Patrones Software
Tema 2:
Patrones de diseño fundamentales

Patrones de Diseño Fundamentales

- Son los patrones de diseño más importantes y fundamentales según Mark Grand.
- Estos patrones se utilizan frecuentemente en la definición de otros patrones.
- Esta categoría comprende los siguientes patrones básicos:
  - ✓ Delegation
  - ✓ Interface
  - ✓ Abstract Superclass
  - ✓ Interface and Abstract Class
  - ✓ Immutable
  - ✓ Marker Interface
  - ✓ Proxy

Patrones de Diseño Fundamentales

- *Delegation, Interface, Abstract Superclass* e *Interface and Abstract Class* son patrones que muestran la forma de organizar las relaciones entre las clases. La mayoría de los patrones hacen uso de al menos uno de estos patrones.
- El patrón *Immutable* describe una manera de evitar errores cuando varios objetos tienen acceso a un mismo objeto de forma concurrente.
- El patrón *Marker Interface* describe como diseñar clases que hacen uso de objetos que implementan una misma interface, haciendo posible su clasificación.
- El patrón *Proxy* es la base de una serie de patrones en los que se gestiona el acceso de un objeto a otro objeto de una forma transparente.

Delegation

Objetivo

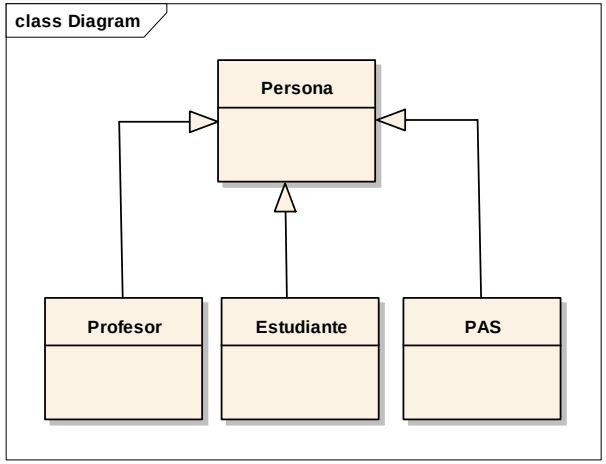
- Es un patrón fundamental de tipo estructural.
- Indica cuándo no usar herencia. La delegación es una forma de extender y reutilizar la funcionalidad de una clase, escribiendo una clase adicional con funcionalidad extra que usa instancias de la clase original para proveer su propia funcionalidad.
- La delegación es una forma de extender el comportamiento de una clase mediante llamadas a métodos de otra clase, más que heredando de ella.

Objetivo

- La delegación es más apropiada que la herencia en muchas situaciones:
  - Por ejemplo, la herencia es útil para modelar relaciones de tipo es-un (relaciones estáticas).
  - Sin embargo, relaciones de tipo es-un-rol-ejecutado-por son mal modeladas con herencia. En este tipo de relaciones, instancias de una clase pueden jugar múltiples roles.

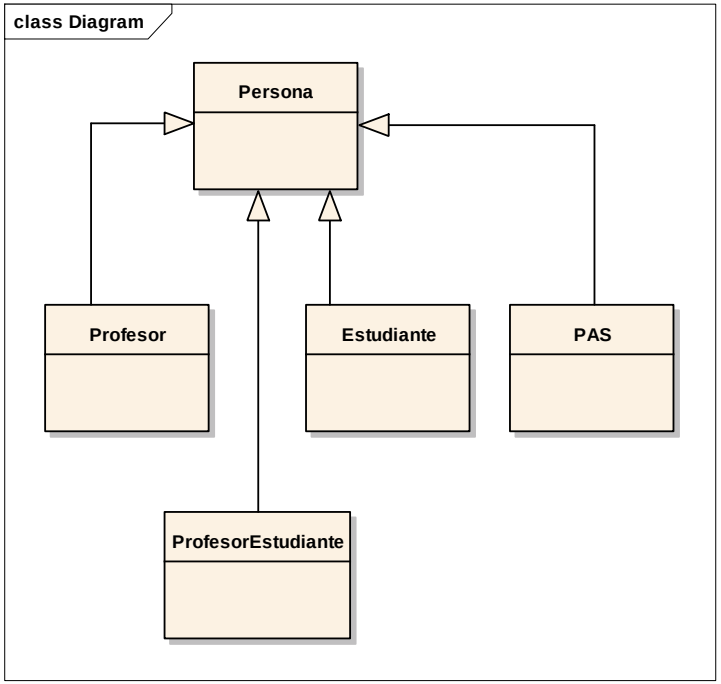
Ejemplo

- **Personas en una Universidad:**



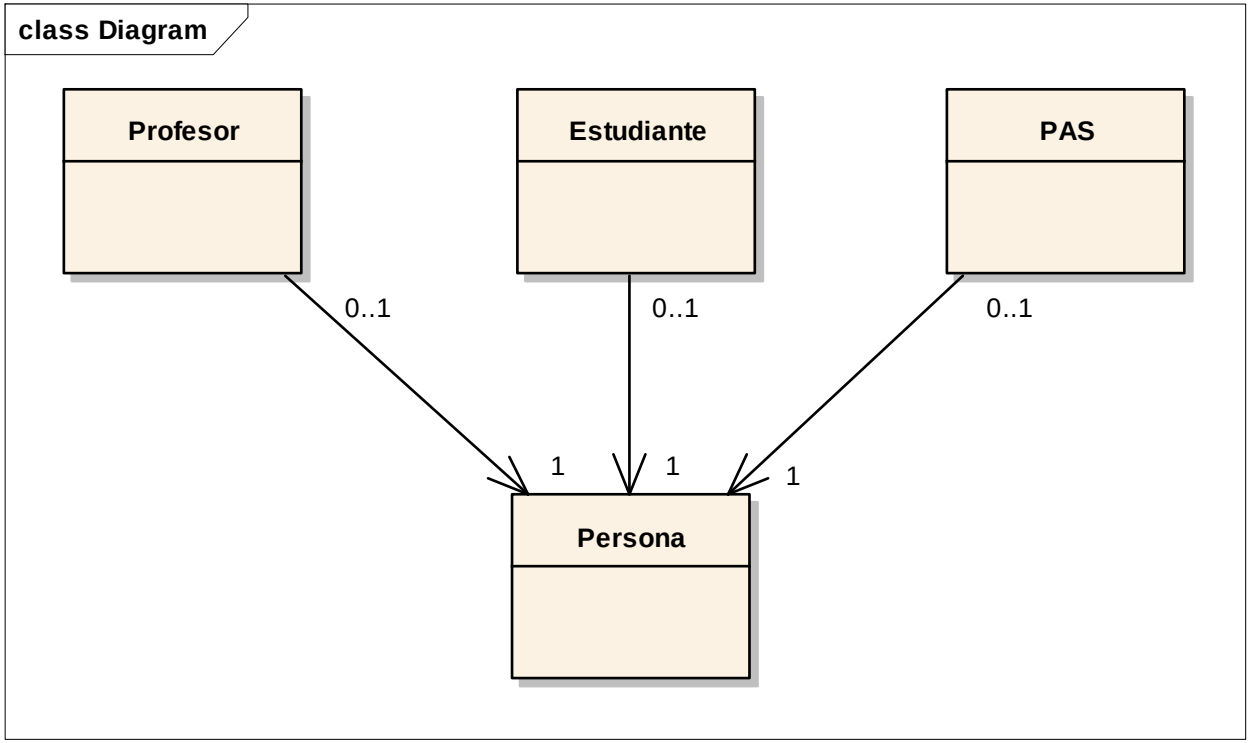
¿Y si tenemos a un profesor estudiante?

¿Y si una misma persona puede jugar más de un rol al mismo tiempo?



Ejemplo

- **Solución: Es posible representar personas en diferentes roles usando delegación.**



Estructura

- La solución general propuesta en este patrón es incorporar la funcionalidad de la clase original usando una instancia de la misma y llamando a sus métodos.

- En el diagrama se muestra una clase con rol *Delegador* que usa una clase con el rol *Delegado*. Aquí se usa la delegación para reutilizar y extender el comportamiento de la clase.

Ejemplo

```
class A {
     void f() { System.out.println("A: ejecutando f()"); }
     void g() { System.out.println("A: ejecutando g()"); }
}

class C {
     // delegation
     A a = new A();
     void f() { a.f(); }
     void g() { a.g(); }
     X x = new X(); // atributo normal
     void y() {
        /* hacer algo */
     }
 }

public class Main {
      public static void main(String[] args) {
           C c = new C();
           c.f();
           c.g();
      }
}
```

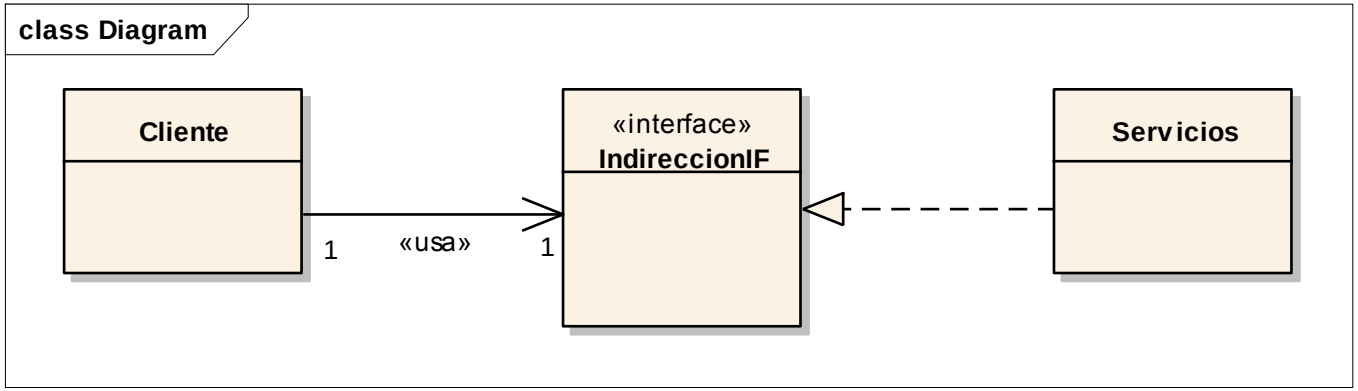

Interface

Objetivo

- Es un patrón fundamental de tipo estructural.
- Mantiene una clase (la interfaz) que usa datos y servicios provistos por otras clases independientes, para proveer un acceso uniforme.
- Usando indirección, esta clase interfaz provee a sus clases herederas acceso uniforme a métodos específicos, sin que deban saber a que clase específica pertenecen.
- Por lo general, los patrones Interfaz y Delegación son usados juntos.

Estructura

- El diagrama general de este patrón es el siguiente:

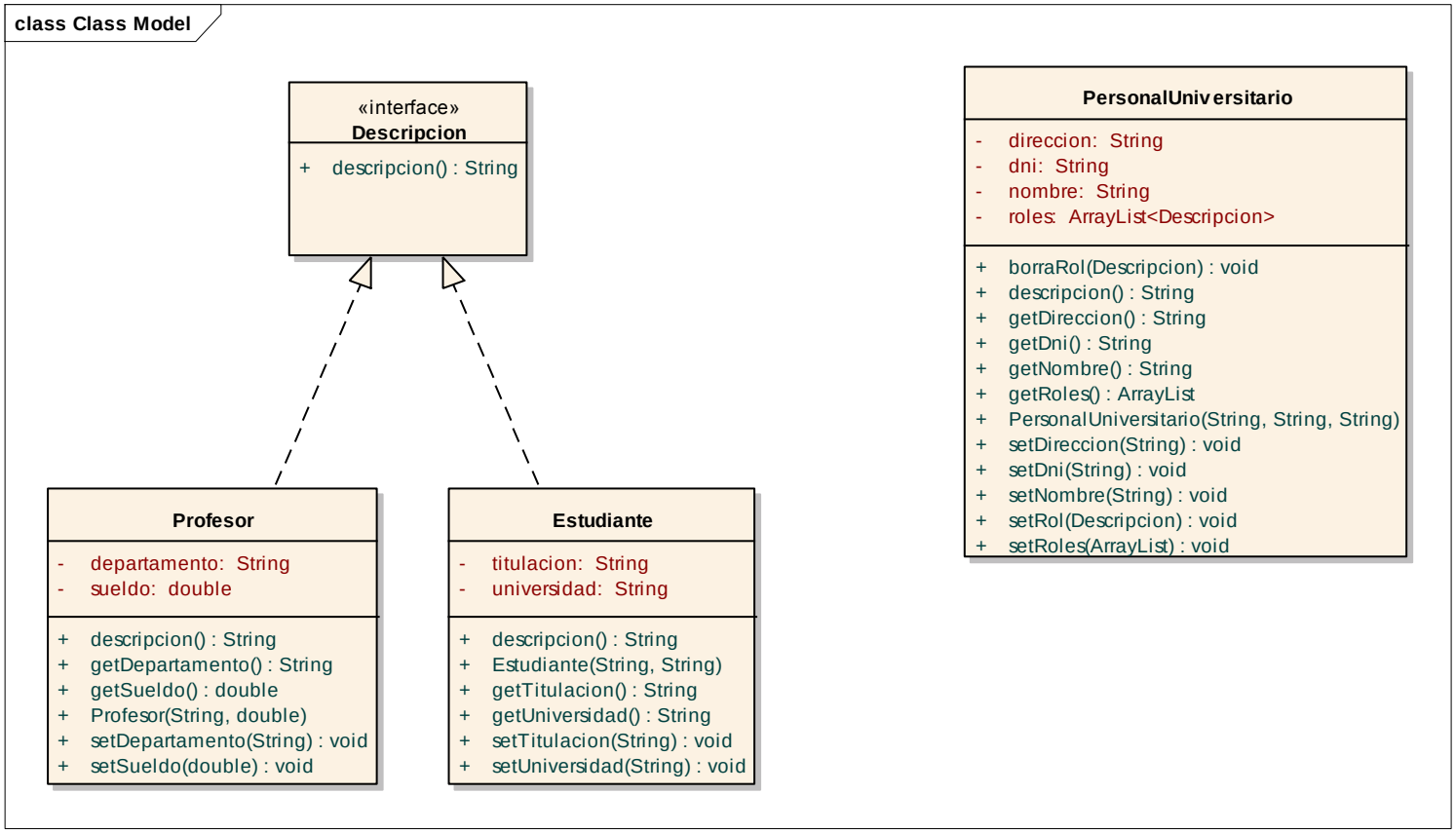


- En la figura la clase *Cliente* usa otras clases que implementan la interfaz *IndirecciónIF*. La interfaz *IndirecciónIF* provee la indirección que mantiene a la clase *Cliente* independiente de las clases que proveen los servicios (clase *Servicios*).

Código de ejemplo

Delegation e Interface

Código de ejemplo



Código de ejemplo

- Identificamos a continuación los elementos del patrón Interface:
  - Interfaz: *Descripcion*
  - Clases que realizan la interfaz: *Profesor* y *Estudiante*
- Identificamos los elementos del patrón Delegation:
  - Delegador: *PersonalUniversitario*. Esta clase tiene un array de elementos de tipo Descripción. El comportamiento de dichos elementos dependerá de la clase concreta (*Profesor* y *Estudiante*) que realice la interfaz *Descripcion*.
  - Delegado: cada una de las clases concretas que realizan la interfaz Descripción (*Profesor* y *Estudiante*).

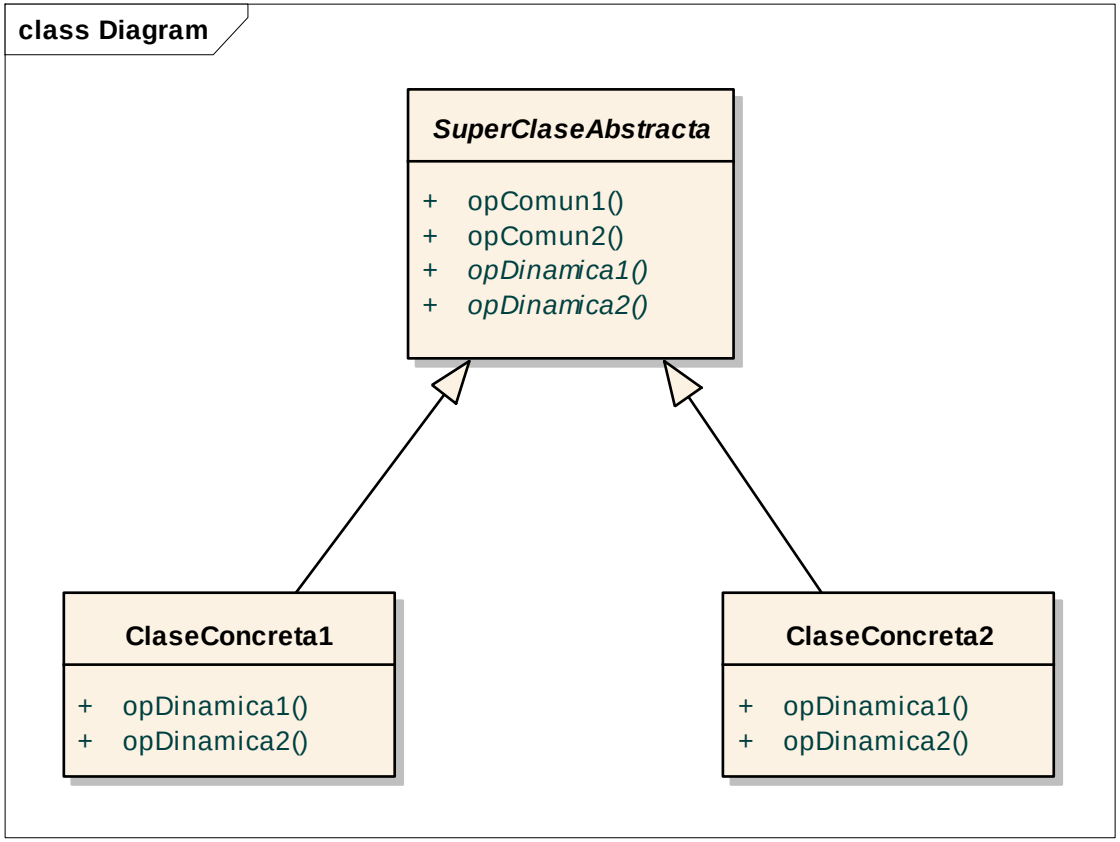
Abstract Superclass

Objetivo

- Garantizar la coherencia de comportamiento de las clases conceptualmente relacionadas dándoles una clase padre abstracta.
- Organizar el comportamiento común de las clases relacionadas en una superclase abstracta.
- En la medida de lo posible, organizar el comportamiento dinámico en métodos abstractos con la misma signatura.

Estructura

- El diagrama general de este patrón es el siguiente:



Estructura

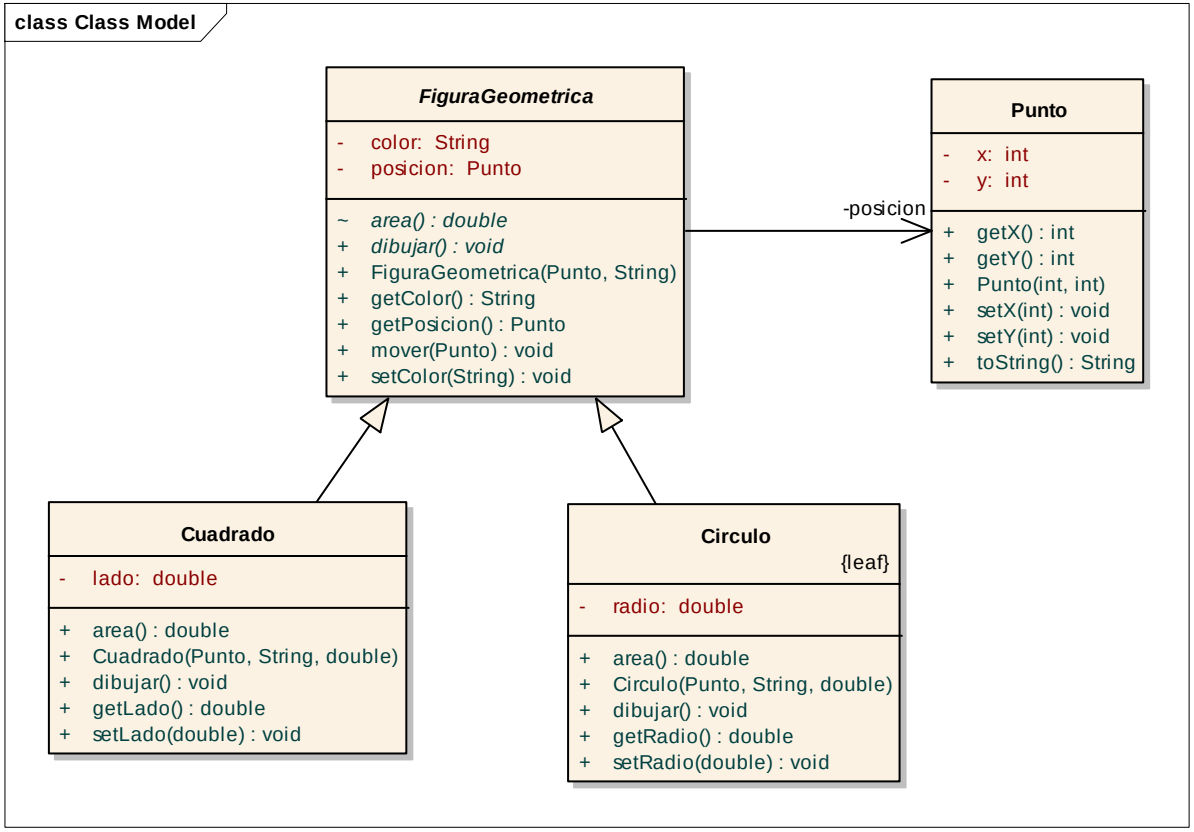
- **SuperClaseAbstracta**. Representa una superclase abstracta que encapsula la lógica común para todas las clases hijas. Las clases relacionadas extienden de esta clase. Tendremos métodos comunes a todas las clases y métodos abstractos con la misma signatura.

- **ClaseConcreta 1 y 2**. Heredan los métodos comunes e implementan los métodos abstractos proporcionando una lógica concreta.

Código de ejemplo

Abstract Super Class

Código de ejemplo



Código de ejemplo

- Identificamos a continuación los elementos del patrón:
  - SuperClaseAbstracta: FiguraGeometrica
  - ClaseConcretaN: en este caso tenemos dos clases concretas que son Cuadrado y Circulo

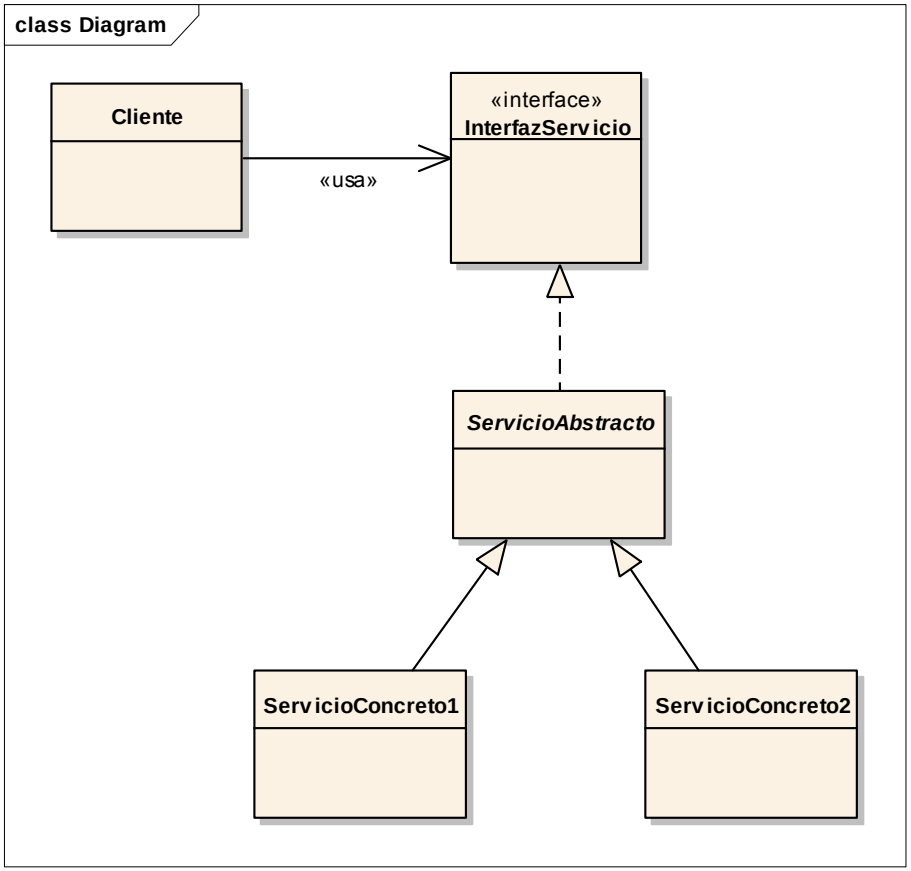
Interface y Abstract Class

Objetivo

- Cuando se necesita mantener las clases clientes independientes de las clases que implementan una funcionalidad o comportamiento.
- De esta forma se garantiza la coherencia entre el comportamiento y la implementación.
- Las clases implementarán una interfaz y heredarán de una clase abstracta.
- Muy utilizado en el diseño de frameworks.

Estructura

- El diagrama general de este patrón es el siguiente:

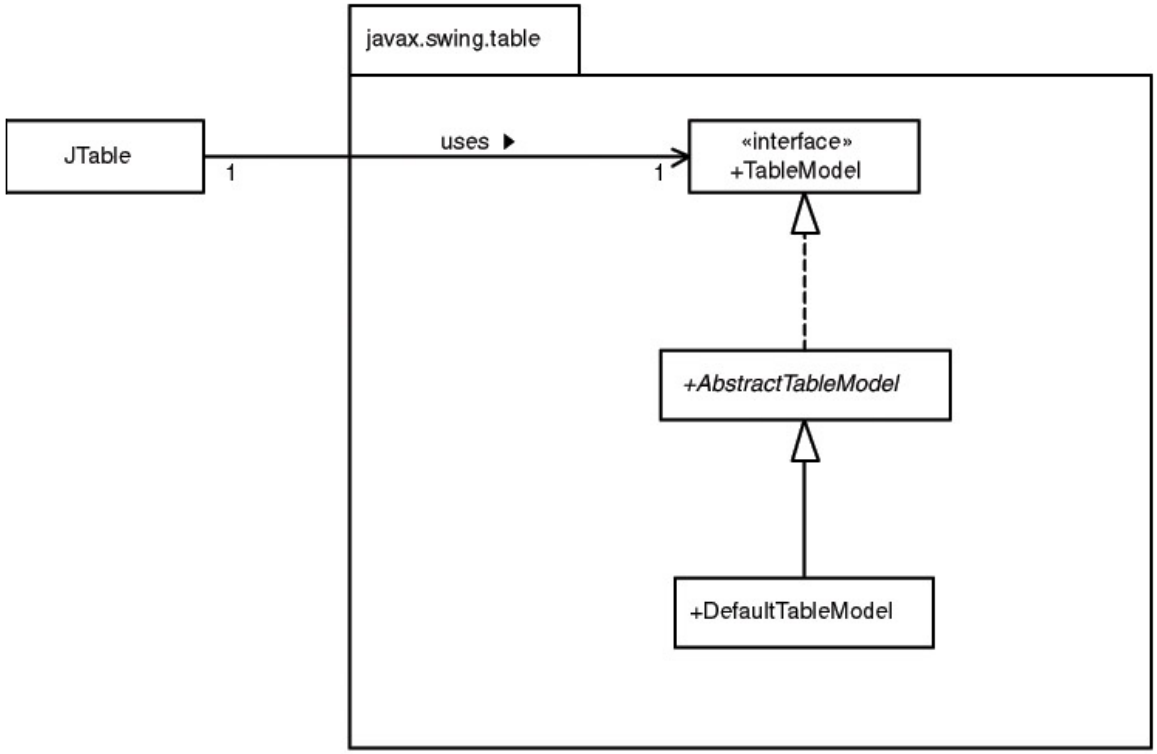


Estructura

- Utilice el patrón si se necesita ocultar a los clientes la clase de un objeto que proporciona un servicio.
- El cliente tiene acceso a los objetos proveedores de servicios de forma indirecta a través de una interfaz. Esta indirección permite a los clientes acceder a la prestación de los servicios sin tener que saber qué tipo de objetos utiliza.
- La clase abstracta proporcionará una funcionalidad similar y contendrá las partes comunes de la aplicación.

Ejemplo

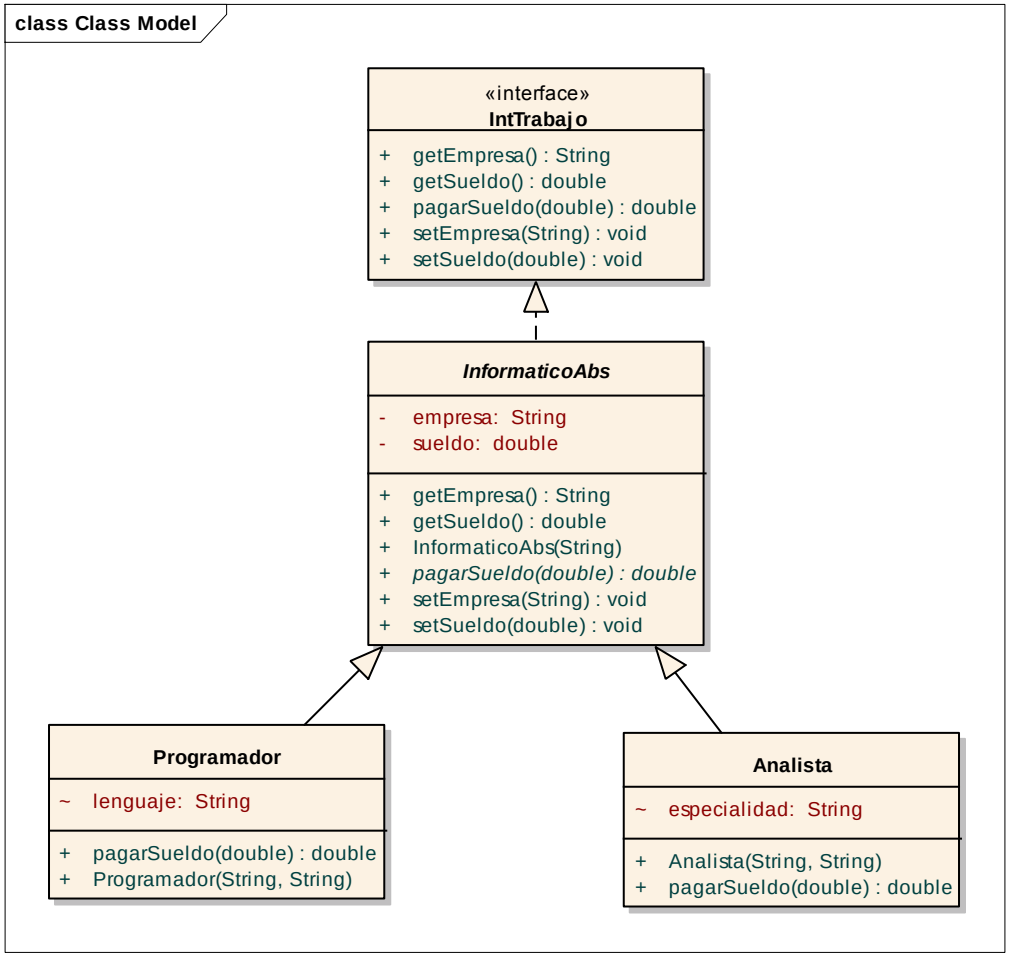
- Ejemplo de uso del patrón en Java:



Código de ejemplo

Interface y Abstract Class

Código de ejemplo



Código de ejemplo

- Identificamos a continuación los elementos del patrón:
  - InterfazServicio: IntTrabajo
  - ServicioAbstracto: InformaticoAbs
  - ServicioConcretoN: Programador y Analista

Inmutable

Objetivo

- Es un patrón fundamental de concurrencia.
- Incrementa la robustez de objetos que comparten referencias al mismo objeto, y reduce el costo (overhead) de los accesos concurrentes a un objeto.
- Evita la necesidad de sincronizar múltiples threads en ejecución que comparten un objeto.
- Se usa en muchos contextos, donde se necesitan instancias de clases que son compartidas por múltiples objetos, y cuyos estados son más frecuentemente "consultados" que "modificados“.

Objetivo

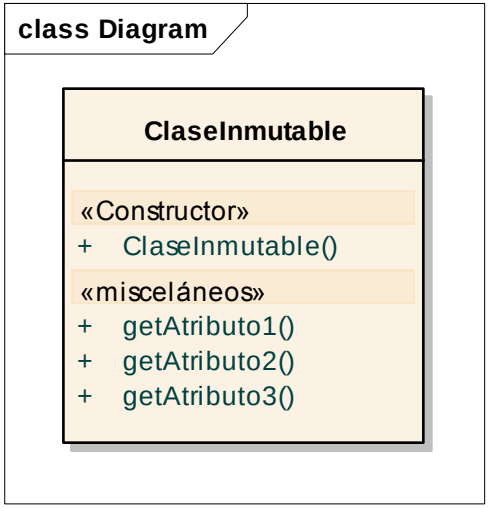
- Cuando múltiples objetos comparten el acceso a un mismo objeto hay que coordinarlos para evitar problemas.
- Si las modificaciones y consultas al objeto compartido son realizadas asincrónicamente, entonces para que el funcionamiento sea el correcto, en el código implementado se debe sincronizar los accesos.
- El patrón Inmutable evita estos problemas, organizando una clase de tal forma que la información del estado de sus instancias nunca cambia después de ser creadas.

Objetivo

- Este patrón recomienda que para evitar la administración de la propagación y sincronización de cambios en el estado de los objetos, se usen múltiples objetos de ese tipo, haciendo estos objetos inmutables, es decir, deshabilitando cualquier cambio en su estado después de construido.
- Aspectos en la implementación de este patrón:
  1. Ningún método (a excepción del constructor) debe modificar los valores de las variables de instancia de la clase.
  2. Cualquier método que calcula un nuevo estado, debe almacenar la información en una nueva instancia de la misma clase, en lugar de modificar el estado de un objeto ya existente.

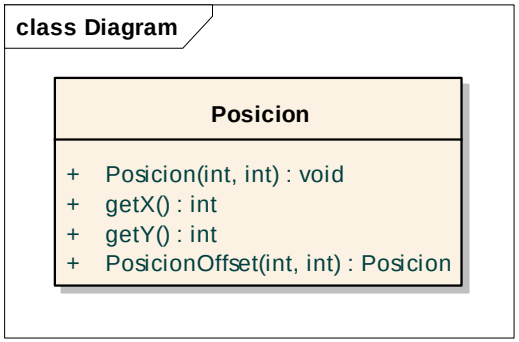
Estructura

- El diagrama general de este patrón es el siguiente:



Ejemplo

- Tenemos una clase que representa posiciones. La clase Posición tiene como atributos x e y, y un constructor que especifica los valores x e y. También tiene métodos para consultar estos valores.
- Finalmente, tiene un método para crear un nuevo objeto Posición dados los valores actuales x e y (por ejemplo, desplazándose horizontal y verticalmente).
- No tiene métodos para modificar los valores x e y.



Ejemplo

```
class Posicion {

        private int x;
        private int y;

        public Posicion(int x, int y) {
                this.x = x;
                this.y = y;
        }

        public int getX() { return x; }
        public int getY() { return y; }

        public Posicion desplazar(int nuevaX, int nuevaY) {
                return new Posicion(x+nuevaX, y+nuevaY);
        }

}
```


Marker Interface

Objetivo

- Este es un patrón fundamental de comportamiento.
- Usa interfaces que no declaran métodos ni variables para indicar atributos semánticos en una clase.
- Funciona especialmente bien cuando clases de utilidad deben determinar algo acerca de objetos sin asumir que son instancias de una clase particular, sino solo determinar si han implementado un determinado interface.

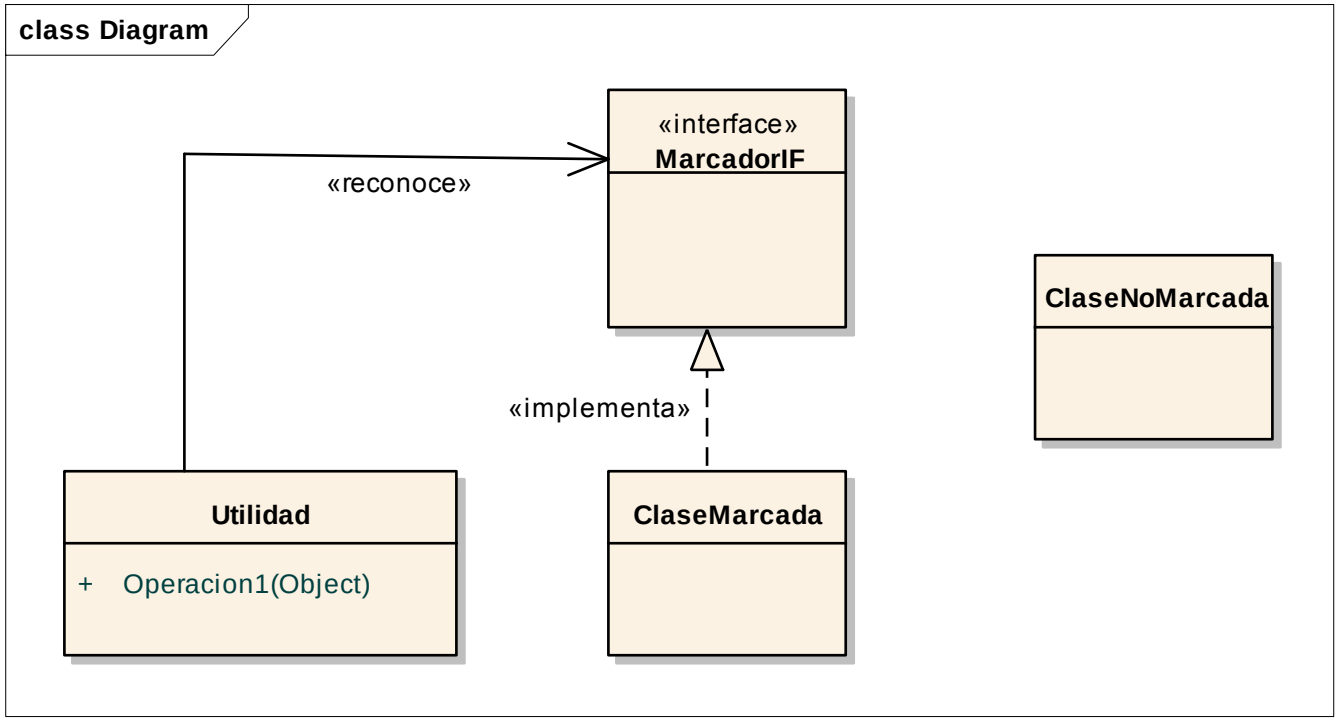
Objetivo

- Este patrón sugiere que cuando una clase A necesite determinar si una instancia de otra clase B está incluida en una determinada clasificación, sin conocer la clase A a la clase B, es suficiente que la clase A determine si la clase B implementa una interfaz marcada. (*Nota: Las clases pueden implementar cualquier número de interfaces*).

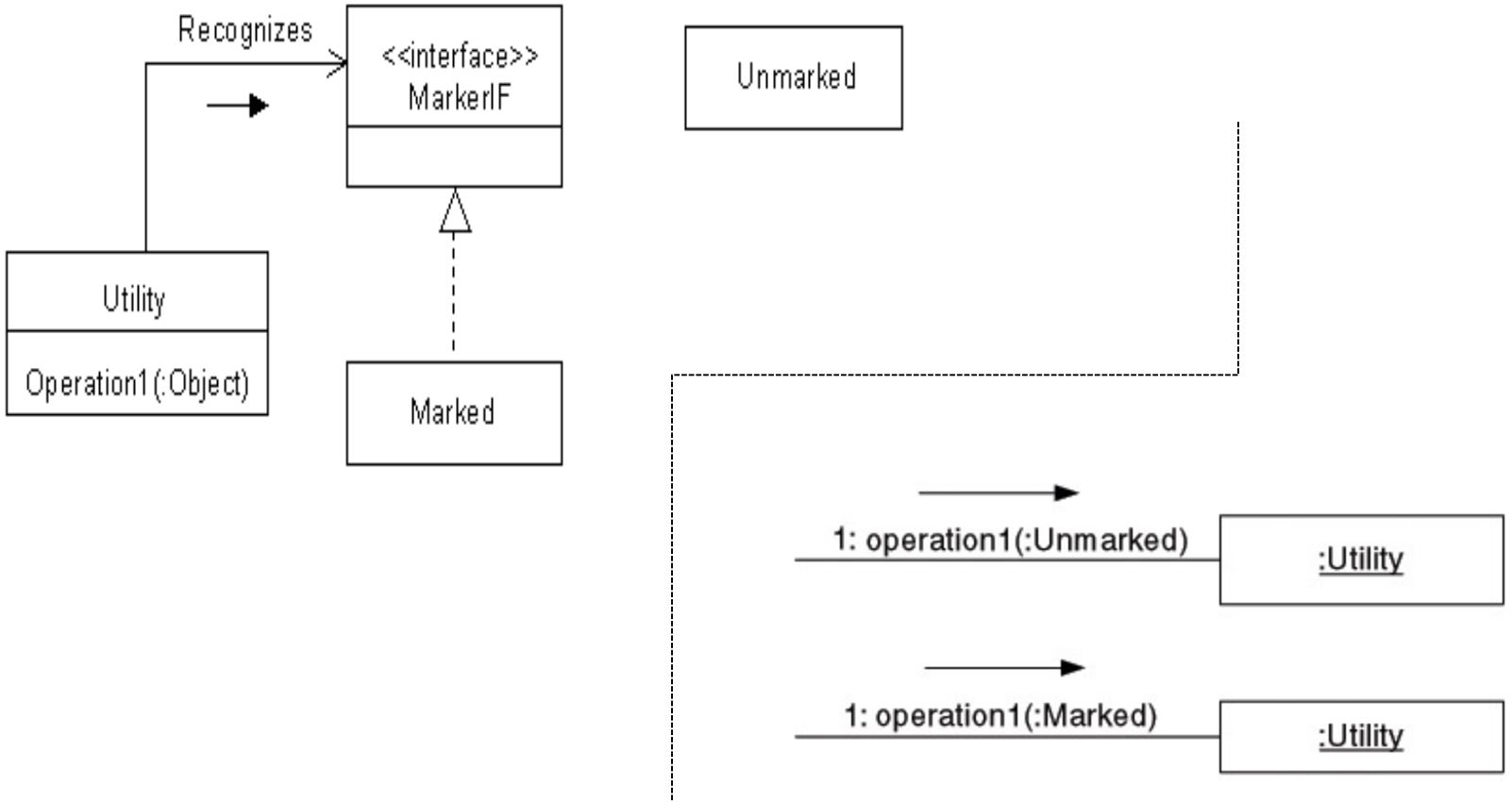
- Ejemplo de uso en Java: Interfaz *Serializable,* interfaz que deben implementar las clases para poder serializar sus objetos.

Estructura

- El diagrama general de las interfaces marcadas se muestra en la siguiente figura.



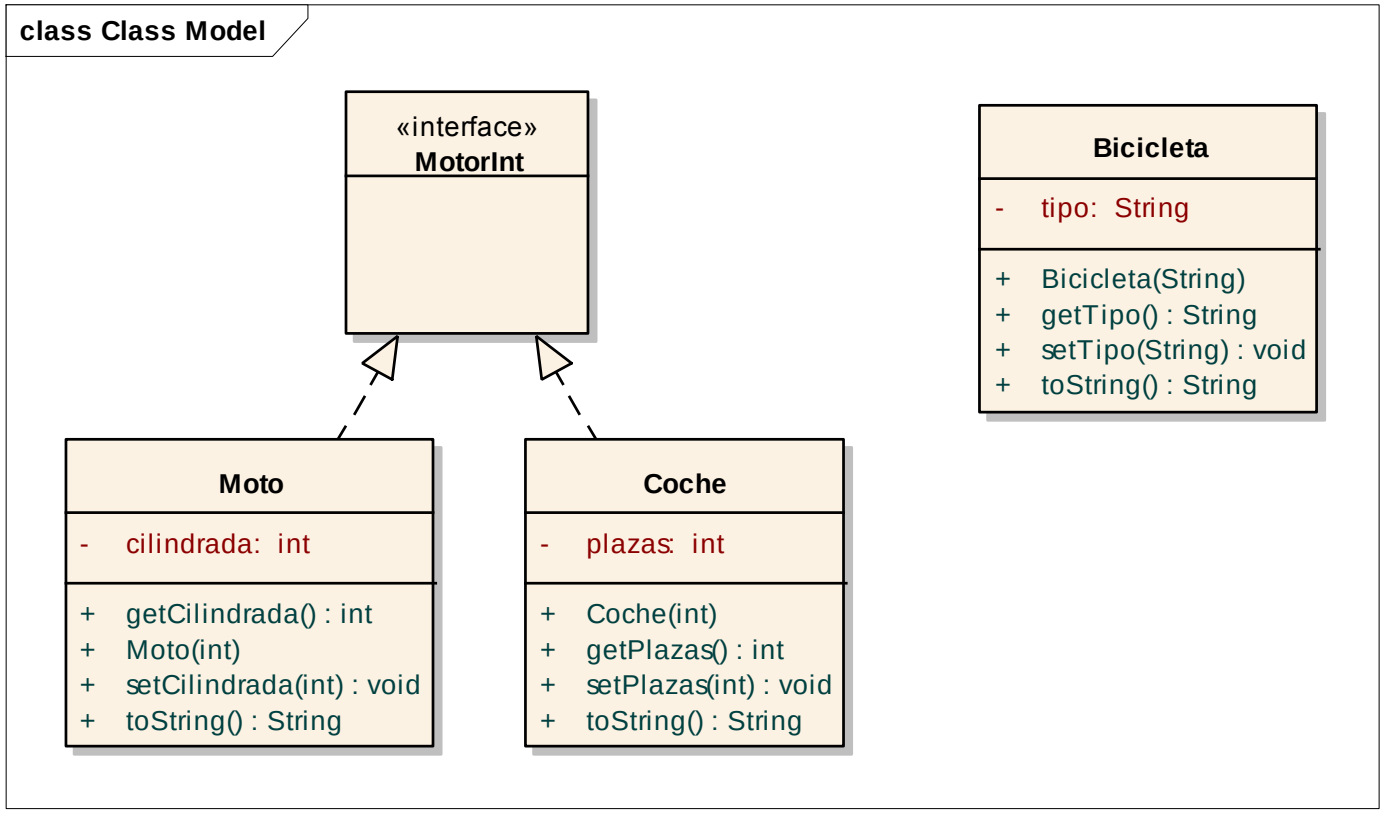
Estructura


Recognizes
<<interface>>
MarkerIF
Unmarked
Utility
Operation1(:Object)
Marked
1: operation1(:Unmarked)
:Utility
1: operation1(:Marked)
:Utility

Código de ejemplo

Marker Interface

Código de ejemplo



Código de ejemplo

- Identificamos a continuación los elementos del patrón:
  - MarcadorIF: MotorInt
  - ClaseMarcada: Moto y Coche

- La clase Bicicleta no implementa la interfaz MotorInt, por lo tanto no está marcada.

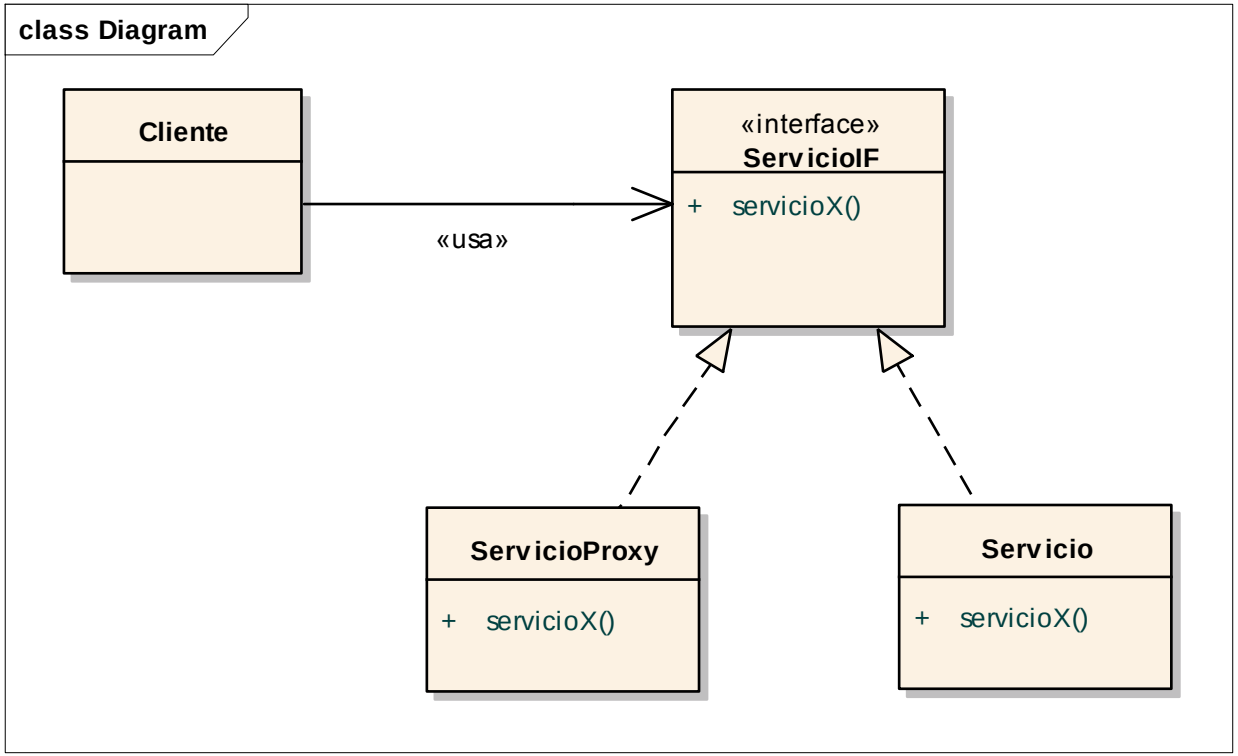
Proxy

Objetivo

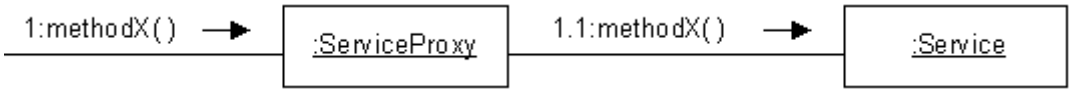
- Este es un patrón estructural fundamental. Es un patrón muy general que ocurre frecuentemente en muchos otros patrones.
- El patrón obliga a que las llamadas a métodos de un objeto ocurra indirectamente a través de un objeto proxy, que actúa como sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.
- Un objeto proxy es un objeto que recibe llamadas a métodos que pertenecen a otros objetos.
- Los objetos clientes llaman a los métodos de los objetos proxy y este invoca los métodos en el objeto específico que provee cada servicio.

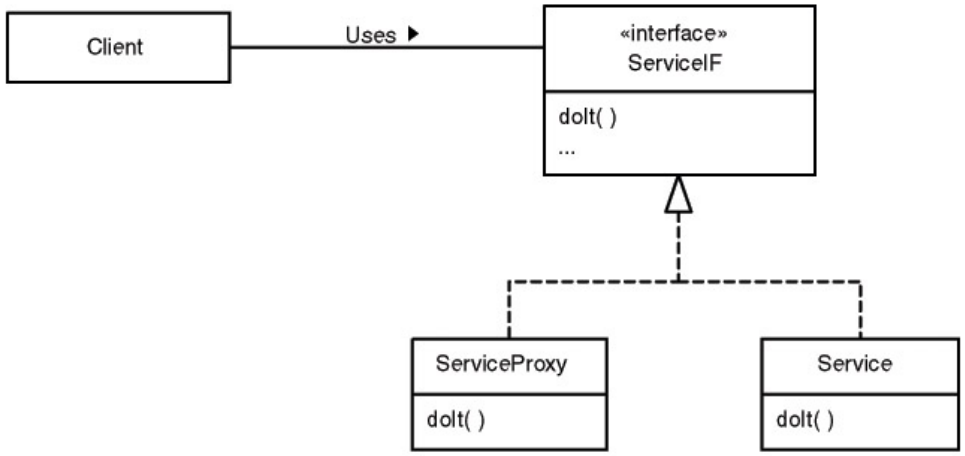
Estructura

- El diagrama general del patrón proxy se muestra en la siguiente figura.



Estructura

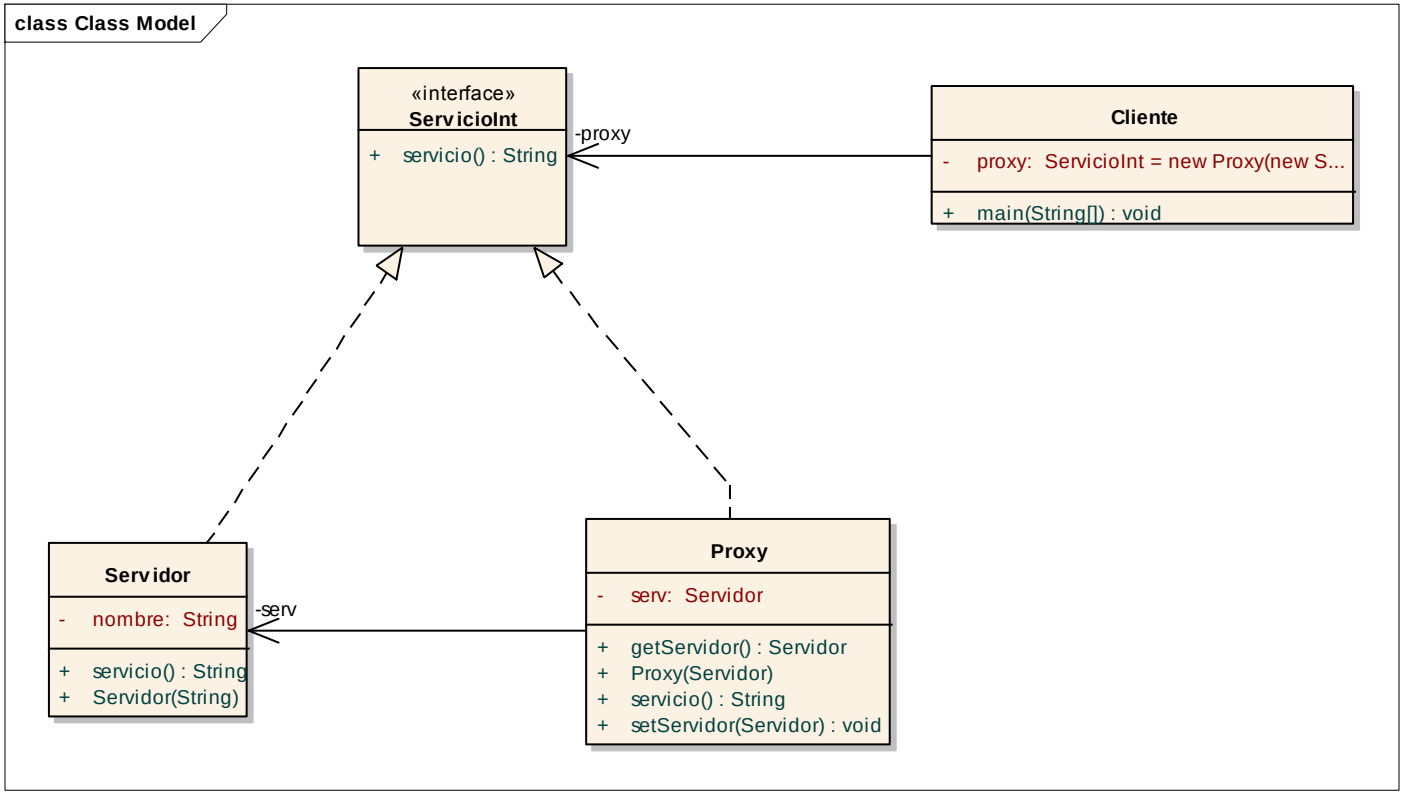

1:methodX( )
:ServiceProxy
1.1:methodX( )
:Service


Client
Uses
«interface»
ServiceIF
doIt( )
...
ServiceProxy
doIt( )
Service
doIt( )

Código de ejemplo

Proxy

Código de ejemplo



Código de ejemplo

- Identificamos a continuación los elementos del patrón:
  - ServicioIF: ServicioInt
  - ServicioProxy: Proxy
  - Servicio: Servidor
  - Cliente: Cliente