## Definición Patrón:

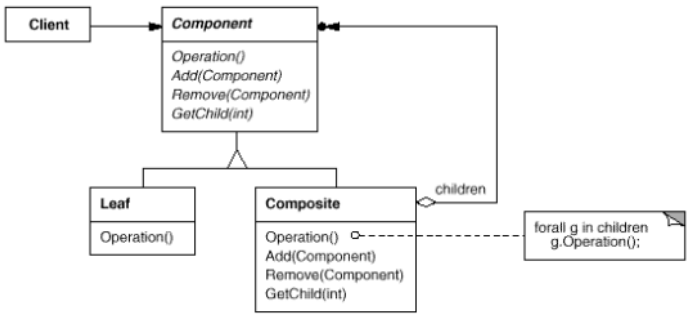
- Un patron es un par problema-solucion
- Los patrones tratan con problemas recurrentes y buenas soluciones a esos problemas
- Definición de GOF:
    - Un patrón de diseño nombra, resume e identifica los aspectos clave de una estructura de diseño común que la hacen útil para crear un diseño orientado a objetos reutilizable.
      El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca en un problema o problema de diseño orientado a objetos en particular. Describe cuándo se aplica, si se puede aplicar en vista de otras restricciones de diseño y las consecuencias y compensaciones de su uso.

## Patrones:

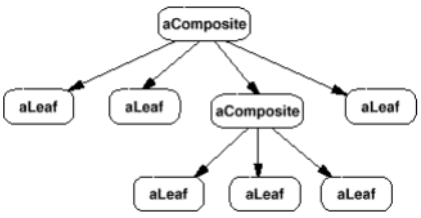
https://sourcemaking.com/design_patterns

### Composite:

- ✓ Intención:
    - Componer objetos en estructuras de arbol para representar jerarquias parte-todo. El Composite permite que los clientes traten a los objetos atomicos y a sus composiciones uniformemente
- ✓ Aplicabilidad:
    - Use el patron Composite cuando
        - Quiere representar jerarquias parte-todo de objetos.
        - Quiere que los objetos “clientes” puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes trataran a los objetos atomicos y compuestos uniformemente.
- ✓ Estructura:



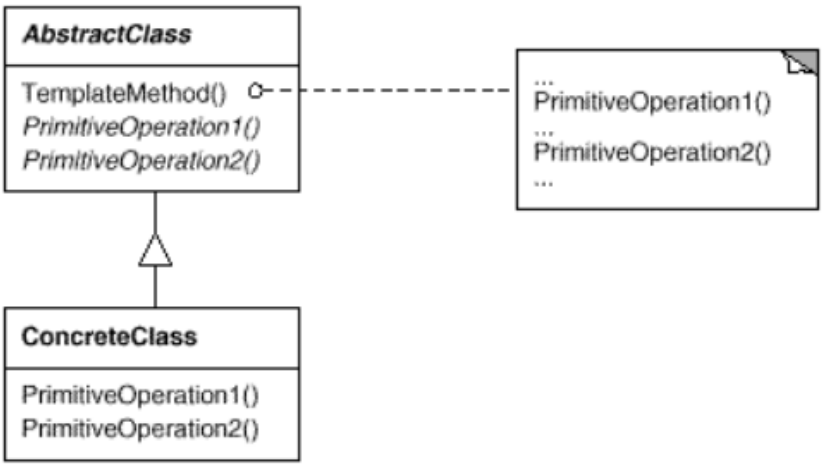

**Una estructura compuesta tipica se vera asi**



- ✓ Participantes:
  - Component
    - Declara la interfaz para los objetos de la composicion.
    - Implementa comportamientos default para la interfaz comun 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
    - Representa arboles “hojas” in la composicion. Las hojas no tienen “hijos”.
    - Define el comportamiento de objetos primitivos en la composicion.
  - Composite
    - Define el comportamiento para componentes con “hijos”.
    - Contiene las referencias a los “hijos”.
    - Implementa operaciones para manejar “hijos”.
- ✓ Consecuencias:
  - 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 y asi recursivamente. En cualquier lugar donde un cliente espera un objeto simple, puede aparecer un compuesto.
  - Simplifica los objetos cliente. Los clientes pueden tratar estructuras compuestas y objetos individuales uniformemente. Los clientes usualmente no saben (y no deberian preocuparse) acerca de si están manejando un compuesto o un simple. Esto simplifica el código del cliente, porque evita tener que escribir código taggeado con estructura de decision sobre las clases que definen la composición
  - Hace mas fácil el agregado de nuevos tipos de componentes porque los clientes no tienen que cambiar cuando aparecen nuevas clases componentes.
  - Puede hacer difícil restringir las estructuras de composición cuando hay algún tipo de conflicto (por ejemplo ciertos compuestos pueden armarse solo con cierto tipo de atómicos)

## Template Method:

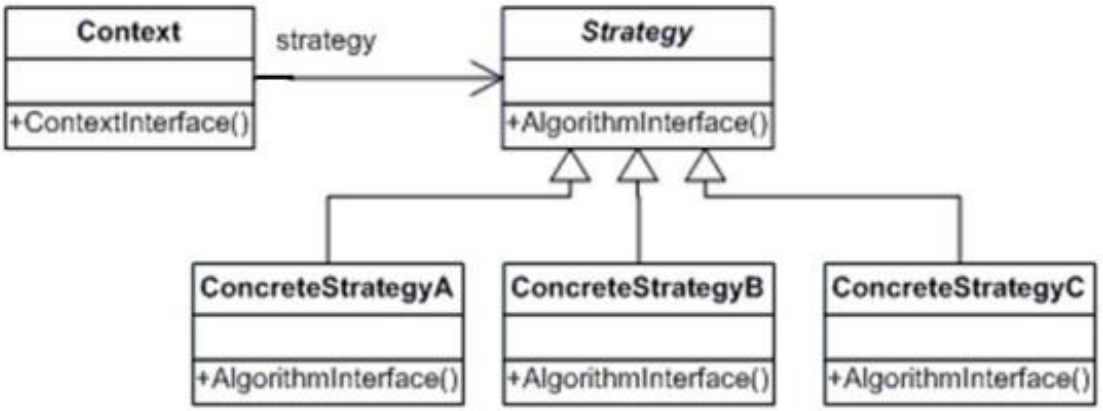
- ✓ Intencion:
  - Definir el esqueleto de un algoritmo en un metodo, difiriendo algunos pasos a las subclases. El template method permite que las subclases redefinan ciertos aspectos de un algoritmo sin cambiar su estructura
- ✓ Aplicabilidad:
  - Para implementar las partes invariantes de un algoritmo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




- ✓ Participantes:
  - Abstract Class:
    - Define operaciones primitivas abstractas que son definidas por las subclases para implementar los pasos de un algoritmo
    - Implementa un método plantilla que define el esqueleto de un algoritmo. El método plantilla llama a las operaciones primitivas así como a operaciones definidas en Abstract Class o a las de otros objetos
  - Concrete Class:
    - Implementa las operaciones primitivas para realizar los pasos del algoritmo específicos de las subclases.

## Strategy:

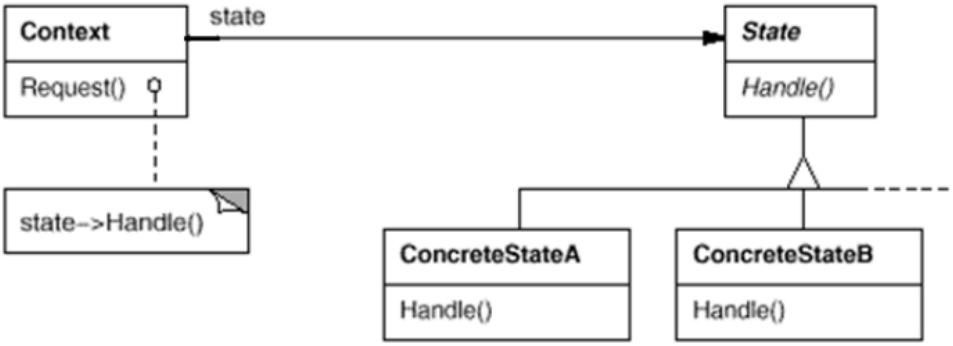
- ✓ Intencion:
  - Definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables. El Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan.
  - Desacoplar un algoritmo del objeto que lo utiliza.
  - Permitir cambiar el algoritmo que un objeto utiliza 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.
- ✓ Estructura:
  - Definir una familia de algoritmos, encapsular cada uno en un objeto y hacerlos intercambiables.
  - Son los clientes del contexto los que generalmente crean las estrategias.




- ✓ Consecuencias:
  - + 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.
  - -Overhead en la comunicación entre contexto y estrategias.
  - - Los clientes deben conocer las diferentes estrategias para poder elegir.
- ✓ Implementación:
  - El contexto debe tener en su protocolo métodos que permitan cambiar la estrategia.
  - Parámetros entre el contexto y la estrategia.

## State:

- ✓ Intencion:
  - Modificar el comportamiento de un objeto cuando su estado interno se modifica.
  - Externamente parecería que la clase del objeto ha cambiado.
- ✓ Aplicabilidad:
  - El comportamiento de un objeto depende del estado en el que se encuentre.
  - Los metodos 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 patron State reemplaza el condicional por clases (es un uso inteligente del polimorfismo)
- ✓ Detalles:
  - 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).
- ✓ Estructura:




DETALLE: El contexto puede pasarse a si mismo cuando llama a Handle() y Handle() cambia el estado del contexto (instancia otro estado).

- ✓ Participantes:
  - Context
    - Define la interfaz que conocen los clientes.
    - Mantiene una instancia de alguna clase de ConcreteState que define el estado corriente
  - State
    - Define la interfaz para encapsular el comportamiento de los estados de Context
  - ConcreteState
    - Cada subclase implementa el comportamiento respecto al estado especifico.
- ✓ Consecuencias:
  - PROS
  - 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.
  - CONTRAS
  - 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

## ¿State o Strategy?

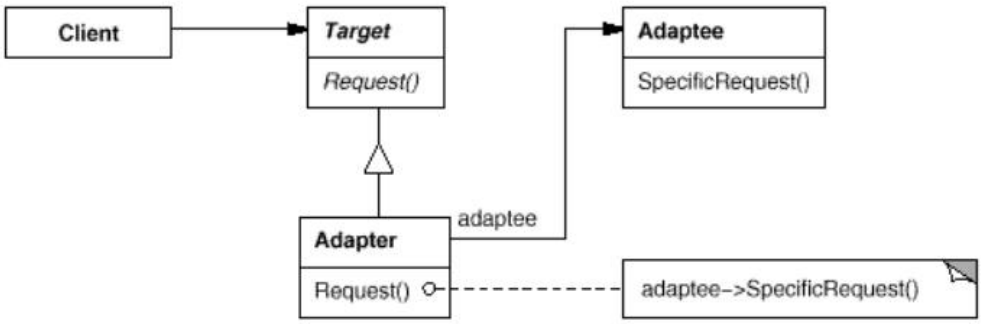
- El patrón State es útil para una clase que debe realizar transiciones entre estados fácilmente.
- El patrón Strategy es útil para permitir que una clase delegue la ejecución de un algoritmo a una instancia de una familia de estrategias
- En Strategy, las diferentes estrategias son conocidas desde afuera del contexto, por las clases clientes del contexto.
- 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.
- Los diagramas se ven muy parecidos, pero el Contexto del Strategy debe contener un mensaje público para cambiar el ConcreteStrategy.
- El **estado** es privado del objeto, ningún otro objeto sabe de él. vs. el **Strategy** suele setearse por el cliente, que debe conocer las posibles estrategias concretas.

- Cada **State** puede definir muchos mensajes. vs. Un **Strategy** suele tener un único mensaje público.
- Los **states** concretos se conocen entre si. vs. Los **strategies** concretos no.

## Wrappers

## Adapter:

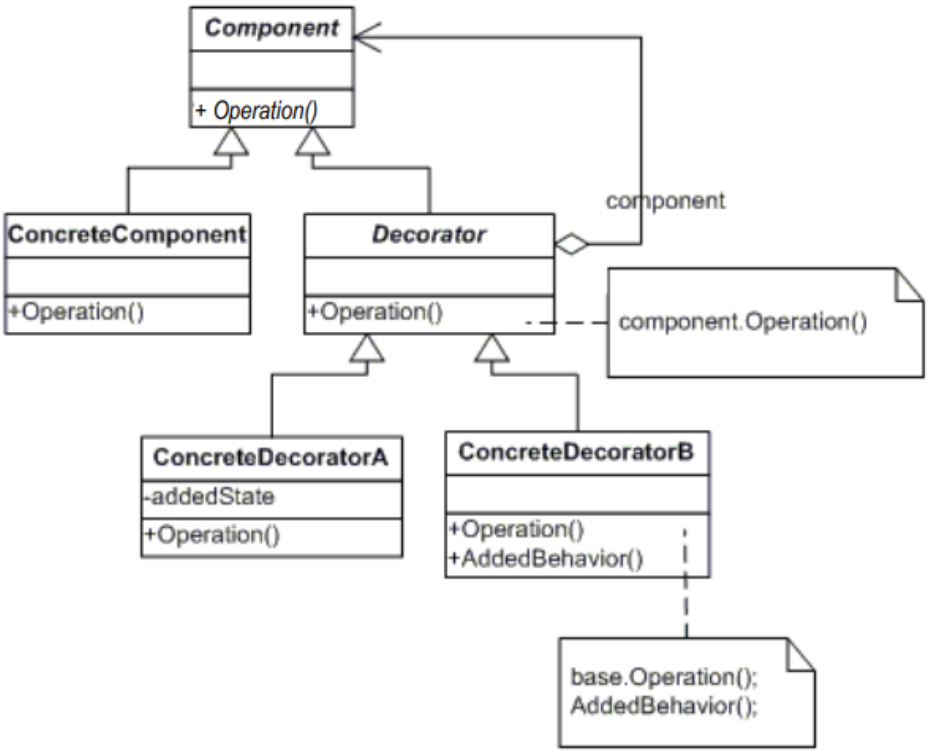
- ✓ Intención:
  - "Convertir" la interfaz de una clase en otra que el cliente espera. El Adapter permite que ciertas clases trabajen en conjunto cuando no podrian por tener interfaces incompatibles
- ✓ Aplicabilidad:
  - Use el adapter cuando:
    - Usted quiere usar una clase existente y su interfaz no es compatible con lo que precisa
- ✓ Estructura:




- ✓ Participantes
  - Target
    - Define la interfaz especìfica que usa el cliente
  - Client
    - Colabora con objetos que satisfacen la interfaz de Target
  - Adaptee
    - Define una interfaz que precisa ser adaptada
  - Adapter
    - Adapta la interfaz del Adaptee a la interfaz de Target

## Decorator:

- ✓ Intencion:
  - Agregar comportamiento a un objeto dinámicamente y en forma transparente.
- ✓ Aplicabilidad:
  - Agregar responsabilidades a objetos individualmente y en forma transparente (sin afectar otros objetos). El orden de las responsabilidades puede variar.
  - Quitar responsabilidades dinámicamente.
  - Cuando subclasificar es impráctico (es estático)
- ✓ Estructura: Definir un decorador (o "wrapper") que agregue el comportamiento cuando sea necesario

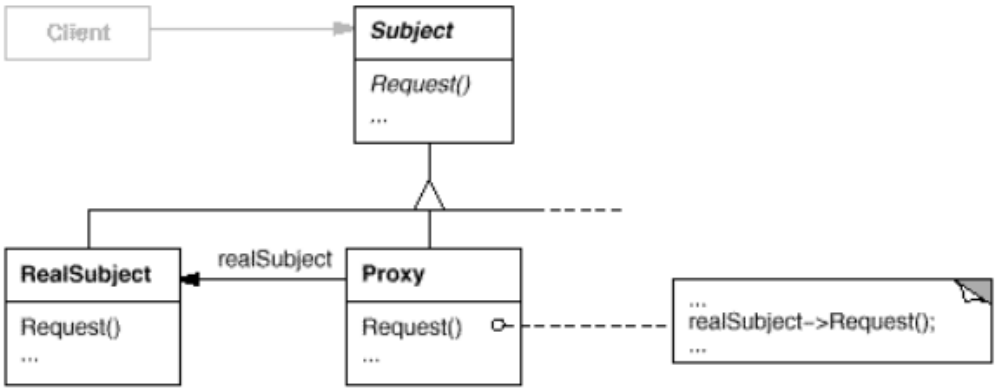



- ✓ Participantes:
  - Component:
    - Define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.
  - Concrete component:
    - Define un objeto al que se pueden añadir responsabilidades adicionales.
  - Decorator:
    - Mantiene una referencia a un objeto Component y define una interfaz que se ajusta a la interfaz del Component
  - Concrete Decorator:
    - Añade responsabilidades al componente.
- ✓ Consecuencias:
  - +Permite mayor flexibilidad que la herencia.
  - + Permite agregar funcionalidad incrementalmente.
  - - Mayor cantidad de objetos, complejo para depurar
- ✓ Implementación:
  - Misma interface entre componente y decorador.
  - No hay necesidad de la clase Decorator abstracta.
  - Cambiar el “skin” vs cambiar sus “guts”

## Proxy:

- ✓ Intencion:
  - Proporcionar un intermediario de un objeto para controlar su acceso.
  - Utilice un nivel adicional de direccionamiento indirecto para admitir el acceso distribuido, controlado o inteligente.
  - Agregue un contenedor y una delegación para proteger el componente real de una complejidad indebida.

- ✓ Aplicabilidad:
  - Cuando se necesita una referencia a un objeto más flexible o sofisticada
  - Necesita admitir objetos que consumen muchos recursos y no desea crear instancias de dichos objetos a menos y hasta que el cliente realmente los solicite.
- ✓ Implementación:
  - Colocar un objeto intermedio que respete el protocolo del objeto que está reemplazando.
  - Algunos mensajes se delegarán en el objeto original. En otros casos puede que el proxy colabore con el objeto original o que reemplace su comportamiento.
- ✓ Estructura:

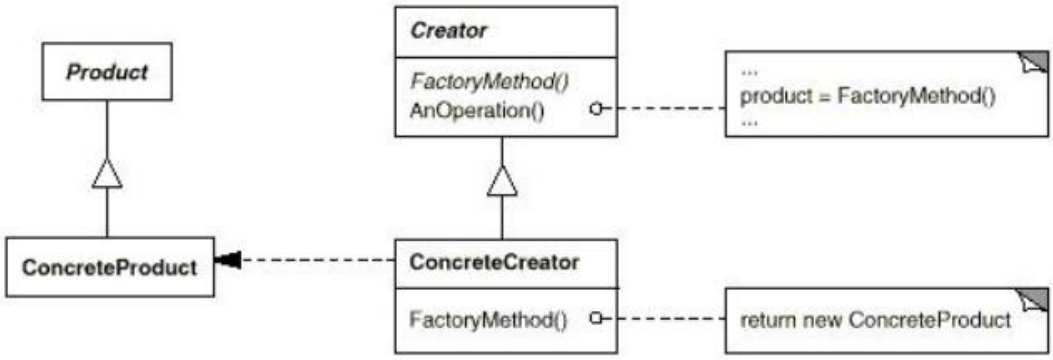



- ✓ Aplicaciones del proxy:
  - Virtual proxy: demorar la construcción de un objeto hasta que sea realmente necesario, cuando sea poco eficiente acceder al objeto real.
  - Protection proxy: Restringir el acceso a un objeto por seguridad.
  - Remote proxy: representar un objeto remoto en el espacio de 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.

## Creacionales

### Factory Method:

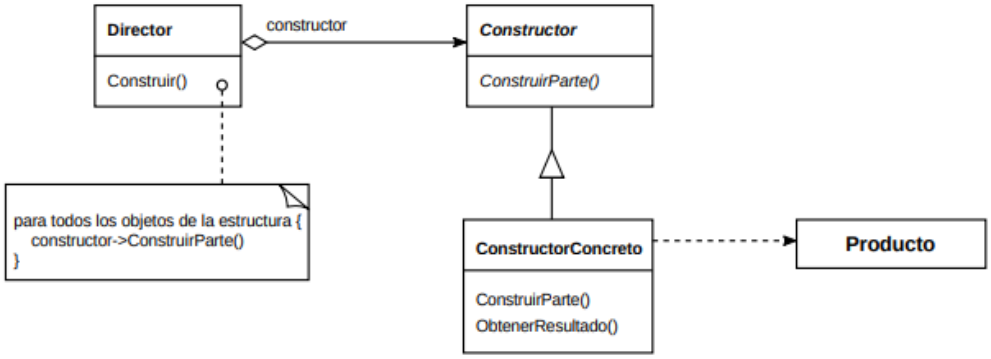
- ✓ Intención:
  - Define una “interface” para la creación de objetos, mientras permite que subclases decidan qué clase se debe instanciar.
- ✓ Aplicabilidad:
  - una clase no puede prever la clase de objetos que debe crear.
  - una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
  - las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar qué subclase de auxiliar concreta es en la que se delega.
- ✓ Estructura




- ✓ Participantes:
  - Product
    - Define la interface of objetos creados por el “factory method”
  - ConcreteProduct
    - Implementa la interface definida por el Product
  - Creator
    - Declara el “factory method” (abstracto o con comportamiento default)
  - ConcreteCreator
    - Implementa el “factory method
- ✓ Consecuencias:
  - 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 (puede ser también una desventaja)
  - Agrega complejidad en el código
    - 1 llamada a un constructor vs diseñar e implementar varios roles

## 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)
- ✓ Aplicabilidad:
  - el algoritmo para crear un objeto complejo debiera ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
  - el proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido
- ✓ Estructura:



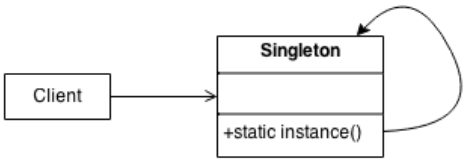

- ✓ Participantes:
  - Builder:
    - Especifica una interfaz abstracta para crear las partes de un objeto Producto.
  - Concrete Builder:
    - Implementa la interfaz Constructor para construir y ensamblar las partes del producto.
    - Define la representación a crear.
    - Proporciona una interfaz para devolver el producto
  - Director:
    - 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”
  - Product:
    - Es el objeto complejo a ser construido
- ✓ Colaboraciones:
  - El cliente crea el objeto Director y lo configura con el objeto Constructor deseado.
  - El Director notifica al constructor cada vez que hay que construir una parte de un producto.
  - El Constructor maneja las peticiones del director y las añade al producto. El cliente obtiene el producto del constructor
- ✓ Consecuencias:
  - Abstrae la construcción compleja de un objeto complejo ● Permite variar lo que se construye Director <-> Builder
  - Da control sobre los pasos de Intenciónón
  - Requiere diseñar e implementar varios roles
  - Cada tipo de producto requiere un ConcreteBuilder
  - Builder suelen cambiar o son parsers de specs (> complejidad)

## Singleton:

- ✓ Intención:
  - Asegurar que una clase solo tiene una instancia y provee una manera uniforme de acceder a ella.
- ✓ Aplicabilidad:

- Debe haber exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
- La única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.

✓ Estructura:






✓ Participantes:
- Singleton
  - Define una operación Instancia que permite que los clientes accedan a su única instancia. Instancia es una operación de clase (es decir, un método de clase en Smalltalk y una función miembro estática en C++).
  - puede ser responsable de crear su única instancia.

✓ Implementación (Java):
- Constructor privado para restringir la creación de instancias de la clase de otras clases.
- Variable estática privada de la misma clase que es la única instancia de la clase.
  - Eager Initialization
    - Static block con try/catch
  - Lazy Initialization
- Método estático público que devuelve la instancia de la clase, este es el punto de acceso global para que el mundo exterior obtenga la instancia de la clase singleton.
  - Lugar para implementar Lazy Initialization

```java
public class Surface {
    private static Surface instance;
    private static Frame[] frames = new Frame[2];
    private int current_idx;
    private Surface(){
        //frames[1] = new Frame("4k");
        //frames[2] = new Frame("4k");
        //current_idx = 1;
    }
    static {
        try{instance = new Surface();
        }catch (Exception e){
            throw new RuntimeException("Unable to create Surface");
        }

    }
    public static Surface getInstance(){return instance;}
    public void bitblit(Bitmap bitmap){
        //blit bitmap onto the current frame
        //...
    }
}
```

Eager Initialization + Static block

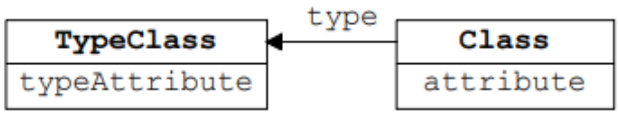
```java
public static synchronize Surface getInstance(){
    if(instance == null){
        instance = new Surface();
    }
    return instance;}
```

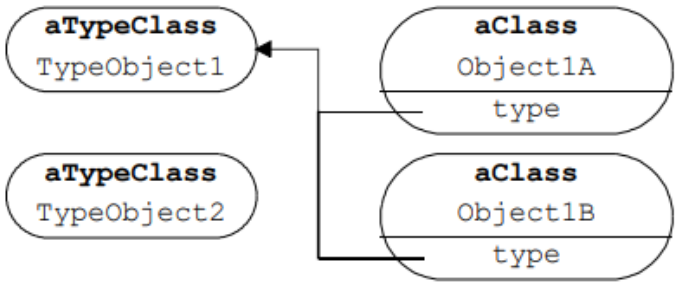
Lazy Initialization

- ✓ Consecuencias:
  - Permite tener una única instancia de Surface
  - Cualquier método de cualquier objeto se puede acceder al Singleton
  - Es dificil de implementar el ciclo de vida del Singleton
    - ¿Como/cuando se resetea?
  - Genera acoplamiento con la clase y su instancia

## Type Object

- ✓ Intención:
- ✓ Aplicabilidad:
  - Las instancias de una clase deben agruparse de acuerdo con sus atributos y/o comportamientos comunes.
  - La clase necesita una subclase para que cada grupo implemente los atributos/comportamientos comunes de ese grupo.
  - La clase requiere un gran número de subclases y/o se desconoce la variedad total de subclases que pueden ser necesarias.
  - Desea poder crear nuevas agrupaciones en tiempo de ejecución que no se predijeron durante el diseño.
  - Desea poder cambiar la subclase de un objeto después de que se haya creado una instancia sin tener que mutarlo a una nueva clase.
  - Desea poder anidar agrupaciones recursivamente para que un grupo sea en sí mismo un elemento en otro grupo.
- ✓ Estructura:






- ✓ Participantes:
  - TypeClass
    - es la clase de TypeObject.
    - tiene una instancia separada para cada tipo de Objeto.
  - TypeObject
    - es una instancia de TypeClass.
    - representa un tipo de Objeto. Establece todas las propiedades de un Objeto que son iguales para todos los Objetos del mismo tipo.
  - Class
    - es la clase de Object.
    - representa instancias de TypeClass.
  - Object
    - es una instancia de Class.
    - representa un elemento único que tiene un contexto único. Establece todas las propiedades de ese elemento que pueden diferir entre elementos del mismo tipo.
    - tiene un TypeObject asociado que describe su tipo. Delega propiedades definidas por su tipo a su TypeObject.

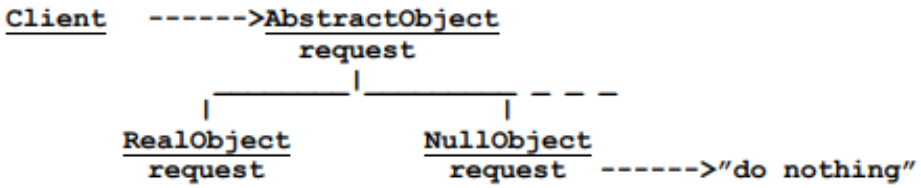
TypeClass y Class son clases. TypeObject y Object son instancias de sus respectivas clases. Como con cualquier instancia, un TypeObject u Object sabe cuál es su clase. Además, un objeto tiene un puntero a su TypeObject para que sepa cuál es su TypeObject. El Object usa su TypeObject para definir su comportamiento de tipo. Cuando el Object recibe solicitudes que son específicas del tipo pero no de la instancia, delega esas solicitudes a su TypeObject. Un TypeObject también puede tener punteros a todos sus Objects.

## Null Object

- ✓ Intención:
  - Proporcione un sustituto para otro objeto que comparte la misma interfaz pero no hace nada. El Objeto Nulo encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores.
- ✓ Aplicabilidad:
  - Algunas instancias colaboradoras no deberían hacer nada.
  - Quiere que los clientes puedan ignorar la diferencia entre un colaborador que proporciona un comportamiento real y el que no hace nada. De esta manera, el cliente no tiene que verificar explícitamente nil o algún otro valor especial.
  - Desea poder reutilizar el comportamiento de no hacer nada para que varios clientes que necesiten este comportamiento funcionen de la misma manera consistentemente.

- Todo el comportamiento que podría necesitar ser un comportamiento de no hacer nada se encapsula dentro de la clase de colaborador. Si parte del comportamiento en esa clase es el comportamiento de no hacer nada, la mayor parte o todo el comportamiento de la clase será el de no hacer nada.

✓ Estructura:




✓ Participantes:
- Client
  - Requiere un colaborador
- AbstractObject
  - Declara la interfaz para el colaborador del Cliente.
  - Implementa el comportamiento predeterminado para la interfaz común a todas las clases, según corresponda.
- RealObject
  - Define una subclase concreta de AbstractObject cuyas instancias proporcionan un comportamiento útil que el Cliente espera.
- NullObject
  - Proporciona una interfaz idéntica a la de AbstractObject para que un objeto nulo pueda sustituirse por un objeto real.
  - Implementa su interfaz para no hacer nada. Lo que significa exactamente no hacer nada es subjetivo y depende del tipo de comportamiento que espera el Cliente. Algunas solicitudes pueden cumplirse haciendo algo que da un resultado nulo.
  - Cuando hay más de una forma de no hacer nada, es posible que se requiera más de una clase NullObject.

✓ Consecuencias:
- Elimina todos los condicionales que verifican si la referencia a un objeto es NULL.
- Hace explícitos elementos del dominio que hace “nada”