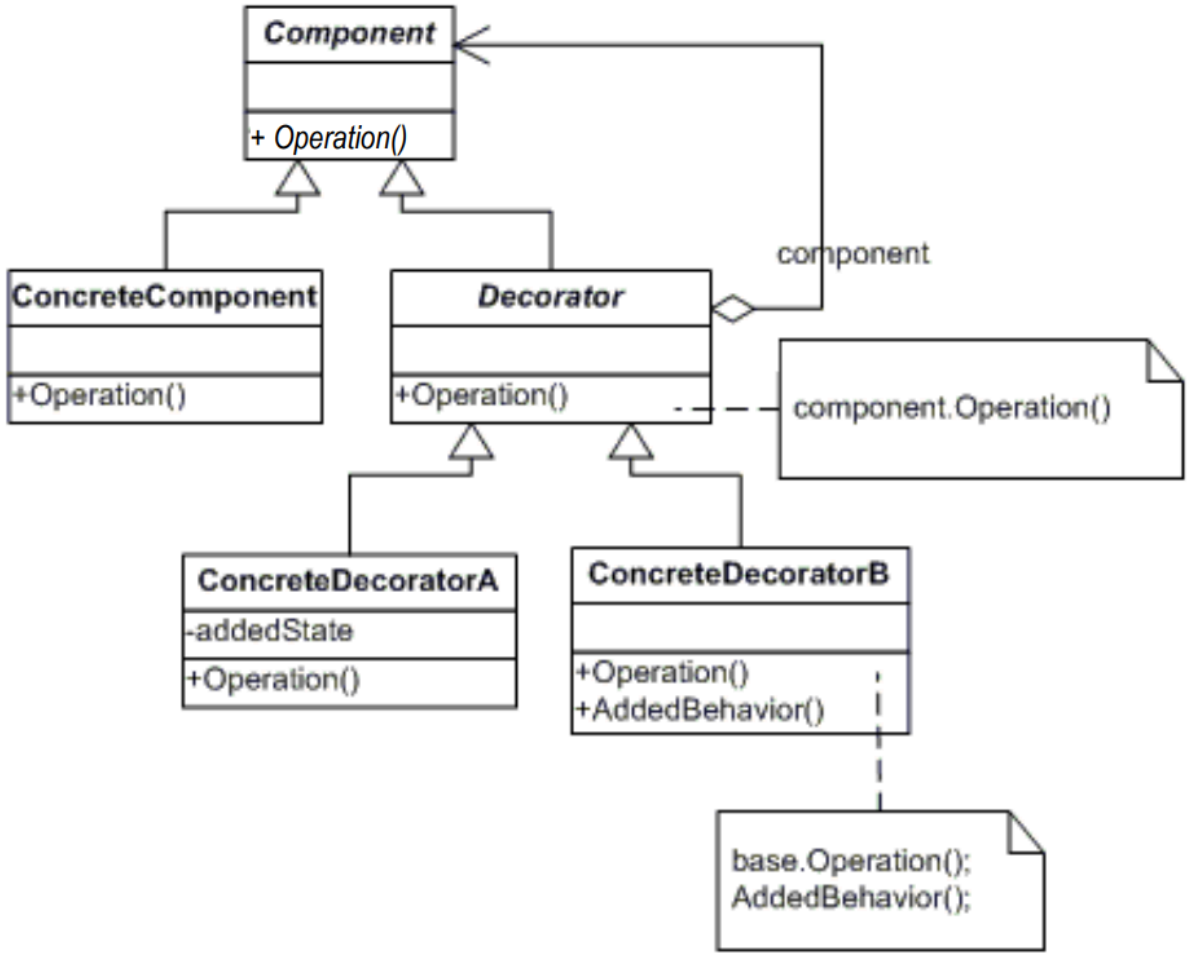
## Applicability

Usar Decorator para:

- Agregar responsabilidades a objetos individualmente y en forma transparente (sin afectar otros objetos)
- Quitar responsabilidades dinámicamente
- Cuando subclasificar es impráctico

# Structure


ComponenteVisual
Dibujar()
VistaTexto
Dibujar()
Decorador
Dibujar()
componente
componente->Dibujar()
DecoradorDesplazamiento
Dibujar()
DesplazarA()
posicionDesplazamiento
DecoradorBorde
Dibujar()
DibujarBorde()
anchoBorde
Decorador::Dibujar();
DibujarBorde();


Component
+ Operation()
component
ConcreteComponent
+Operation()
Decorator
+Operation()
component.Operation()
ConcreteDecoratorA
-addedState
+Operation()
ConcreteDecoratorB
+Operation()
+AddedBehavior()
base.Operation();
AddedBehavior();

## Participants

- Component (ComponenteVisual):
    - Define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente
- ConcreteComponent (VistaTexto):
    - Define un objeto al que se pueden añadir responsabilidades adicionales
- Decorator (Decorador):
    - Mantiene una referencia a un objeto componentes define una interfaz que se ajusta a la interfaz del componente
- ConcreteDecorator (DecoradorDesplazamiento/DecoradorBorde):
    - Añade responsabilidades al componente

## Consequences

- Positivas:
    - Permite mayor flexibilidad que la herencia
    - Permite agregar funcionalidad incrementalmente
- Negativas:
    - Mayor cantidad de objetos, complejo para depurar

## Implementation

Cuestiones a tener en cuenta:

- Misma interfaz entre componente y decorador
- No hay necesidad de la clase Decorator abstracta
- Cambiar el “skin” vs cambiar sus “guts”

# Proxy

## Intent

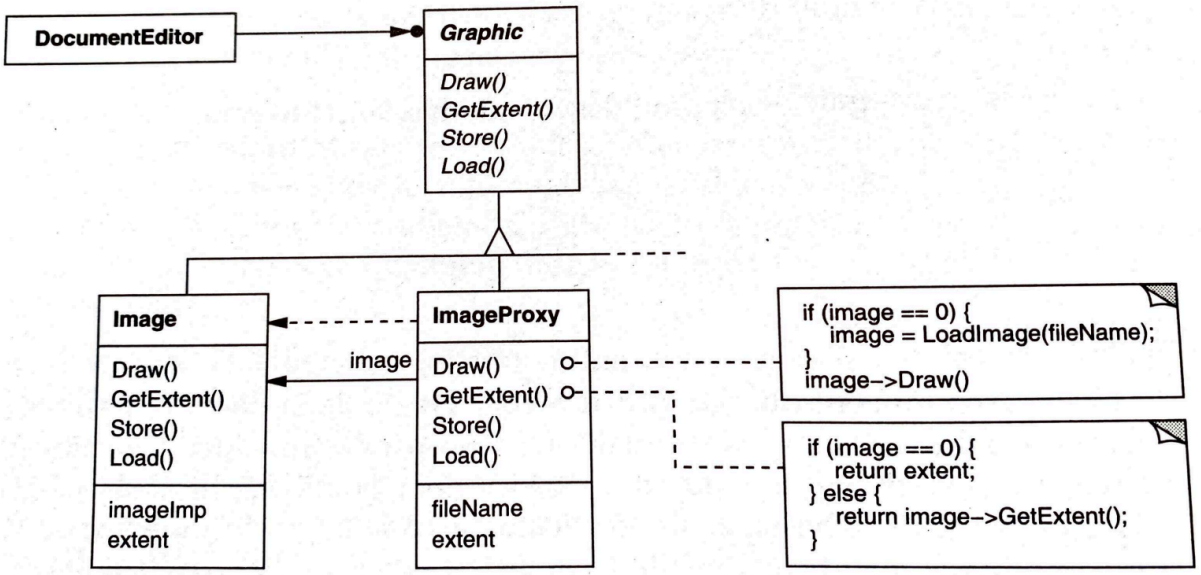
Proporcionar un intermediario de un objeto para controlar su acceso

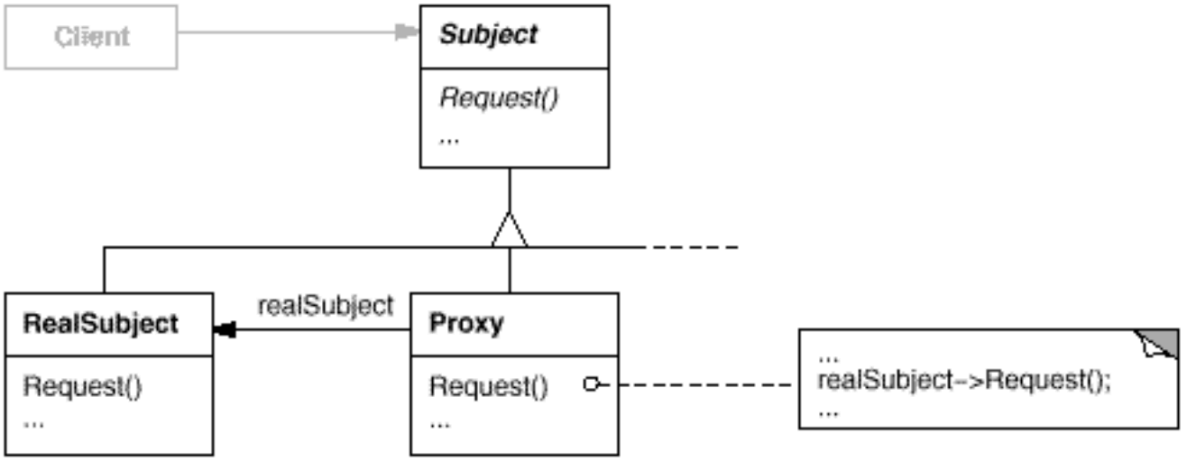
## Applicability

Usar Proxy cuando/para:

- Se necesita una referencia a un objeto más flexible o sofisticada

## Structure






## Participants

- Proxy (ImageProxy):
    - Mantiene una referencia que permite al proxy acceder al objeto real. El proxy puede referirse a un Subject en caso de que las interfaces de RealSubject y Subject sean la misma
    - Proporciona una interfaz idéntica a la de Sujeto, de manera que un proxy pueda ser sustituido por el sujeto real
    - Controla el acceso al sujeto real, y puede ser responsable de su creación y borrado
    - Otras responsabilidades dependen del tipo proxy:
        - Los proxies remotos son responsables de codificar una petición y sus argumentos para enviar la petición codificada al sujeto real que se encuentra en un espacio de direcciónes diferente
        - Los proxies virtuales pueden guardar información adicional sobre el sujeto real, por lo que pueden retrasar el acceso al mismo. Por ejemplo, el ImageProxy guarda la extensión de la imagen real
- Subject (Graphic):
    - Define la interfaz común para el RealSubject y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que aparece un RealSubject
- RealSubject (Image):
    - Define el objeto real representado

# Applications

- Virtual proxy: demora la construcción de un objeto hasta que sea realmente necesario, cuando sea poco eficiente acceder al objeto real
- Protection proxy: restringe el acceso a un objeto por seguridad
- Remote proxy:
    - Representa 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

# Factory Method

## Intent

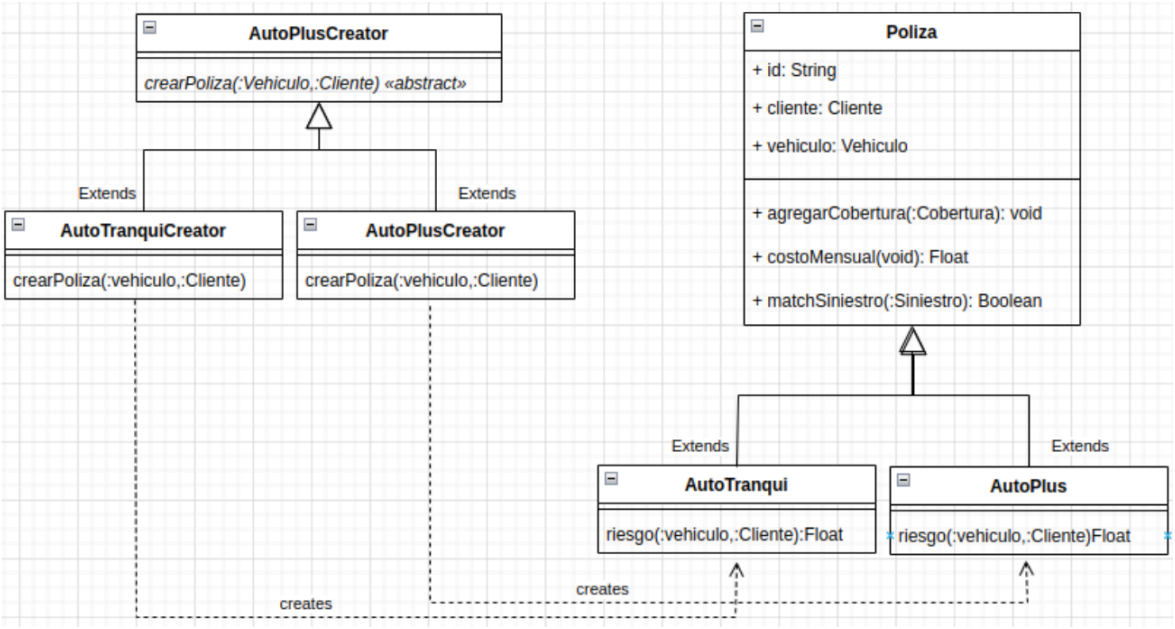
Define una “interface” para la creación de objetos, mientras permite que subclases decidan qué clase se debe instanciar

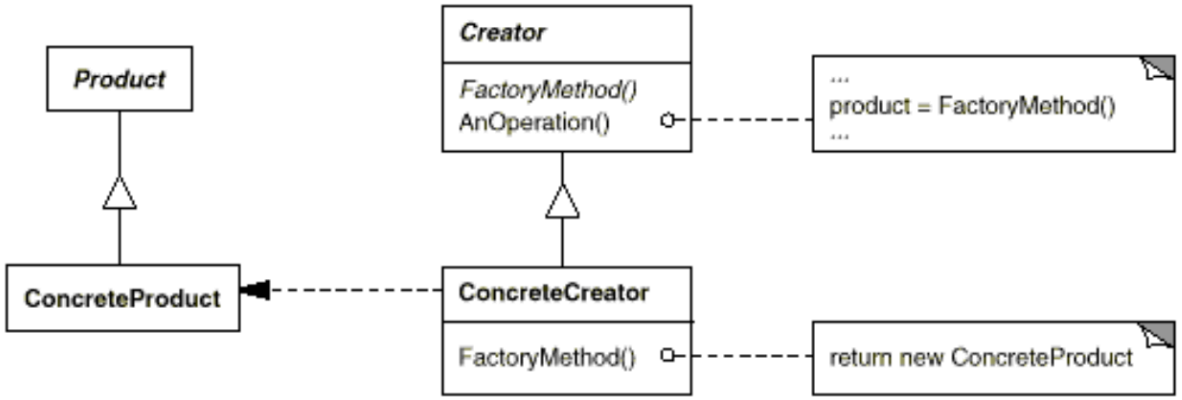
## Applicability

Usar Factory Method cuando:

- Una clase no puede prever la clase de objetos que debe crear
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que esta crea
- Las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar qué subclase auxiliar concreta es en la que se delega

## Structure






## Participants

- Product (Poliza):
  - Define la interface of objetos creados por el “factory method”
- Concrete Product (AutoPiola/AutoPlus):
  - Implementa la interface definida por el Product
- Creator (PolizaCreator (en el UML está erroneamente como AutoPlusCreator)):
  - Declara el “factory method” (abstracto o con comportamiento default)
- ConcreteCreator (AutoPiolaCreator/AutoPlusCreator):
  - Implementa el “factory method”

## Consequences

- Positivas:
  - Abstrae la construcción de un objeto:
    - No acoplamiento entre el objeto que necesita un objeto y el objeto creado
  - Facilita agregar al sistema nuevos tipos de productos:
    - Cada Producto “solo” requiere un ConcreteCreator (*)
- Negativas:
  - Agrega complejidad en el código:
    - 1 llamada a un constructor vs diseñar e implementar varios roles
  - Cada Producto “solo” requiere un ConcreteCreator (!)

# Null Object

## Intent

Proporciona un sustituto para otro objeto que comparte la misma interfaz pero no hace nada

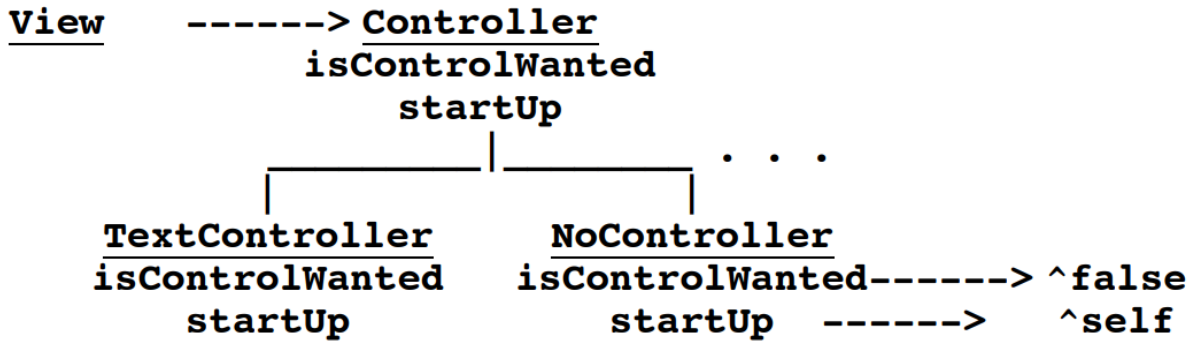
El Null Object encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores

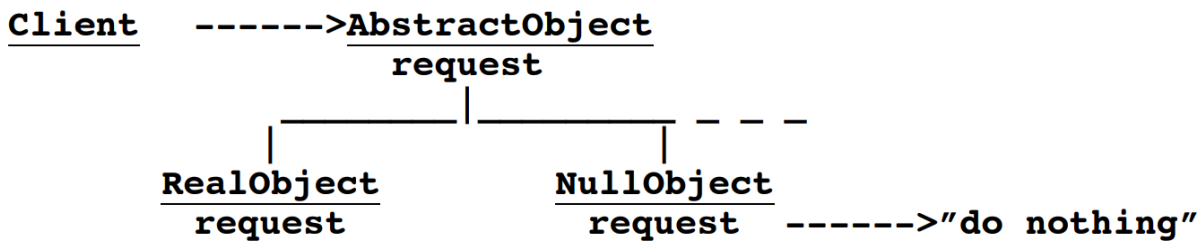
## Applicability

Usar Null Object cuando:

- An object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists
- Some collaborator instances should do nothing
- You want clients to be able to ignore the difference between a collaborator which provides real behavior and that which does nothing. This way, the client does not have to explicitly check for nil or some other special value
- You want to be able to reuse the do nothing behavior so that various clients which need this behavior will consistently work the same way
- All of the behavior which might need to be do nothing behavior is encapsulated within the collaborator class. If some of the behavior in that class is do nothing behavior, most or all of the class' behavior will be do nothing

## Structure






## Participants

- Client (View):
  - Requires a collaborator
- AbstractObject (Controller):
  - Declares the interface for Client's collaborator
  - Implements default behavior for the interface common to all classes, as appropriate
- RealObject (TextController):
  - Defines a concrete subclass of AbstractObject whose instances provide useful behavior that Client expects
- NullObject (NoController):
  - Provides an interface identical to AbstractObject's so that a null object can be substituted for a real object
  - Implements its interface to do nothing. What exactly it means to do nothing is subjective and depends on what sort of behavior Client is expecting. Some requests may be fulfilled by doing something which gives a null result
  - When there is more than one way to do nothing, more than one NullObject class may be required

## Consequences

- Positivas:
  - Elimina todos los condicionales que verifican si la referencia a un objeto es NULL
  - Hace explícito elementos del dominio que hacen "nada"
- Negativas:
  -

# Builder

## Intent

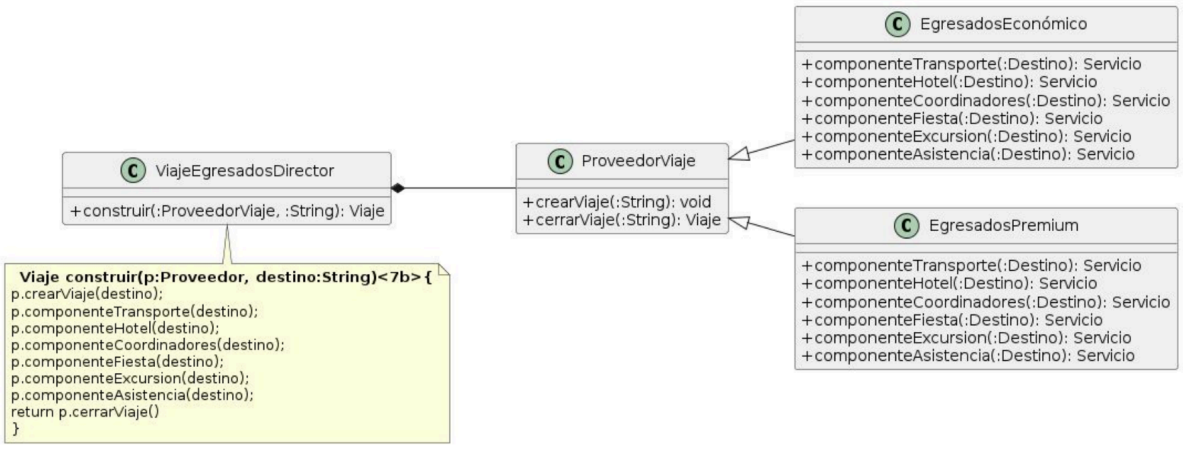
Separa la construcción de un objeto complejo de su representación (implementación) de tal manera que el mismo proceso puede construir diferente representaciones (implementaciones)

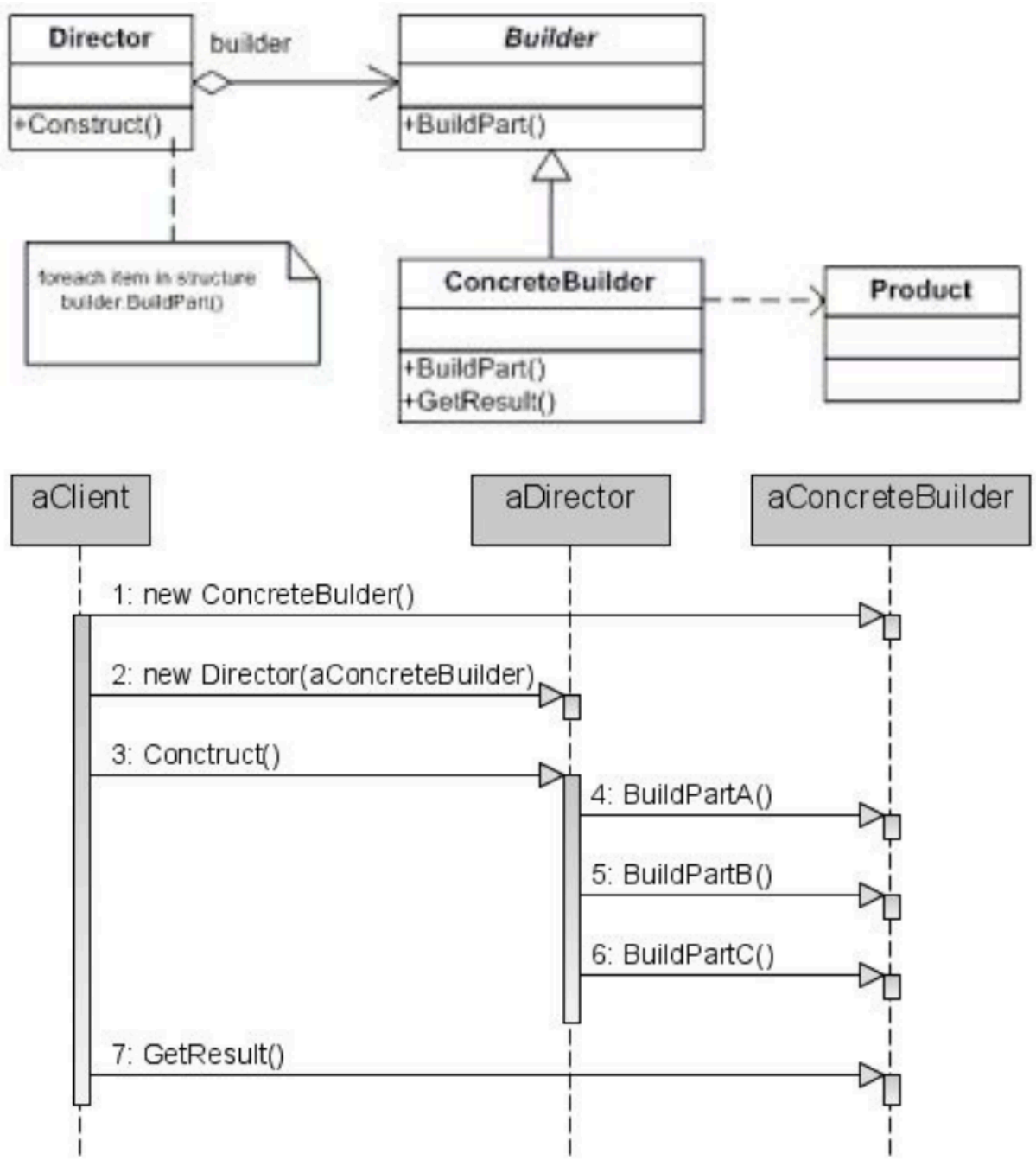
## Applicability

Usar Builder cuando:

- El algoritmo para crear un objeto complejo debería ser independiente de las partes que componen a dicho objeto y de cómo se ensamblan

## Structure






## Participants

- Builder (ProveedorViaje):
    - Especifica una interface abstracta para crear partes de un Producto
- Concrete Builder (EgresadosEconomico/EgresadosPremium):
    - Construye y ensambla partes del producto
    - Guarda referencia al producto en construcción
- Director (ViajeEgresadosDirector):
    - Conoce los pasos para construir el objeto

    - Utiliza el Builder para construir las partes que va ensamblando
    - En lugar de pasos fijos puede seguir una “especificación” (ver knwon uses@ GOF)
- Product (Viaje):
    - Es el objeto complejo a ser construido

## Consequences

- Positivas:
    - Abstrae la construcción compleja de un objeto complejo
    - Permite variar lo que se construye Director <-> Builder
    - Da control sobre los pasos de construcción
- Negativas:
    - Requiere diseñar e implementar varios roles
    - Cada tipo de producto requiere un ConcreteBuilder
    - Builder suelen cambiar o son parsers de specs (> complejidad)

## Implementation

Cuestiones a tener en cuenta:

- El Director solo sabe hacer una cosa
- Los Builders pueden saber hacer cosas que no requiera un Director pero si otro:
    - Ej: componentePsicologico()
- Otros Directors pueden usar los mismos Builders
- Nuevas definiciones de Viajes de Egresado ⇒nuevos directores
- Nuevos servicios ⇒ nuevos Builders

# Type Object

## Intent

"Desacoplar las instancias de sus clases para que esas clases se puedan implementar como instancias de un clase"

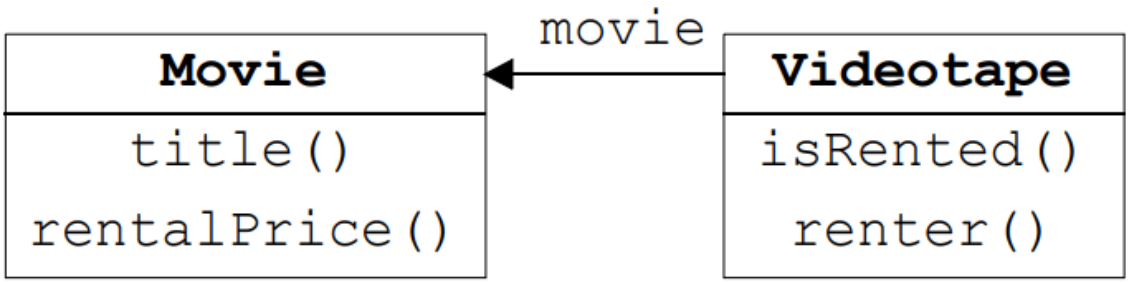
Traducción: identificar elementos del dominio que actúan como clases/metadata de otros elementos del dominio

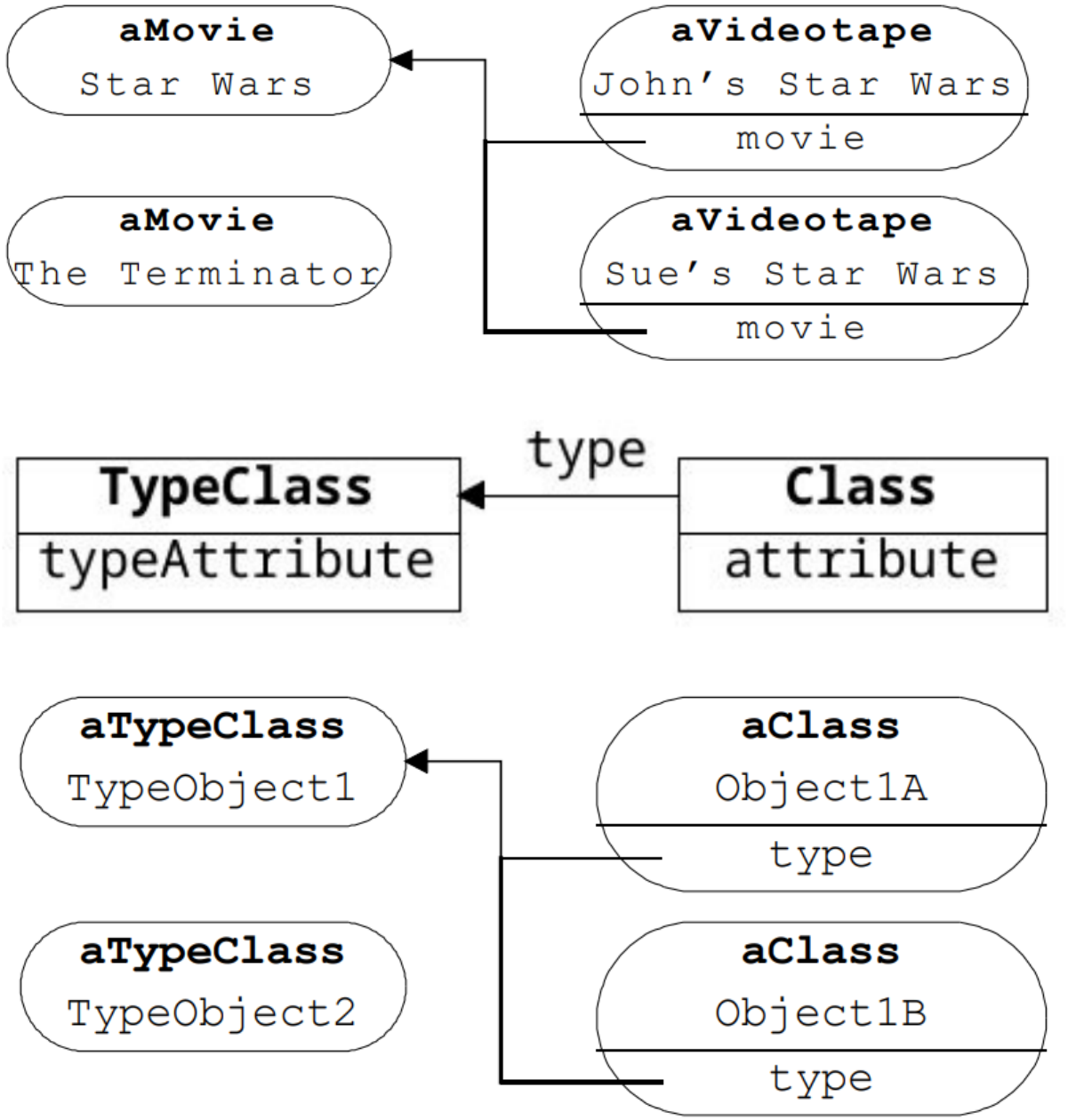
## Applicability

Usar Type Object cuando/para:

- Instances of a class need to be grouped together according to their common attributes and/or behavior
- The class needs a subclass for each group to implement that group's common attributes/behavior
- The class requires a large number of subclasses and/or the total variety of subclasses that may be required is unknown
- You want to be able to create new groupings at runtime that were not predicted during design
- You want to be able to change an object's subclass after it's been instantiated without having to mutate it to a new class
- You want to be able to nest groupings recursively so that a group is itself an item in another group

## Structure






## Participants

- TypeClass (Movie):
  - Is the class of TypeObject
  - Has a separate instance for each type of Object
- TypeObject (Star Wars, The Terminator, Independence Day)
  - Is an instance of TypeClass
  - Represents a type of Object. Establishes all properties of an Object that are the same for all Objects of the same type
- Class (Videotape):

    - Is the class of Object
    - Represents instances of TypeClass
- Object (John's Star Wars, Sue's Star Wars)
    - Is an instance of Class
    - Represents a unique item that has a unique context. Establishes all properties of that item that can differ between items of the same type
    - Has an associated TypeObject that describes its type. Delegates properties defined by its type to its TypeObject