

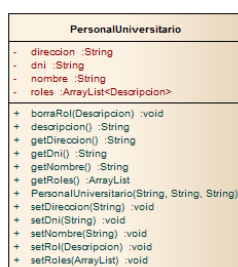
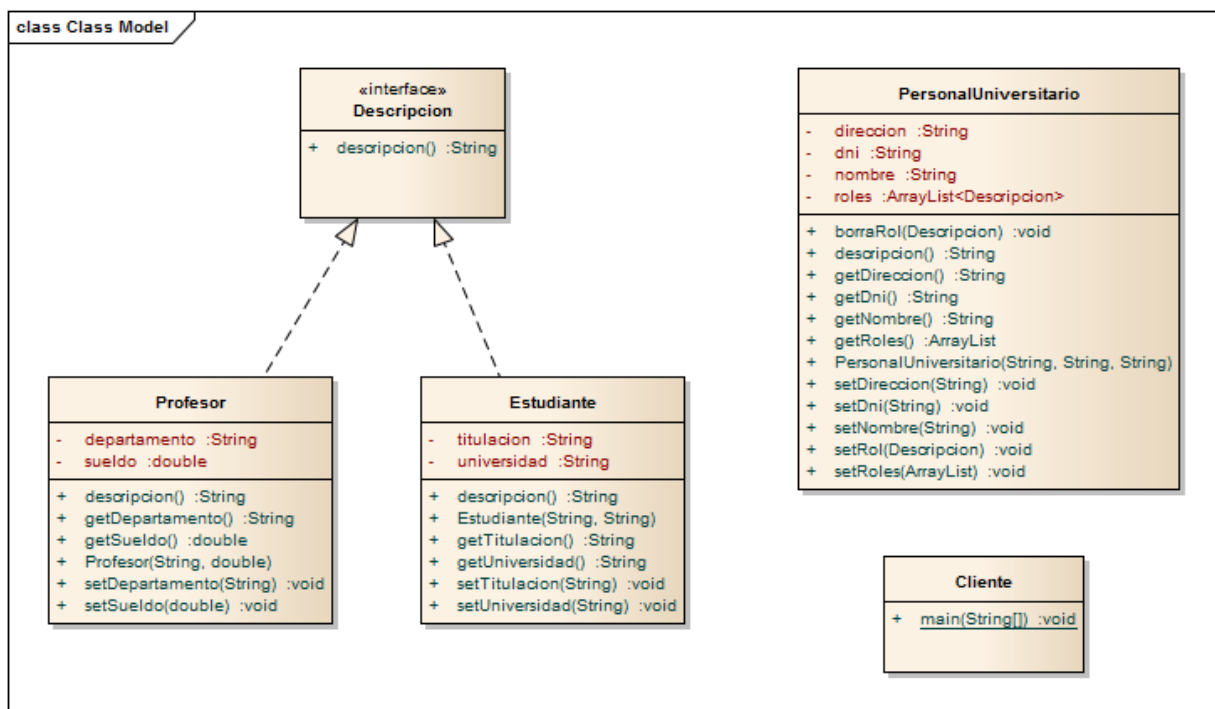
Patrones fundamentales

PATRÓN DELEGATION

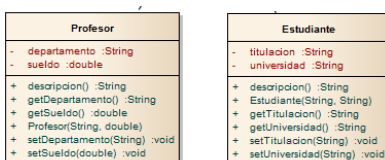
- ¿Cuándo no usar herencia?. Forma de extender y reutilizar la funcionalidad.
- Clase nueva con funcionalidad extra que usa instancias de la clase original para proveer su propia funcionalidad.
- La delegación es una forma de extender el comportamiento de una clase mediante llamadas a métodos de otra clase, más que heredando de ella.



- La clase general (que en una herencia sería el padre como clase abstracta), se transforma en una interface que contiene las interfaces de los métodos que estarán en común en las clases. Así mismo, la clase que sería el padre en la herencia, ahora sería una clase normal, que contiene los atributos comunes más una lista de roles.



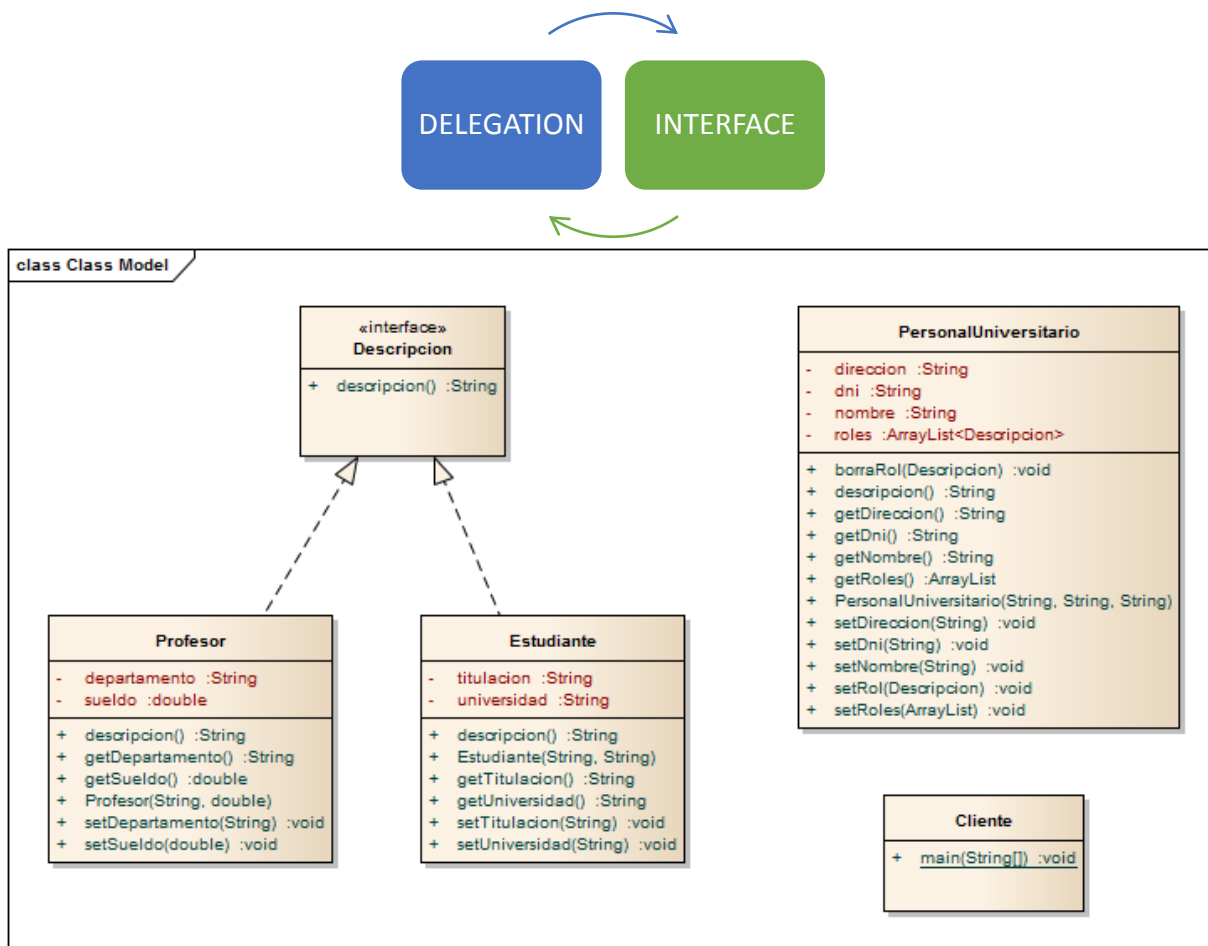
→ Clase general



→ Clases concretas

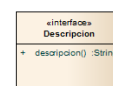
PATRÓN INTERFACE

- La clave es la indirección.
- La interfaz provee a sus clases herederas de métodos específicos.
- La idea es que hay un contrato de servicios, es decir, el cliente sabrá la interfaz de los métodos sin necesariamente saber su codificación.
- La terminología a utilizar será: INTERFAZ, DELEGADOR, DELEGADOS.

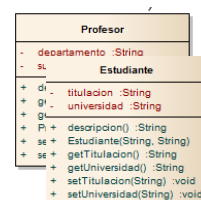


EJEMPLO:

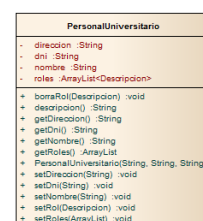
- Interfaz: Descripción (describe las interfaces de los métodos comunes de las clases concretas).
- Delegador: Personal universitario. Esta clase tiene un arreglo de elementos de tipo Descripción. El comportamiento de dichos elementos dependerá de la clase concreta.
- Delegado: Cada una de las clases concretas que realizan la interfaz.



Interfaz

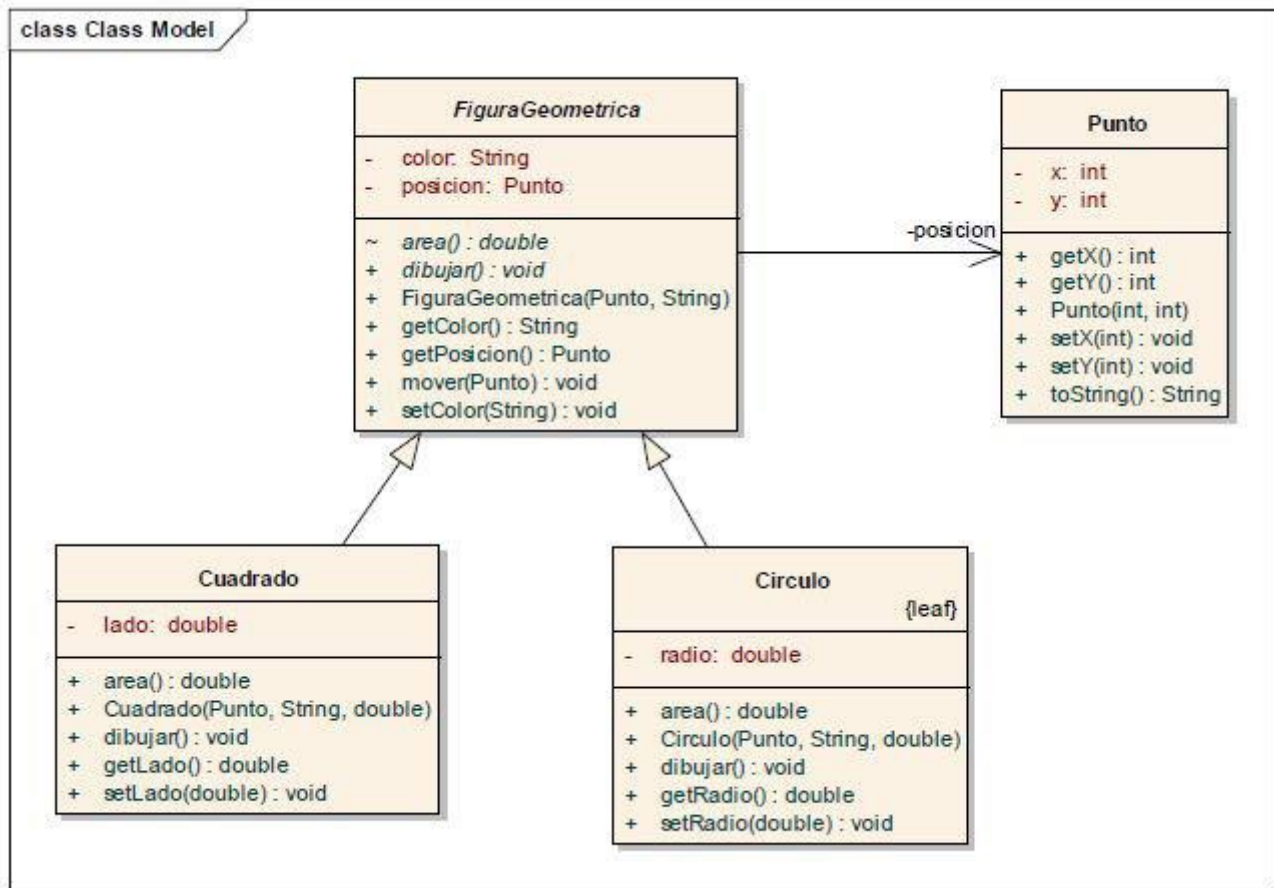


Clases delegadas



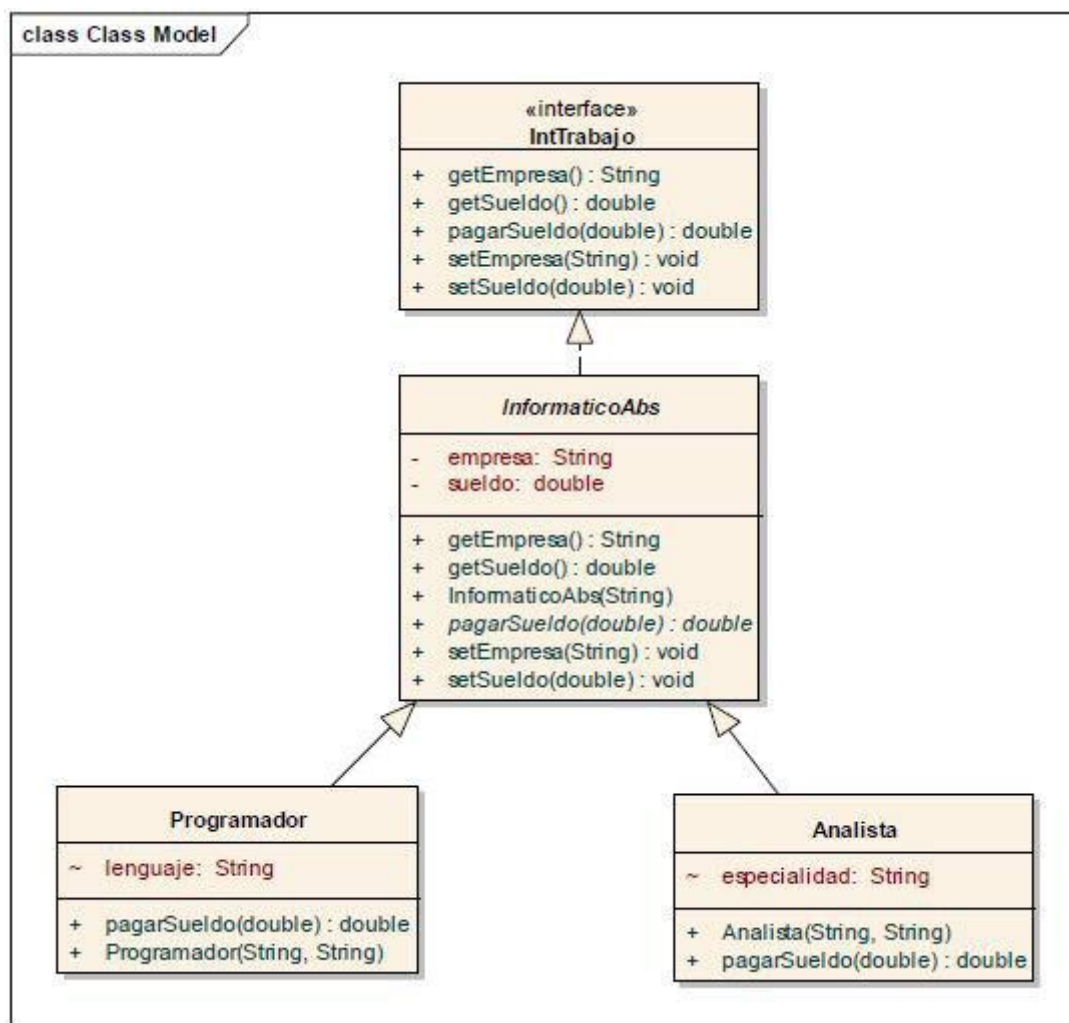
Clase delegadora

- Se proporciona una clase padre abstracta.
- Utiliza la idea de POLIMORFISMO.
- Organiza el comportamiento común de las clases relacionadas con la superclase abstracta.



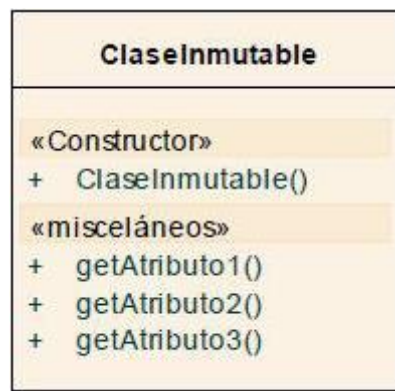
PATRÓN INTERFACE Y ABSTRACT CLASS

- Se proporciona una clase padre abstracta.
- Dicha clase abstracta es ocultada al cliente utilizando una interfaz de por medio.
- El cliente dispondrá de un "catálogo" de servicios, que son las interfaces puestas en el interface. Además, se tendrá la garantía de que las clases concretas implementan dichos servicios.



- Es un patrón de concurrencia.
- Incrementa la robustez y reduce el costo.
- Aconsejable en casos en los que sea mucho más prioritaria la consulta que la escritura.
- Se mantiene una única instancia del objeto.
- Si se accede a este recurso de forma asíncrona, al estar varios hilos en condición de carrera, se deben establecer **mecanismos de sincronización** para evitar la **corrupción de los datos**.
 - El patrón inmutable permite el acceso sin esta necesidad a cualquier hilo, ya que se garantiza que no se modificará el estado de la clase que implementa el patrón inmutable.
 - Para conseguir esto, solo debe haber getters en la clase inmutable y todos los atributos deben ser privados.

Solo getters → PATRÓN INMUTABLE

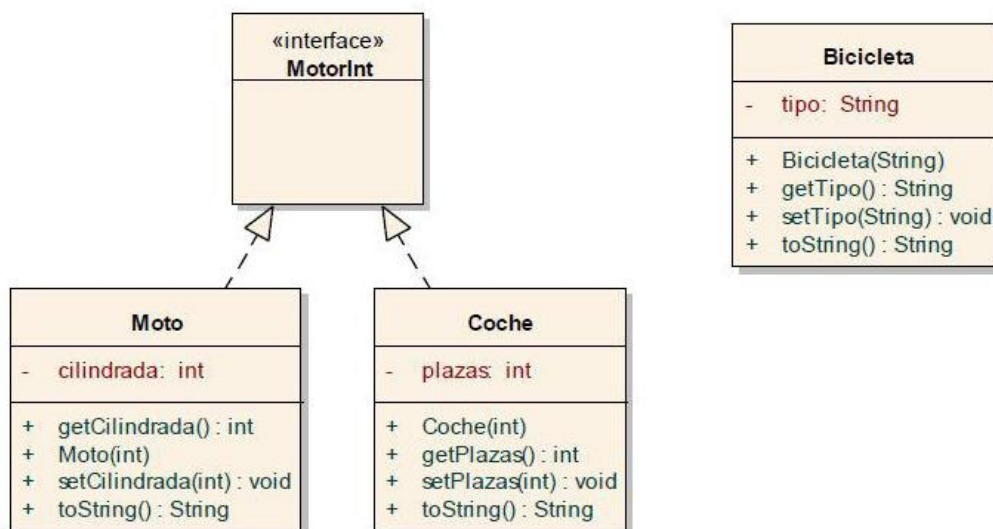


```
class Posicion {  
  
    private int x;  
    private int y;  
  
    public Posicion(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
  
    public Posicion desplazar(int nuevaX, int nuevaY) {  
        return new Posicion(x+nuevaX, y+nuevaY);  
    }  
}
```

- 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, si no solo determinar si han implementado un determinado interface.

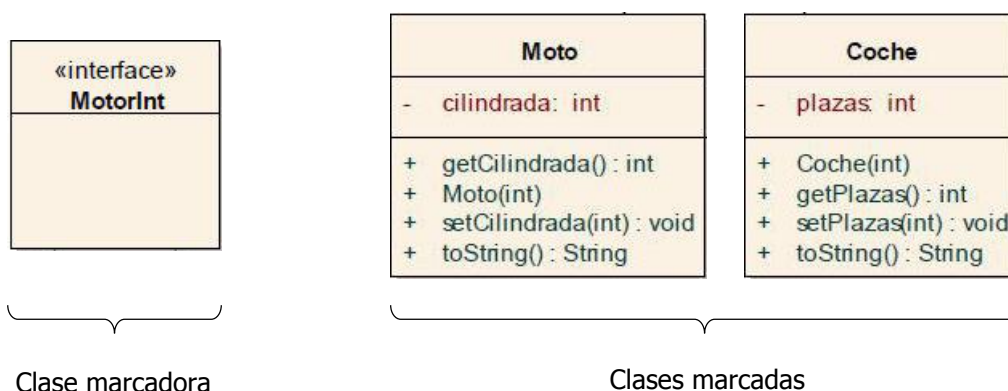


Interface sin métodos ni atributos → PATRÓN MARKER INTERFACE



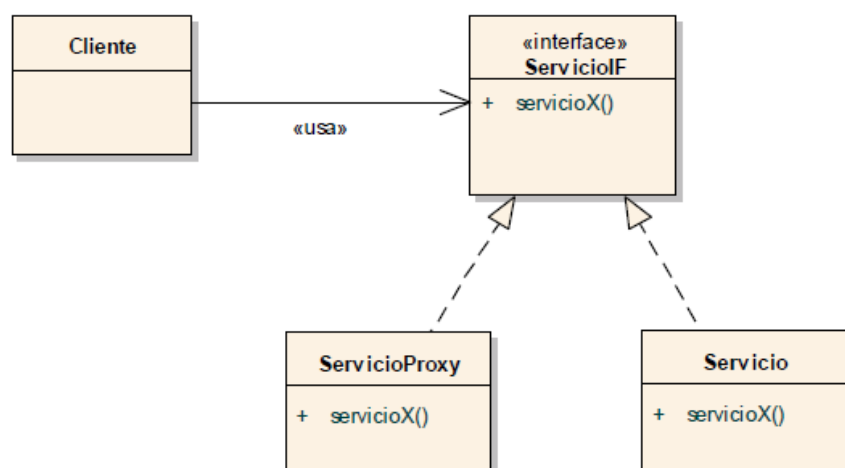
EJEMPLO:

- Clase marcadora: **MotorInt**
- Clases marcadas: **Moto y Coche**

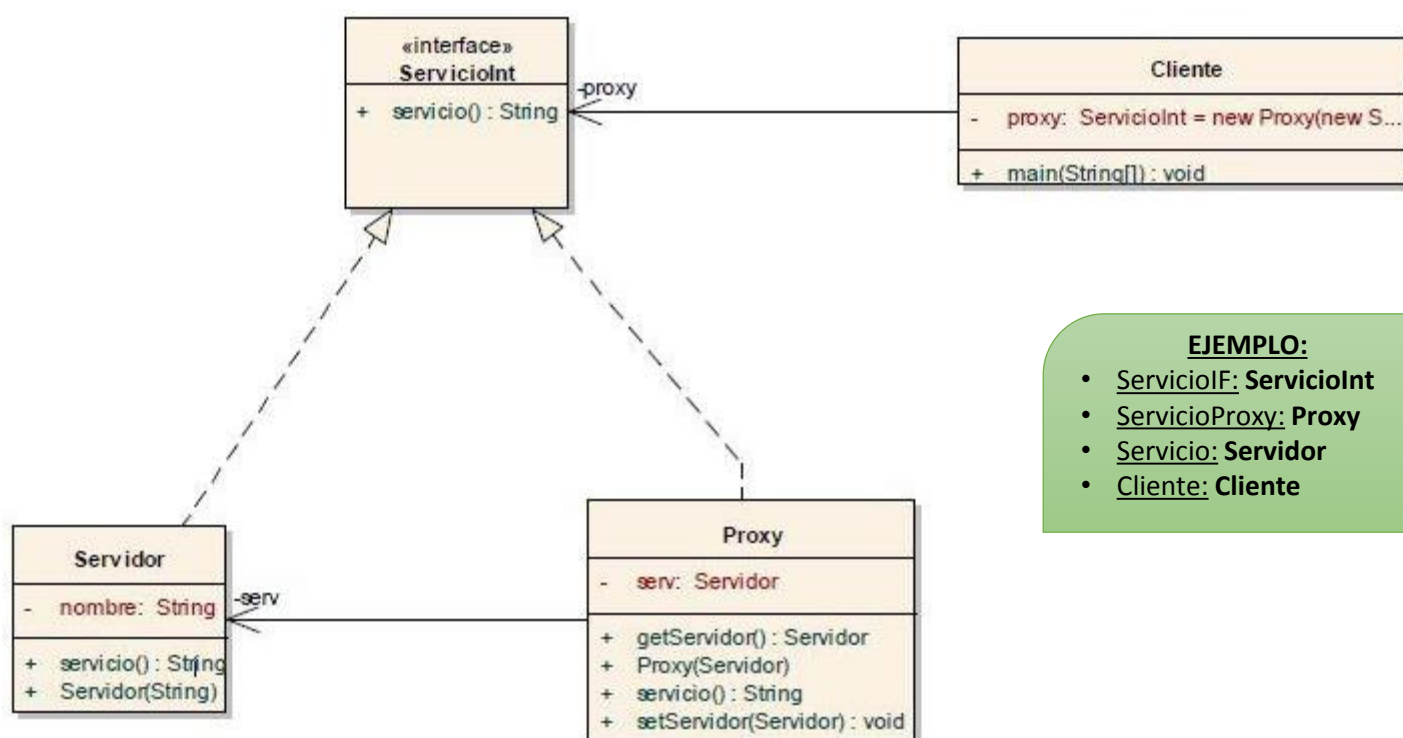


- El patrón obliga a que las llamadas a métodos de un objeto ocurra indirectamente a través de un objeto proxy, que actúa como sustituto del objeto original, delegando luego las llamadas a los métodos de los objetos respectivos.
- Un objeto proxy es un objeto que recibe llamadas a métodos que pertenecen 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.

Objeto que da servicios. No se ve implementación → PATRÓN PROXY



La idea consiste en un objeto, que delega las llamadas a métodos que pertenecen a otros objetos. Se decide que funciones se invocarán en tiempo de ejecución.



EJEMPLO:

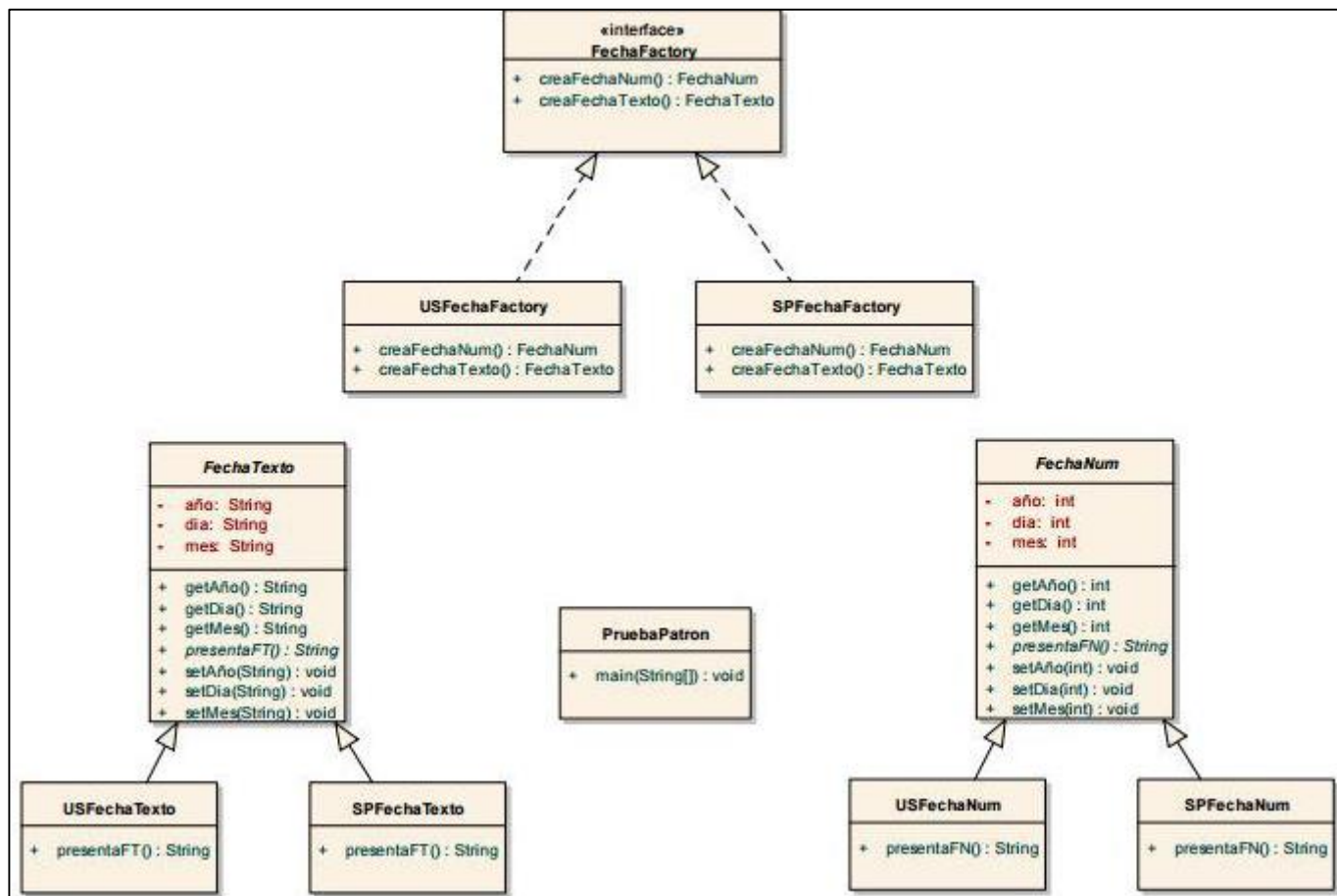
- ServicioIF: **ServicioInt**
- ServicioProxy: **Proxy**
- Servicio: **Servidor**
- Cliente: **Cliente**

Patrones de creación

PATRÓN ABSTRACT FACTORY

- Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar clases concretas.
 - Cliente debe ser independiente del proceso de creación de los productos.
 - Es necesario crear los objetos como un conjunto, de forma que sean compatibles.
 - Colección que revela clases que revela interfaces y relaciones, pero no sus implementaciones.

Familias de productos. Revelan interfaces y relaciones, pero no implementaciones



EJEMPLO:

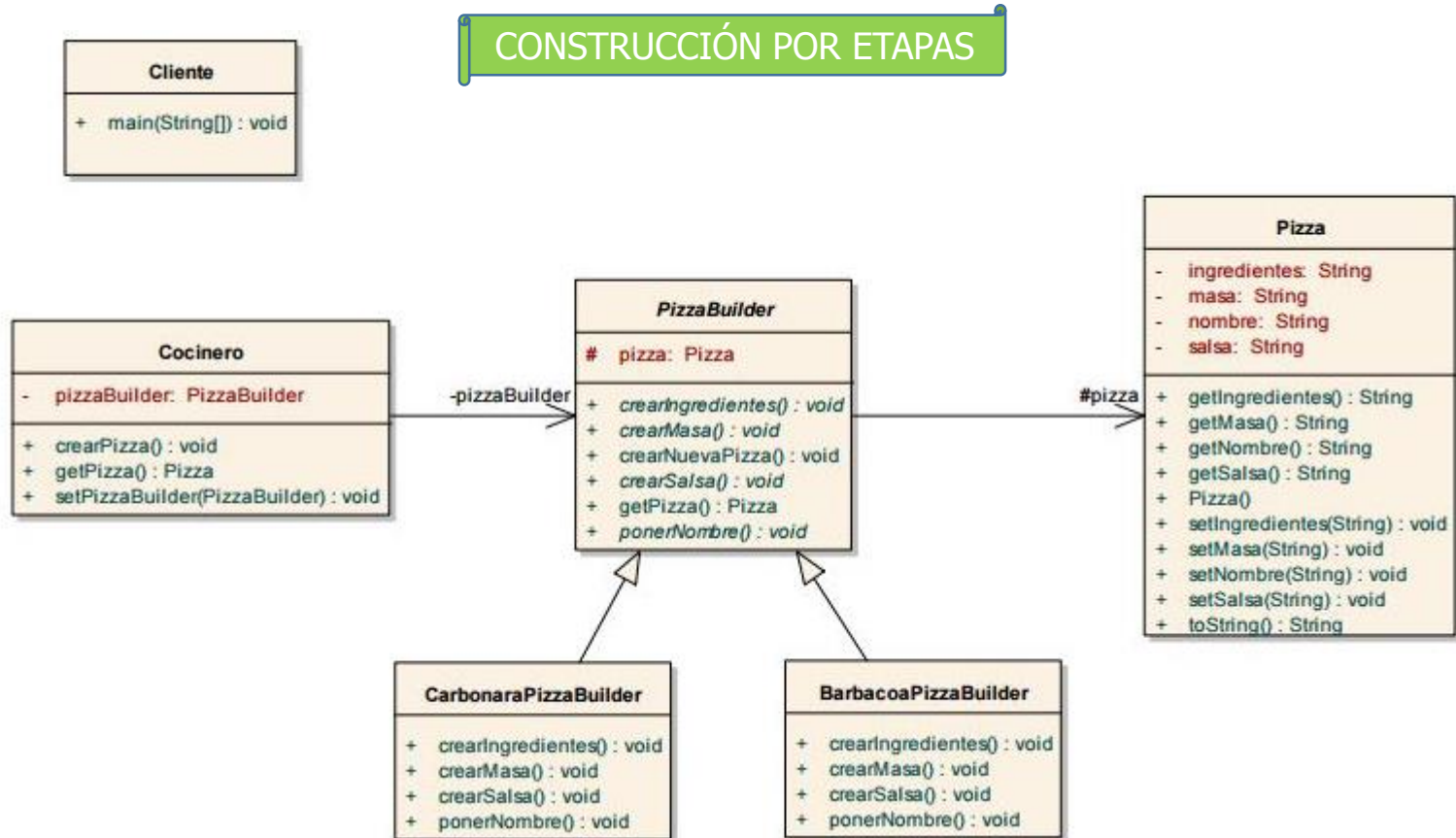
- Fábrica abstracta: **FechaFactory**
- Fábricas concretas: **USFechaFactory** y **SPFechaFactory**
- Producto abstracto: **FechaTexto** y **FechaNum**
- Producto concreto: **USFechaTexto**, **SPFechaTexto**, **USFechaNum** y **SPFechaNum**
- Cliente: **PruebaPatrón**

Las consecuencias del patrón son:

1. Aisla las clases concretas.
2. Facilita el intercambio de familias de productos.
3. Promueve la consistencia entre productos.
4. Dificulta la incorporación de nuevos tipos de productos.

PATRÓN BUILDER

- Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción puede crear diferentes representaciones.
- Se utiliza cuando:
 - Una clase tiene una estructura compleja.
 - Atributos dependientes entre sí.
 - La creación del objeto, lleva unas etapas, bien definidas, en las que según el objeto se llevarán a cabo unas u otras acciones, que darán lugar a diferentes representaciones del objeto.



EJEMPLO:

- Constructor: **PizzaBuilder**
- Constructor concreto: **CarbonaraPizzaBuilder** y **BarbacoaPizzaBuilder**
- Producto: **Pizza**
- Director: **Cocinero**
- Cliente: **Cliente**

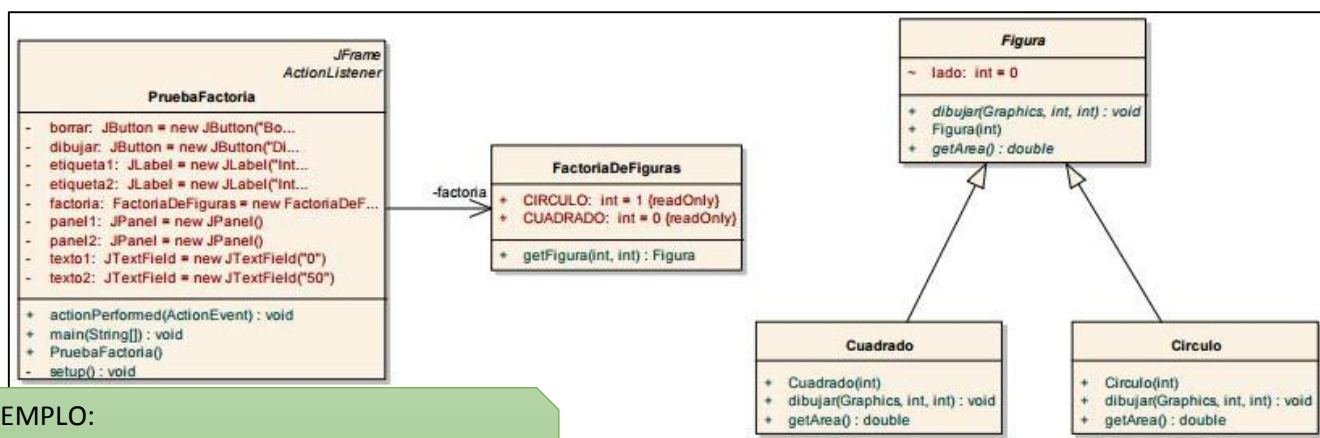
Las consecuencias del patrón son:

1. Permite variar la representación interna de un producto.
2. Aísla el código de construcción y representación.
3. Control más fino sobre el proceso de construcción.
4. Reduce el acoplamiento.

PATRÓN FACTORY METHOD

- Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación del objeto.
 - Se utiliza cuando:
 - Una clase no puede (o no cree conveniente) prever la clase de objetos que debe crear, por lo que delega esta acción en las subclases.
 - Se sabe cuándo crear un objeto, pero desconoce su tipo.
 - Se desea crear un framework extensible.
 - Se necesitan algunos constructores sobrecargados con la misma lista de parámetros.
- PROHIBIDO EN JAVA.** Utilizar varios métodos de fabricación con distinto nombre.
- Atributos dependientes entre sí.

Delegar la construcción de objetos



EJEMPLO:

- Producto: **Figura**
- Productos concretos: **Cuadrado y Circulo**
- Creador: **FactoriaDeFiguras**

VARIACIONES

- El creador puede proporcionar una implementación estándar, para los métodos de fabricación. Por ello, Creador no tiene que ser una clase abstracta o una interfaz, sino una clase completamente definida.
- El producto puede ser implementado como una clase abstracta, al ser ya una clase, puede incluir implementaciones para los otros métodos.
- El método de fabricación, puede recibir un parámetro tipo, reduciendo así el número de métodos de fabricación.

Las consecuencias del patrón son:

- Proporciona enlaces para las subclases.
- Conecta jerarquías de clases paralelas.

PATRÓN PROTOTYPE

- Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.
- Este patrón se usa en los casos en los que crear una instancia de una clase sea un proceso muy complejo y requiera mucho tiempo. Lo que hace es crear una instancia original, y, cuando necesitemos otra, en lugar de volver a crearla, copiar esa original y modificarla.
- Se utiliza cuando:
 - Crear una instancia de una clase sea muy complejo y tarda mucho, lo que hace es crear una instancia original y cuando necesitemos otra, copia esa original y la modifica.
 - Un sistema debe ser independiente de cómo se crean, se componen y se representan sus productos.
 - Las clases a instanciar sean especificadas en tiempo de ejecución, para evitar construir una jerarquía de clases de fábrica paralela a la jerarquía de clases de los productos.
 - Las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes. Puede ser más adecuado tener un número equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de cada clase cada vez con el estado apropiado.

Delegar la construcción de objetos

VARIACIONES

- **Constructor de copia:** Este tipo de constructores toma una instancia de la misma clase como parámetro y devuelve una nueva copia con los mismos valores que el parámetro. Es posible tener un constructor al que se le pasan dos parámetros, el objeto a ser copiado y un valor booleano para indicar si debe aplicarse copia superficial o profunda.
- **Método clone:** Java define un método en la clase Object llamado clone para hacer copias de objetos.

CONSEJOS DE IMPLEMENTACIÓN

- Todas las clases en Java heredan un método de la clase Object llamado clone. Un método clone de un objeto retorna una copia de ese objeto. Esto solamente se hace para instancias de clases que dan permiso para ser clonadas.
- El método clone() es declarado "protected". Si necesitamos hacer que un objeto pueda ser clonado por cualquiera, tenemos que redefinir su método clone() y hacerlo público.
- Una clase da permiso para que su instancia sea clonada si, y solo si, ella implementa el interface Cloneable..
- Copia superficial: Significa que las variables de los objetos clonados contiene los mismos valores que las variables del objeto original y que todas las referencias al objeto son a los mismos objetos. Es decir, la copia superficial copia solamente el objeto que será clonado, no los objetos a los que se refiere.
- Copia profunda: Significa que las variables de los objetos clonados contienen los mismos valores que las variables del objeto original, excepto que estas variables que se refieren a objetos realizan copias de los objetos referenciados por el objeto original. La copia profunda copia el objeto que será clonado y los objetos a los que referencia.
- Implementar la copia profunda puede ser delicado. Se necesita decidir si se quiere hacer copia profunda o superficial de los objetos copiados indirectamente. También hay que tener cuidado con el manejo de las referencias circulares.

Las consecuencias del patrón son:

1. El Prototipo aísla las clases de productos concretos del cliente. Esto reduce el número de clases conocidas por el cliente y le permite trabajar con diferentes clases concretas sin ser modificado.
2. Podemos añadir y quitar productos en tiempo de ejecución, según se vayan necesitando, mediante la clonación. Podemos ajustar la representación interna de los valores de una clase en tiempo de ejecución basándonos en las condiciones del programa.
3. Otra manera de crear nuevos objetos es variando sus estructuras. Si implementamos clone como copia profunda, se pueden añadir diferentes estructuras al conjunto de prototipos disponibles.
4. Si tenemos clases de prototipo que copiar implica que tenemos el acceso suficiente a sus valores o métodos para cambiarlas después de ser clonadas. Puede necesitar añadir métodos de acceso.
5. Se puede crear un registro de clases prototípicas que pueden ser clonadas y pedir al objeto registro una lista de los posibles prototipos. A veces es mejor clonar que escribir desde cero.
6. Reduce la herencia: Permite clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto. Por tanto no es necesaria una jerarquía de clases creador.
7. El principal inconveniente de este patrón es que cada subclase de Prototipo debe implementar la operación clonar, lo cual puede ser difícil.

PATRÓN SINGLETON

- Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.
- Se utiliza cuando:
 - Sólo puede haber una instancia de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido.
 - La única instancia debería ser extensible por herencia y los clientes deberían poder usar una instancia extendida sin modificar su código.

Una clase solo tiene una instancia. Un solo punto de acceso global a ella.

Singleton
- unicaInstancia: Singleton - datosDelSingleton
+ Instancia(): Singleton + OperacionDelSingleton() + ObetenerDatosDelSingleton()

El método estático Instancia devuelve unicaInstancia

```

Instancia(){
    return unicaInstancia;
}
  
```

- Singleton:** Define una operación Instancia que permite a los clientes acceder a su única instancia. Instancia es una operación estática (de clase).
- Puede ser responsable de crear su única instancia.

VARIACIONES

- Se puede tener más de una instancia dentro de la clase. Aquellos objetos que conocen la existencia de múltiples instancias pueden utilizar algún método para obtener una de ellas. **Ventaja: el resto de la aplicación permanece inalterada.**
- El método de acceso a **Singleton** puede ser el punto de acceso al conjunto total de instancias, aunque todas tengan un tipo distinto. El método de acceso puede determinar, en tiempo de ejecución, qué tipo de instancia devolver.

```

public class SingletonOpt
{
    private static SingletonOpt instancia = new SingletonOpt();

    private SingletonOpt() {}

    public static SingletonOpt getInstancia()
    {
        return instancia;
    }
}
  
```

Las consecuencias del patrón son:

- Existe exactamente una instancia de una clase **Singleton**.
- Acceso controlado a la única distancia.** Otras clases que quieran una referencia a la única instancia de la clase **Singleton** conseguirán esa instancia llamando al método estático **getInstancia()** de la clase.
- Espacio de nombres reducido.** Este patrón es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenen las instancias.
- Permite un número variable de instancias.** El patrón permite la creación de más de una instancia de la clase **Singleton**, facilitando el control del número de instancias usadas en la aplicación. Solo hay que cambiar la operación de acceso a la instancia del **Singleton**.
- Tener subclases de una clase **Singleton** es complicado y resultan clases imperfectamente encapsuladas. Una clase derivada de un **Singleton** no es un **Singleton**. Para hacer subclases de una clase **Singleton**, se debería tener un constructor que no sea privado.

ContadorLogin
- instancia: ContadorLogin - usuarios: ArrayList<String> = new ArrayList<S...
+ borrarLogin(String) : void - ContadorLogin() + devolverEstadoCuenta(String) : boolean + getInstancia() : ContadorLogin

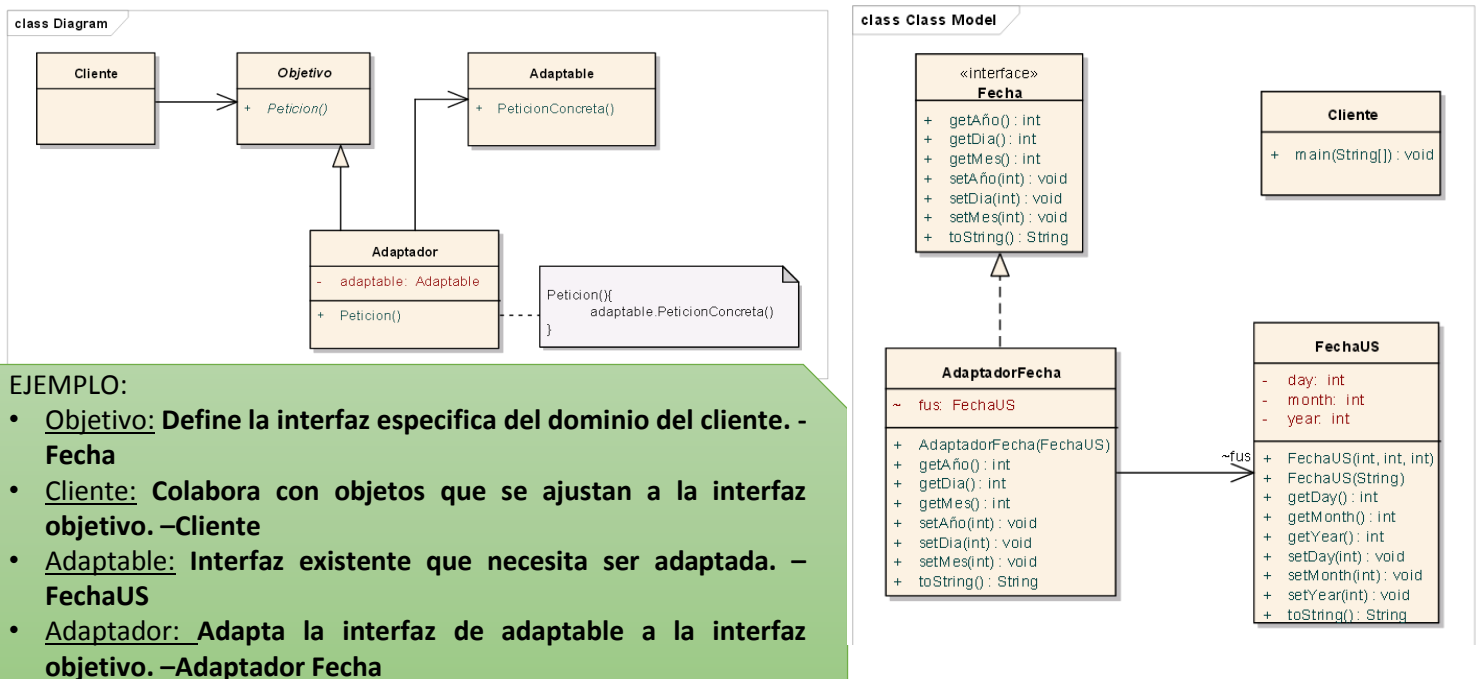
ControlEntrada
- entrada: BufferedReader = new BufferedRea... - res: String - usuario: String
+ main(String[]) : void

Patrones estructurales

PATRÓN ADAPTER (Wrapper - encapsulamiento)

- Sirve de intermediario entre dos clases, convirtiendo las interfaces de una clase para que pueda ser utilizada por otra.
- Se utiliza cuando:
 - Hay una clase existente y su interfaz no concuerda con la que se necesita.
 - Traducción entre interfaces de varios objetos.
 - Existe un objeto intermediario reutilizable que se encarga de gestionar todas las clases pero no es hasta tiempo de ejecución cuando se elige la clase a utilizar.
 - Adaptador de objetos.

Cooperación de clases con interfaces incompatibles



VARIACIONES

- Un adaptador y múltiples adaptables.
- Adaptadores no basados en interfaz ?.
- Capa de interfaz entre el invocador y adaptador, otra entre el adaptador y el adaptable: La capa entre el invocador y el adaptador permite que se puedan introducir fácilmente nuevos adaptadores en el sistema en tiempo de ejecución. Y entre el adaptador y el adaptable hace que los adaptadores sean cargados dinámicamente en tiempo de ejecución.

Las consecuencias del patrón son:

- El cliente es independiente de las clases finales que utiliza.
- Es posible utilizar la clase adaptador para monitorizar que clases llaman a que métodos de las clases finales.
- En el **adaptador de clase**, la clase adaptadora...
 - Puede redefinir y ampliar la interfaz de la clase adaptada.
 - Adapta una clase adaptable a Objetivo, pero se refiere únicamente a una clase adaptable concreta, por lo tanto no nos servirá para todas sus subclases.
- En el adaptador de objeto, la clase adaptadora...
 - Permite que un mismo adaptador funcione con muchos adaptables
 - Puede utilizar todas las subclases de la clase adaptada, ya que tiene constancia de los objetos que instancia.
 - Hace que sea más difícil redefinir el comportamiento de adaptable. Se necesitará crear una subclase de adaptable y hacer que el adaptador se refiera a la subclase, en vez de a la clase adaptable en sí.

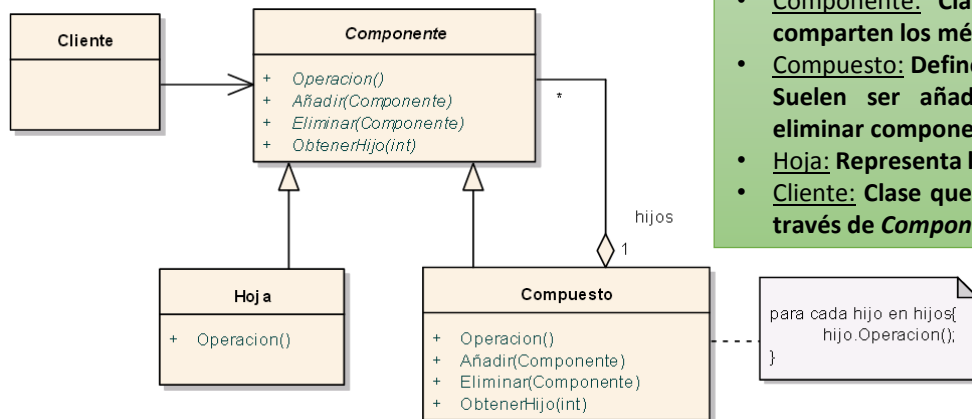
- ## Nexo entre abstracción funcional (caja negra) e implementación interna (lo que hay dentro)
-
- ```
classDiagram
 class Cliente
 class Abstraccion {
 + Operacion()
 }
 class Implementador {
 + OperacionImp()
 }
 class AbstraccionRefinada {
 + Operacion()
 }
 class ImplementadorConcretoA {
 + OperacionImp()
 }
 class ImplementadorConcretoB {
 + OperacionImp()
 }
 Cliente --> Abstraccion
 Abstraccion <|-- AbstraccionRefinada
 Implementador <|-- ImplementadorConcretoA
 Implementador <|-- ImplementadorConcretoB
 Abstraccion o-- Implementador
 Abstraccion ..> Implementador : Operacion(){ implementador.OperacionImp(); }
```
- **Abstracción:** Define la interfaz que el cliente necesita. **Mantiene una referencia al objeto Implementador.**
  - **Abstracción refinada:** Extiende la Abstracción para proporcionar una implementación específica.
  - **Implementador:** Define la interfaz que las implementaciones concretas deben seguir. **No incluye la implementación.**
  - **ImplementadorConcreto:** Implementa la interfaz Implementador y define su propia implementación.

- Abstracción: **OfertasAbst**
- Abstracción refinada: **OfertasAbstRef**
- Implementador: **OfertasImp**
- ImplementadorConcreto:  
**OfertasImpCiudad, OfertasImpEmpresa,**  
**OfertasImpFecha, OfertasImpOcupacion,**  
**OfertasImpSalario**

# PATRÓN COMPOSITE (Handle/Body)

- Construir objetos de complejidad mayor mediante otros más sencillos, de forma recursiva, formando una jerarquía en estructura de árbol. Permite que todos los elementos individuales o compuestos se traten de forma uniforme por los clientes.
- Se utiliza cuando:
  - Tenga que representar un componente modelado como una estructura rama-hoja (o parte-todo o contenedor-contenido).
  - La estructura puede tener cualquier nivel de complejidad y dinamismo.
  - Quiera tratar de forma uniforme toda la estructura del componente, utilizando operaciones comunes para toda la jerarquía.
  - Desea que los clientes sean capaces de obviar las diferencias entre objetos compuestos individuales.

Construir de forma recursiva objetos complejos de forma sencilla (árbol compuesto de hojas)



- Componente:** Clase abstracta común a todos los objetos que comparten los métodos y operaciones de esta clase.
- Compuesto:** Define los métodos propios de los objetos compuestos. Suelen ser añadir componentes (individuales o compuestos), eliminar componentes y recuperar objetos que lo componen.
- Hoja:** Representa las clases individuales, no compuestas.
- Cliente:** Clase que crea y manipula los objetos de la composición a través de *Componente*.

## VARIACIONES

- Nodo raíz:** Sirve para mejorar la manejabilidad del sistema, en ocasiones se crea un objeto que actúa como base para la jerarquía completa de objetos *Composite*. Si el objeto raíz se representa como una clase separada, se puede implementar con un *Singleton*, garantizando su acceso.
- Ramificación basada en reglas:** En estructuras complejas (múltiples nodos y ramas) se puede interponer una serie de reglas para determinar como unir tipos de nodos a ciertas ramas.

## Las consecuencias del patrón son:

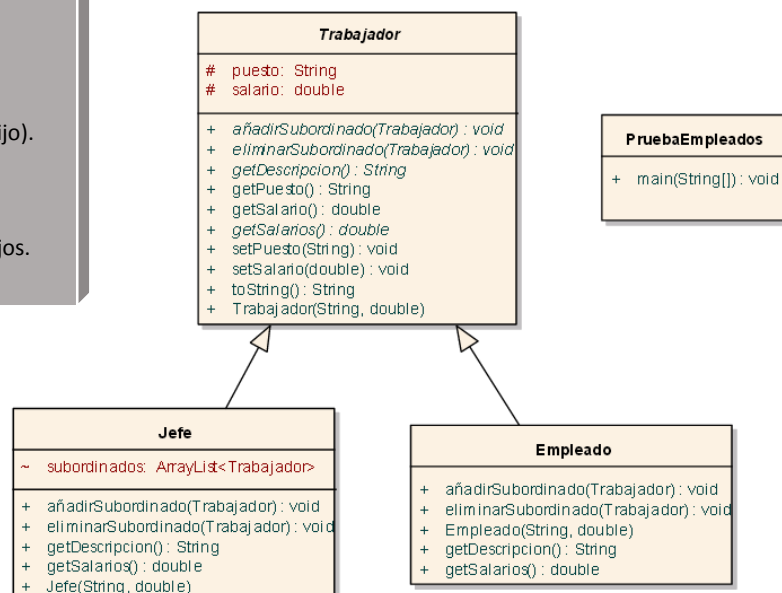
- Se crea una jerarquía de objetos básicos y de composiciones de estos, de forma recursiva.
- Simplifica el cliente, al tratar todos los objetos de la misma forma.
- Facilita agregar nuevas clases de componentes insertándolas en la jerarquía de clases como hijas de compuesto u hojas. Los clientes siguen siendo compatibles con la nueva estructura.
- Desventaja:** Generaliza el diseño haciendo difícil restringir los componentes de un objeto compuesto. No podemos confiar en el sistema de tipos y se deben hacer comprobaciones en tiempo de ejecución (con `instanceOf()`)

## CONSEJOS DE IMPLEMENTACIÓN

- Gestionar los enlaces al padre.
- Estudiar si se desea una agrupación u objetos simples.
- Operaciones de gestión de nodos hijos (añadir, eliminar y obtener hijo).
- Ordenar hijos (en caso de interés)
- Dstrucción de objetos.
- Utilización de iteradores para recorrer los integrantes.
- Visitor** puede realizar operaciones para evitar distribuirlas por los hijos.

### EJEMPLO:

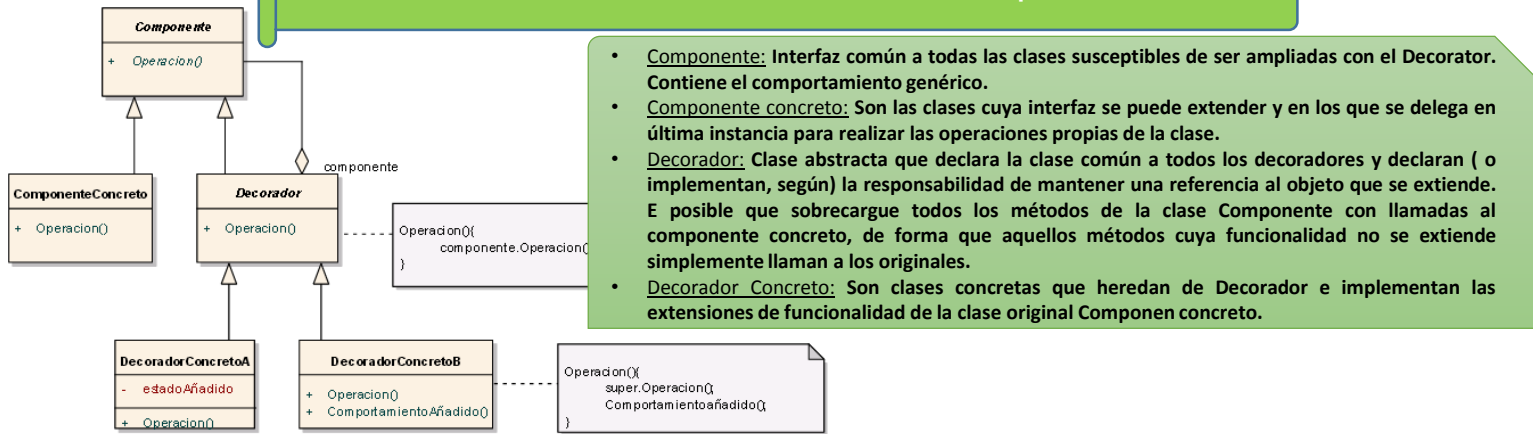
- Componente:** `Trabajador`
- Compuesto:** `Jefe`
- Hoja:** `Empleado`
- Cliente:** `PruebaEmpleados`



# PATRÓN DECORATOR (Wrapper - envoltorio)

- Añadir nuevas responsabilidades dinámicamente a un objeto sin modificar su apariencia externa o su función, es una alternativa a crear demasiadas subclases por herencia.
- Se utiliza cuando:
  - Para añadir funcionalidad a una clase sin las restricciones que implica la utilización de la herencia.
  - Cuando se quiera añadir funcionalidad a una clase de forma dinámica en tiempo de ejecución y que sea transparente a los usuarios.
  - Haya características que varíen independientemente, que deban ser aplicadas de forma dinámica y que se pueden combinar arbitrariamente sobre un componente.

Alternativa a crear demasiadas subclases por herencia.

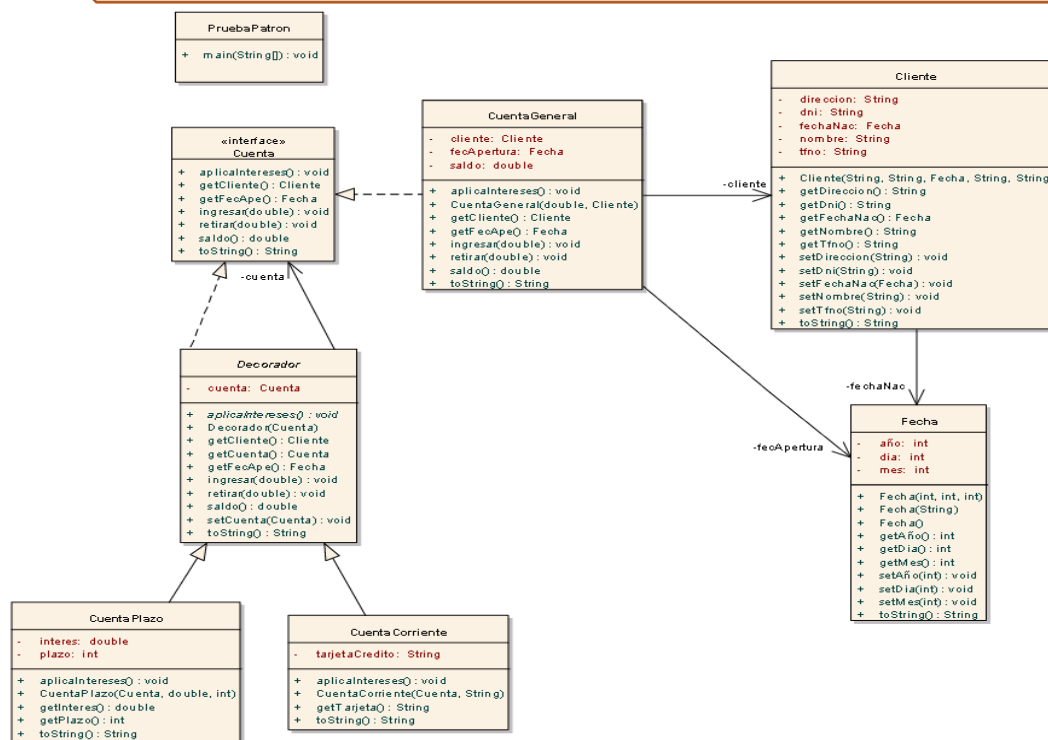


## VARIACIONES

- **Decorator único:** Algunas implementaciones no utilizan una clase Decorator abstracta ya que solo hay una posible variante para el componente.
- **Decoradores redefinidos:** Se pueden redefinir nuevos Decoradores redefiniendo los anteriores, lo cual modificara algunas partes del comportamiento del componente.
- Si solamente hay una clase Componente concreto y ninguna interfaz Componente, entonces la clase Decorator es normalmente una subclase de la clase Componente concreto.

## Las consecuencias del patrón son:

1. Ofrece más flexibilidad que la herencia estática. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Por el contrario con la herencia hay que crear una nueva clase para cada nueva responsabilidad.
2. Se consiguen componentes pequeños muy parecidos.
3. Se reduce el número de clases y el árbol de herencia de clases. Teniendo menos clases se simplifica el diseño y la implementación de los programas.
4. Una dificultad asociada al patrón Decorator es que un objeto decorador no es un objeto componente, por lo que se pierde la identidad del objeto. Los objetos componente se esconden detrás de los objetos decorador.



## EJEMPLO:

- **Componente:** Cuenta
- **ComponenteConcreto:** CuentaGeneral
- **Decorator:** Decorador
- **DecoratorConcreto:** CuentaCorriente, CuentaPlazo

- Simplifica el acceso a un conjunto de subsistemas o un sistema complejo. Representa una única interfaz unificada, que envuelve el subsistema, y es responsable de colaborar con los componentes del subsistema.
- Se utiliza cuando:
  - Tenga que reducir la dependencia entre clases, Facade ofrece un punto de acceso al resto de clases, si estas cambian o se sustituyen por otras, sólo hay que actualizar la clase Facade sin que el cambio afecte a las aplicaciones cliente. Facade no oculta las clases si no que ofrece una forma más sencilla de acceder a ellas, en los casos en que se requiere se puede acceder se puede acceder directamente a ellas.
  - Se ofrece un acceso sencillo a subsistemas complejos proporcionando una interfaz más sencilla sin eliminar las opciones avanzadas.
  - Tenga que descomponer un sistema en capas, puede utilizar Facade para ofrecer una interfaz de acceso ente las mismas.
  - Reducir el acoplamiento entre los clientes y los subsistemas.

Facilita el acceso a un conjunto de subsistemas.

### VARIACIONES

- Se puede implementar la fachada como una interfaz o una clase abstracta. Con esto dejemos los detalles de implementación para un momento posterior y reducimos el acoplamiento.
- Varias fachadas pueden proporcionar distintas interfaces al mismo conjunto de subsistemas.

### Las consecuencias del patrón son:

1. Oculta a los clientes parte de la complejidad de los subsistemas reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar.
2. Disminuye el acoplamiento entre los subsistemas y el cliente. Un acoplamiento débil nos permite modificar los componentes del sistema sin que sus clientes se vean afectados. Se facilita la estructuración en capas.
3. No impide que los clientes usen las clases del subsistema directamente en caso de necesitarlo.

- Reducir el número de objetos detallados de muy bajo nivel en un sistema mediante la compartición de objetos.
  - **Peso ligero:** Objeto compartido que puede usarse a la vez en varios contextos.
  - **Estado intrínseco:** Se guarda con el propio objeto, consiste en información que es independiente de su contexto y que puede ser compartida.
  - **Estado extrínseco:** Depende del contexto y cambia con él, no puede ser compartido.
- Los pesos ligeros modelan conceptos o entidades que normalmente son demasiado numerosos como para ser representados como objetos.
- **EJEMPLO:** *Un editor de documentos tendrá un peso ligero para cada letra del alfabeto con su código de carácter (estado intrínseco), sin embargo las coordenadas de posición en el documento y su estilo se obtienen con los algoritmos de maquetación y formateado (estado extrínseco).*
- En el API de Java se utiliza el patrón para representar bordes (BorderFactory).
- Se utiliza cuando se cumplan todas estas condiciones:
  - Una aplicación utiliza muchos objetos con características similares.
  - Los costes de almacenamiento de tantos objetos son elevados.
  - Las características de los objetos se pueden dividir en similares (estado intrínseco) y diferentes (estado extrínseco), permitiendo compartir las comunes.
  - Los grupos de objetos similares pueden reemplazarse por un objeto compartido cuyas partes diferentes han sido eliminadas.
  - La aplicación no depende de la identidad de un objeto. Puesto que los objetos de peso ligero pueden ser compartidos.

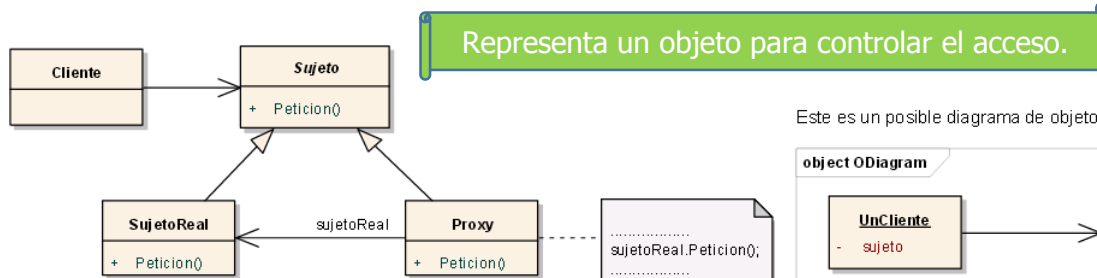
Reducir objetos mediante la compartición de estos.

## Las consecuencias del patrón son:

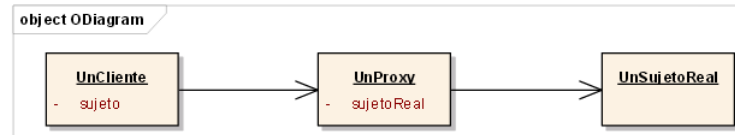
1. Se reduce el número de objetos que hay que gestionar.
2. Se pueden incrementar los costes en tiempo de ejecución si se realizan transferencias, búsquedas o cálculos de su estado extrínseco.
3. Cuanto mayor sea el número de objetos peso ligero compartidos, mayor ahorro de espacio tendremos.

# PATRÓN PROXY (Virtual Proxy o Surrogate)

- Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.
- Se utiliza cuando:
  - Proxy remoto:** Cuando el objeto está en un sistema remoto y necesite un representante local.
  - Proxy virtual:** Para retrasar la creación de objetos costosos hasta que sean necesarios.
  - Proxy de protección:** Para controlar los derechos de acceso a un objeto.
  - Proxy de sincronización:** Para gestionar accesos de múltiples clientes a un recurso.
  - Referencias inteligentes:** Para gestión y mantenimiento del acceso a un objeto real, permite contabilizar su utilización e incluso destruirlo cuando no se necesita. También permite sincronizar accesos concurrentes al mismo, bloqueándolo mientras está en uso.



Este es un posible diagrama de objetos de una estructura de proxies en tiempo de ejecución:



- Sujeto:** Define la interfaz común para el **Proxy** y el **SujetoReal** de forma que se pueda usar un **Proxy** donde se espere un **SujetoReal**.
- Proxy:** Ofrece una interfaz equivalente al de la clase **SujetoReal**, y redirige las llamadas de los métodos al objeto real. Puede realizar un pre-procesamiento y un post-procesamiento sobre los servicios ofrecidos por la clase real.
- SujetoReal:** Es la clase que implementa los servicios reales ofrecidos, puede ser una instancia local o remota.
- Cliente:** La aplicación, utiliza la interfaz de la clase proxy para hacer uso de la clase real.

## Las consecuencias del patrón son:

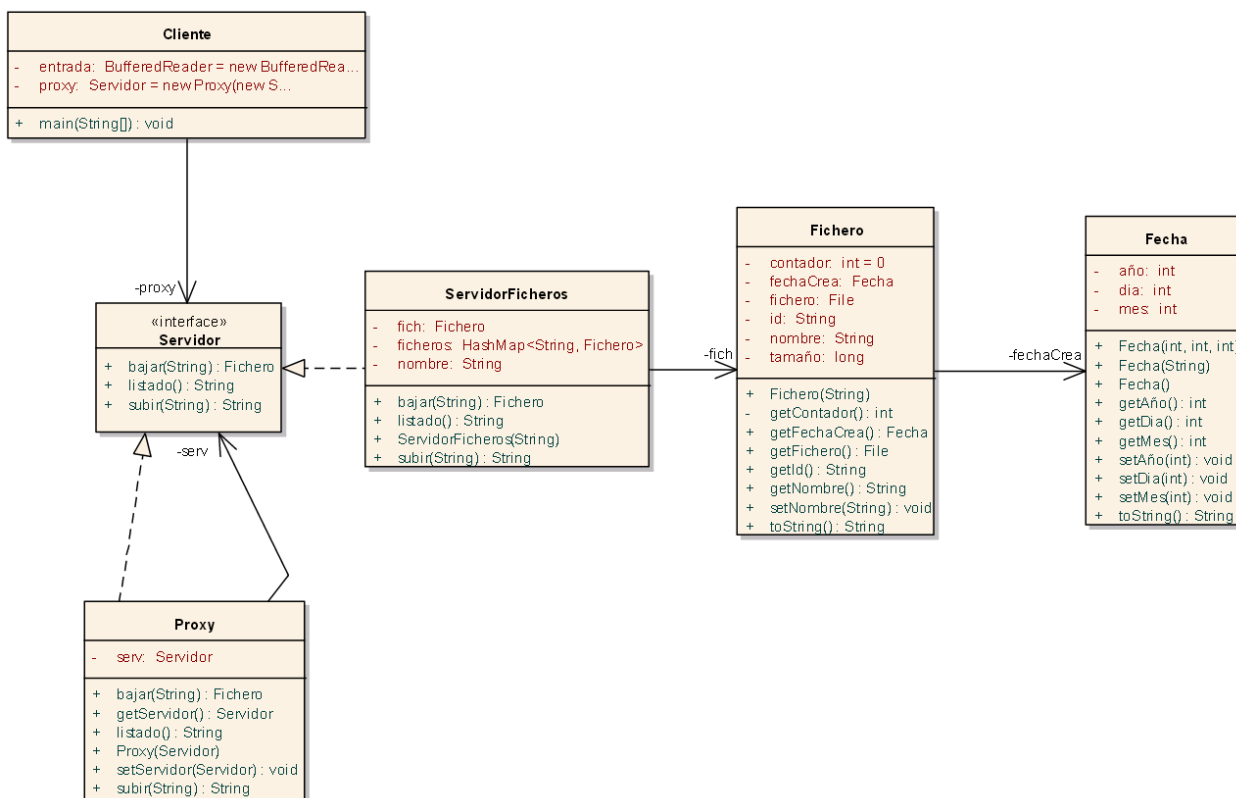
- El patrón **Proxy** introduce un nivel de indirección al acceder a un objeto.
- Puede mejorar la eficiencia al retrasar la instanciación de un objeto costoso hasta que sea necesario utilizarlo. Así el objeto proxy lo sustituye ofreciendo la misma interfaz, solo cuando es necesario le solicita al objeto real la información que necesita.
- Aumenta la seguridad.
- Los clientes se desentienden de la ubicación de los componentes accedidos.

## CONSEJOS DE IMPLEMENTACIÓN

- Se diseña la clase **Proxy** con igual interfaz que la clase real, si es posible se hereda de una clase abstracta común a la clase real (o se implementa una interfaz).
- Según el tipo de proxy (remoto, virtual, etcétera) se le añaden métodos de pre-procesamiento y post-procesamiento donde se implementan las funciones de control.
- Se enlaza el **Proxy** con la clase real mediante instancias locales, referencias o punteros, sockets.
- Se implementa el cliente haciendo uso de la clase **Proxy** en lugar de la real.

## EJEMPLO:

- Sujeto:** Servidor
- Proxy:** Proxy
- Sujeto real:** ServidorFicheros
- Cliente:** Cliente





# Patrones de comportamiento

# PATRÓN CHAIN OF RESPONSABILITY

- Desacoplar el emisor y el mensaje del objeto que presta el servicio. Para conseguir esto se establece una cadena en el sistema de forma que el emisor envía un mensaje al primer elemento de la cadena, este lo procesa si puede, y si no, lo redirige a otro objeto de la cadena hasta que uno pueda procesarlo.
- Se utiliza cuando:
  - Hay un grupo de objetos en un sistema que pueden responder al mismo tipo de mensajes y a priori no se sabe quien es.
  - Los mensajes deben ser manejados por uno entre varios objetos. No se especifica el receptor del mensaje.
  - Algunos mensajes deben ser manejados en el nivel donde se reciben mientras que otros deben ser redirigidos a algún otro objeto (la solicitud se especifica dinámicamente).

Cadena, un objeto recibe un mensaje, lo atiende si le corresponde y si no, se lo pasa a otro objeto.

## **VARIACIONES**

- **Estrategias de gestión:** Se centran en cómo se implementa el comportamiento del manejador:
  - Manejador por defecto: Utilizado cuando no hay una clase explícita a la que reenviar el mensaje (mensajes ignorados).
  - Manejar y ampliar: La propagación de los mensajes implica un comportamiento base y uno ampliado.
  - Manejadores dinámicos: Permiten que se cambie la estructura de reenvío de mensajes en tiempo de ejecución mediante un método que establece el siguiente elemento de la cadena.
- **Estrategias de reenvío:** Permiten gestionar o redirigir los mensajes producidos por un componente:
  - Manejar por defecto: Maneja cualquier mensaje que no sea reenviado de forma específica.
  - Propagar por defecto: Redirige cualquier mensaje que no sea explícitamente reenviado.
  - Reenviar a un manejador por defecto.
  - Ignorado por defecto: Se descarta cualquier mensaje que no sea explícitamente manejado o dirigido. Se reduce el número de mensajes en circulación.

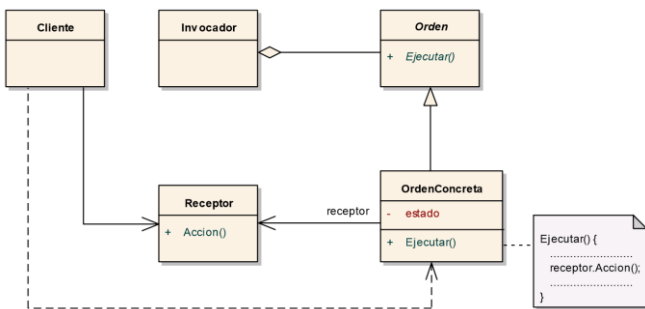
## **Las consecuencias del patrón son:**

1. El objetivo de este patrón, como el de muchos otros, es el de desacoplar objetos, reduciendo el acoplamiento. Esto se consigue con Cadena de Responsabilidad ya que un objeto solo necesita saber cómo debe procesar un evento o mensaje.
2. Este patrón ofrece una gran flexibilidad ante el procesamiento de eventos, ya que delega las responsabilidades en elementos más simples. Todo esto permite que un conjunto de clases se comporte como una unidad.
3. Debido a que no hay un receptor explícito no se garantiza la recepción del mensaje. Se puede alcanzar el final de la cadena sin que se haya procesado la petición.
4. **Desventaja:** A medida que el redireccionamiento de la cadena se hace más complejo, se hace más difícil de comprobar y depurar, ya que hay que controlar si los mensajes se están enviando correctamente.

# PATRÓN COMMAND

- Encapsular un comando en un objeto. Este objeto contiene el comportamiento y los datos necesarios para una acción específica. Permite parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones. Además permite deshacer operaciones.
- Se utiliza cuando:
  - Parametrizar objetos con una acción a realizar.
  - Se quiera desacoplar la fuente de una petición del objeto que la cumple.
  - Haya que implementar un mecanismo de rehacer/deshacer acciones.
  - Haya que poner en cola y ejecutar comandos en momentos distintos.
  - Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída del sistema.
  - Estructurar un sistema mediante operaciones de alto nivel basadas en operaciones más sencillas (primitivas).

Permite parametrizar a los clientes con diferentes peticiones.



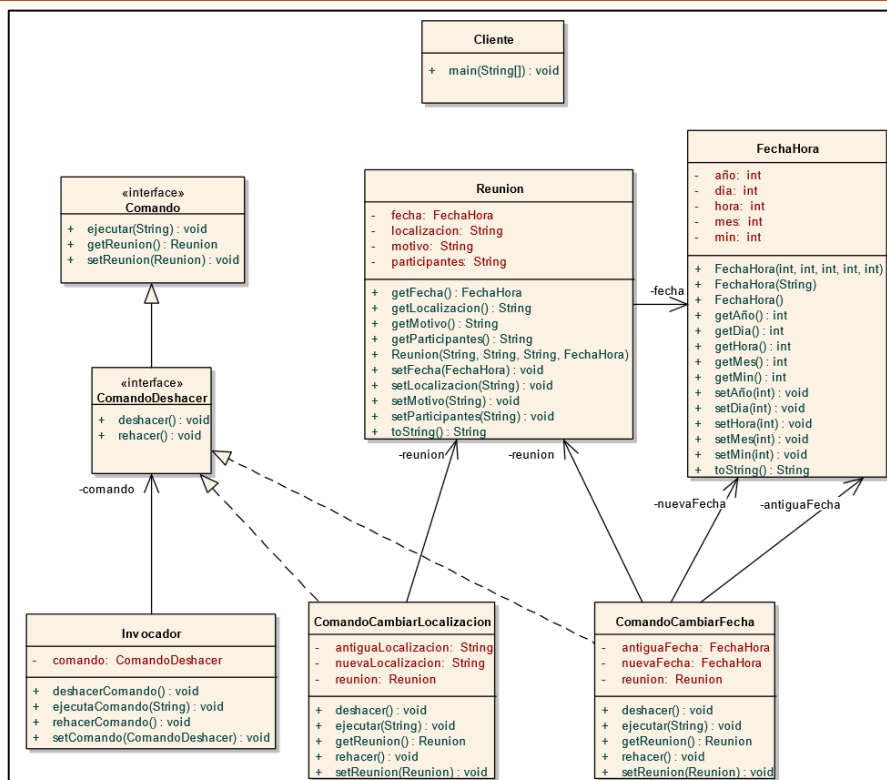
- Orden (comando):** Interfaz en la que se definen los métodos que se usarán por el invocador. Además estos métodos serán implementados por cada uno de los comandos concretos. En el diagrama solo se muestra el método ejecutar, pero se pueden añadir más métodos como deshacer y rehacer. *ComandoDeshacer*
- OrdenConcreta (comando concreto):** Implementación de la interfaz Comando. Mantiene una referencia con el receptor. Implementa el método ejecutar para que realice la acción propia de cada comando concreto. *ComandoCambiarLocalización*
- Invocador:** El que llama al método ejecutar del objeto Comando. *Invocador*
- Receptor:** Para quien Comando cumple la petición solicitada. *Reunión*
- Cliente:** Crea un objeto OrdenConcreta y establece su receptor. *Cliente*

## VARIACIONES

- Deshacer:** El patrón se puede extender para proporcionar la capacidad de deshacer los comandos. Simplemente se amplía la interfaz **Comando (Orden)** y se añade el método deshacer invirtiendo el último comando. Si se desea deshacer un conjunto de comandos y no solo el último debemos guardar un histórico de comandos.
- MacroCommand:** Es una colección de comandos. Se suele componer mediante el patrón **Composite**. Un objeto **MacroCommand** contiene una lista de subcomandos. Cuando se llama al método ejecutar se redirige a los subcomandos.

### Las consecuencias del patrón son:

- Se desacopla la parte de la aplicación que invoca a los comandos de la implementación de los mismos.
- Al tratarse los comandos como objetos, se puede realizar herencia de los mismos, composiciones de comandos (mediante el patrón **Composite**).
- Permite reemplazar objetos comando en tiempo de ejecución y hace que los comandos sean objetos normales, teniendo todas las propiedad usuales de los objetos.
- Facilita la introducción de nuevos comandos. Simplemente escribiendo otra implementación de la interfaz e introduciéndola en la aplicación.



## PATRÓN INTERPRETER (Intérprete)

- Definir un intérprete para un lenguaje sencillo.
- Se utiliza cuando:
  - Haya que interpretar un lenguaje sencillo. Para gramáticas complejas, la jerarquía de clases para la gramática se convierte en algo grande e inmanejable.
  - Podemos representar las sentencias del lenguaje como árboles abstractos de sintaxis (AST) – Abstract Syntax Trees.
  - La resolución del problema a evaluar puede ser expresada en términos de ese lenguaje.
  - La eficiencia no sea un aspecto fundamental.

Interpretar lenguaje sencillo.

### Las consecuencias del patrón son:

1. La gran ventaja del patrón **Intérprete** es que la gramática puede ser ampliada o cambiada muy fácilmente. Para añadir una regla, solo hay que crear otra clase que implemente la interfaz **ExpresionAbstracta**. Esta clase implementará una nueva regla en el método interpretar.
2. El patrón **Intérprete**, en cambio, no es adecuado cuando la gramática es grande ya que puede generar problemas para depurar y mantener el sistema.

## PATRÓN TEMPLATE METHOD

- Proporcionar un método que permita que las subclases redefinan partes del método sin reescribirlo.
- Se utiliza cuando:
  - Haya que proporcionar un esqueleto para un método, permitiendo que las subclases redefinan partes específicas del método.
  - Centralizar partes de un método que se definen en todos los subtipos de una clase, pero que siempre tienen una pequeña diferencia en cada subclase.
  - Queramos evitar la duplicación de código entre clases de una jerarquía.
  - Controlar las operaciones que es necesario redefinir en las subclases.

Varias versiones de los métodos sin redefinirlos..

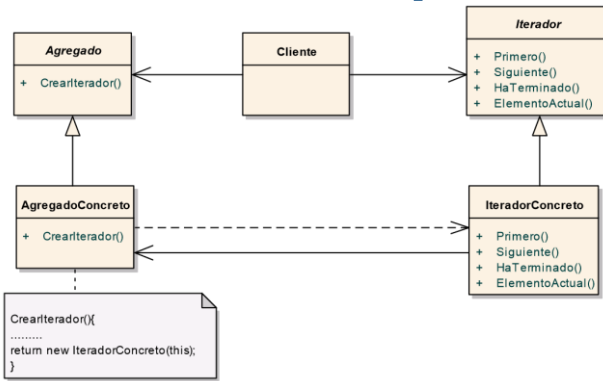
### Las consecuencias del patrón son:

1. Reutilización de código, ya que, sin el método plantilla el código se duplica en muchas subclases.
2. Desventaja: Podría ser que el método plantilla llame a demasiados métodos abstractos. Es mejor hacer que el método plantilla llame a un número limitado de métodos abstractos.

# PATRÓN ITERATOR

- Proporcionar una forma coherente de acceder secuencialmente a los datos de una colección, independientemente del tipo de colección.
- Se utiliza cuando:
  - Queramos proporcionar una forma uniforme, coherente e independiente de la implementación, de desplazarse por los elementos de una colección.
  - Queramos permitir el recorrido de múltiples colecciones, permitiendo que distintos clientes naveguen simultáneamente por la misma colección.

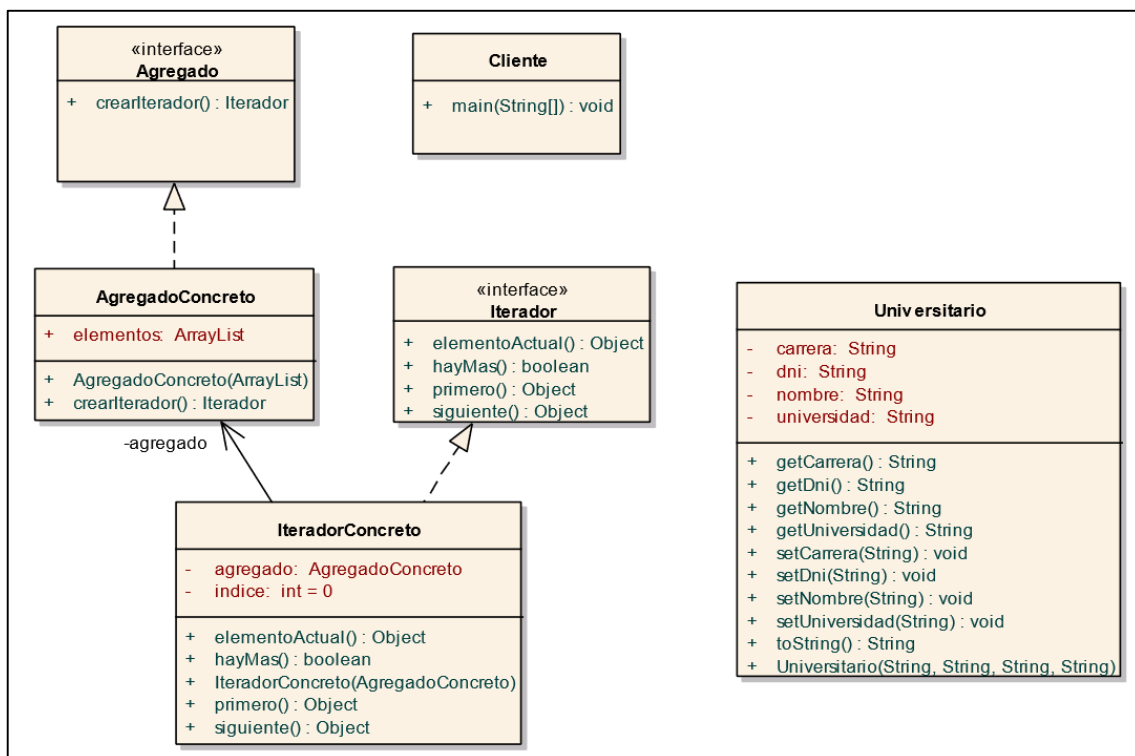
Recorrer una colección de datos indeterminados.



- Iterador**: Interfaz en la que se definen los métodos de iteración. Como mínimo, se definen los métodos `primero()`, `siguiente()`, `hayMas()` y recuperación del elemento actual. **Iterador**
- IteradorConcreto**: Clases "iteradoras" que implementan la interfaz **Iterador**. Sus instancias se crean por medio de **AgregadoConcreto**. Debido al alto acoplamiento con esta clase, a menudo **IteradorConcreto** suele ser una clase interna de **AgregadoConcreto**. Mantiene la posición actual en el recorrido del agregado. **IteradorConcreto**
- Agregado**: Interfaz que define el método de fabricación para crear un iterador. **Agregado**
- AgregadoConcreto**: Implementa la interfaz **AgregadoConcreto** y crea los iteradores concretos según se necesiten. **AgregadoConcreto**
- Cliente**: Hace uso del iterador implementado. **Universitario**

## Las consecuencias del patrón son:

- Podemos recorrer una estructura de objetos sin conocer los detalles internos de sus clases.
- Al definir una interfaz uniforme para los iteradores concretos, se simplifica el recorrido de las colecciones y se permite utilizar polimorfismo.
- Al tener que recorrer agregados complejos podemos hacerlo de muchas formas distintas. Facilitan cambiar el algoritmo de recorrido.
- Los iteradores permiten a los clientes manejar múltiples puntos de navegación para la misma colección, ya que un iterador es como un puntero de una colección. Simplemente instanciando nuevos iteradores, podemos conseguir puntos de navegación en una misma colección.



## Iteradores en Java

- Están en la biblioteca `java.util.iterator`.
- Métodos `hasNext()` y `next()`.

## PATRÓN MEDIATOR

- Proporcionar la posibilidad de que un único objeto se encargue de gestionar la comunicación entre distintos objetos, sin que estos necesiten conocerse entre sí. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente.
- Se utiliza cuando:
  - Las reglas de comunicación sean complejas entre los objetos de un sistema.
  - Queramos adaptar un comportamiento distribuido entre clases sin necesidad de especializarlas.
  - Se desee que los objetos sigan siendo simples y manejables.
  - Las clases son difíciles de utilizar porque tienen muchas relaciones de interdependencia con otras.

Gestiona la comunicación entre objetos, promoviendo el bajo acoplamiento

### Las consecuencias del patrón son:

1. Los componentes individuales de la aplicación se hacen más simples y es mas fácil tratar con ellos porque no necesitan pasarse mensajes entre sí.
2. Simplifica la estrategia de comunicación ya que es responsabilidad exclusiva del mediador.
3. Reduce la herencia. Un mediador localiza el comportamiento que de otra manera estaría distribuido en varios objetos. Para cambiar el comportamiento solo tenemos que crear una subclase de Mediador.
4. Desacopla a los colegas.
5. **Desventaja:** el código del mediador puede ser difícil de mantener conforme aumenta el número de clases y la complejidad de estas.

## PATRÓN MEMENTO

- Guardar el estado interno de un objeto para que luego se pueda restaurar, sin revelar su contenido al resto del mundo.
- Se utiliza cuando:
  - Haya que tomar una instantánea del estado de un objeto.
  - Esa instantánea se utilice para restaurar el estado original.
  - Una interfaz directa que obtuviera el estado interno del objeto violaría el principio de encapsulación, porque revelaría el funcionamiento interno.

Instantánea de un objeto ALTAMENTE encapsulado

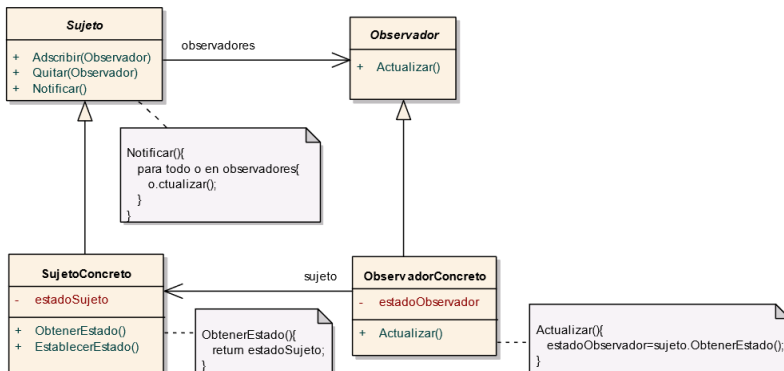
### Las consecuencias del patrón son:

1. Preserva la encapsulación. No es posible acceder a la información que hay en su interior. Además el cliente ya no necesita saber nada sobre el funcionamiento del originador, como obtener un recuerdo y como utilizarlo.
2. El originador es mucho más simple ya que solo necesita crear y utilizar recuerdos en vez de registrar múltiples estados.
3. **Desventaja:** Puede resultar muy costoso si se realizan muchas copias y las copias son voluminosas. Además, el objeto (conserje) que se encarga de gestionar los mementos no sabe cuanto ocupan y su almacenamiento puede provocar un coste inesperado.

# PATRÓN OBSERVER (publisher-suscriptor / dependents)

- Permite definir dependencias 1-a-muchos de forma que los cambios en un objetos e comuniquen a los objetos que dependen de él.
- Se utiliza cuando:
  - Existe al menos un emisor de mensajes.
  - Uno o más receptores de mensajes podrían variar dentro de una aplicación o entre aplicaciones.
  - Si se produce un cambio en un objeto, se requiere el cambio de otros y no se sabe cuántos se necesitan cambiar.
  - No queremos que estén fuertemente acoplados.

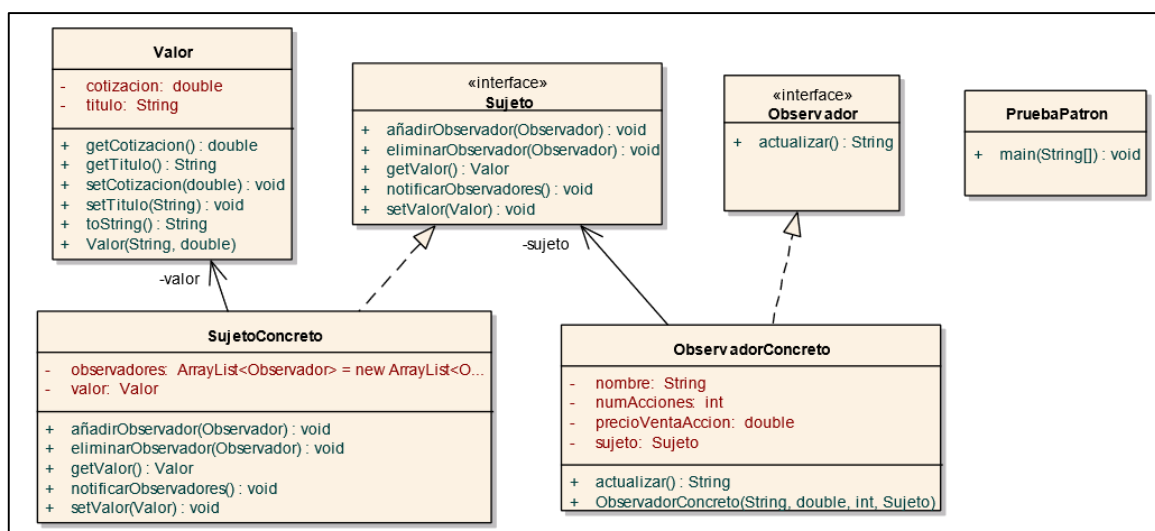
Los objetos suscritos reciben lo que el padre (emisor, publicador) publica.



- **Sujeto observado:** Interfaz que define como pueden interactuar los observadores con el sujeto. Define métodos para añadir y quitar observadores y avisarles de que se han producido cambios en el sujeto. *Sujeto*
- **SujetoConcreto:** Implementa la interfaz *Sujeto*. Contiene una lista de observadores a los que avisa cuando cambia su estado. *SujetoConcreto*
- **Observador:** Interfaz para actualizar los objetos ante cambios en un sujeto. *Observador*
- **ObservadorConcreto:** Mantiene una referencia a un objeto *SujetoConcreto*. Implementa la interfaz *Observador* y define los métodos para responder a los mensajes recibidos del sujeto. *ObservadorConcreto*
- **Cliente:** Clase principal. *PruebaPatron*

## Las consecuencias del patrón son:

1. Desacoplamiento entre sujetos y observadores, convirtiéndolos en entidades reutilizables por separado.
2. Es un medio muy flexible de distribuir la información desde un objeto a muchos, de forma dinámica en tiempo de ejecución y sin que las clases implicadas sean conscientes del esto.
3. El sujeto puede incluir cierta información en el mensaje de actualización, de forma que cada observador pueda decidir si el cambio de estado le afecta o no.
4. Un sujeto puede ser a su vez observador respecto de otros.
5. **Desventaja:** Un pequeño cambio en el sujeto puede provocar mucho procesamiento en los observadores.



## Observadores en Java

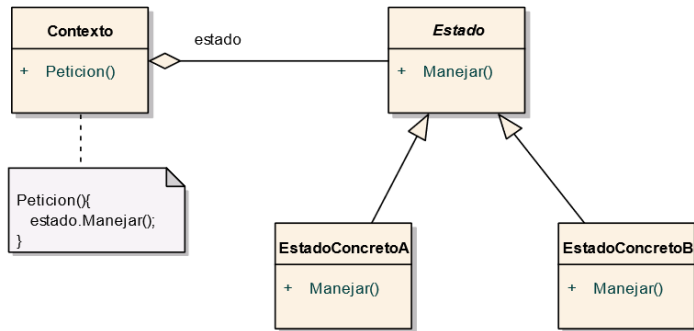
En el API de Java, existe la interfaz **Observer** y la clase **Observable**. Esto permite que muchos objetos reciban eventos de otro objetos, en lugar de los sistemas de eventos básicos que solo permiten notificar a un único objeto.



# PATRÓN STATE

- Permitir que un objeto se comporte de distinta forma dependiendo de su estado interno, como si cambiase la clase a la que pertenece. Permite cambiar fácilmente el comportamiento de un objeto en tiempo de ejecución.
- Se utiliza cuando:
  - El comportamiento de un objeto dependa de su estado y queramos cambiar de comportamiento en tiempo de ejecución.
  - Distintas operaciones de una clase tengan una estructura de control condicional compleja que dependa del estado, permite convertir cada rama en una clase y tratar el estado como un objeto a parte.

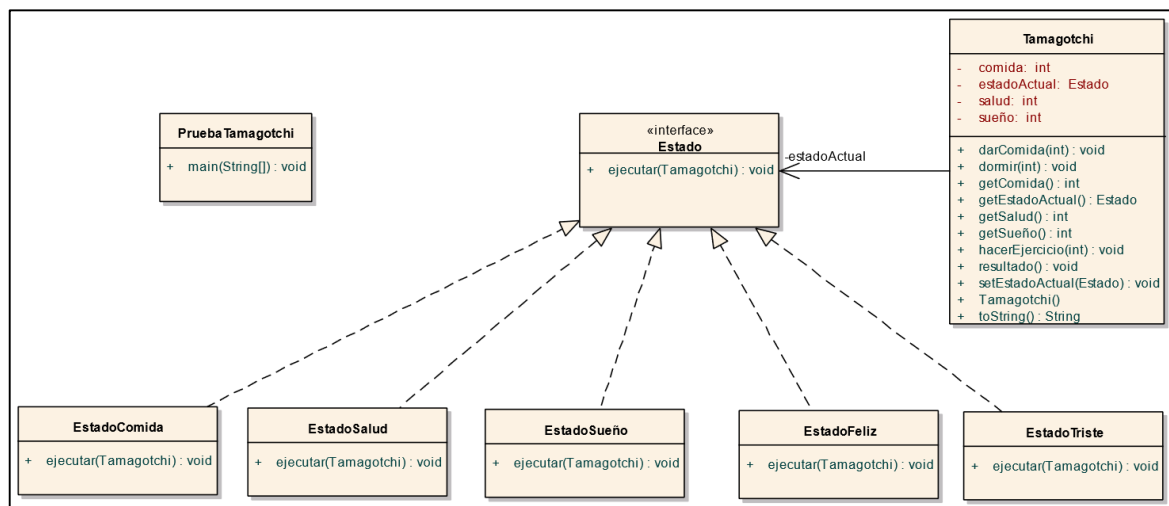
Permite crear clases que modelan comportamiento en vez de datos.



- Contexto:** Clase que mantiene una referencia de estado actual y es la interfaz que utilizan los clientes. Delega todas las llamadas a los métodos específicos del estado en el objeto *Estado* actual. *Tamagotchi*
- Estado:** Interfaz que define los métodos que dependen del estado del objeto. *Estado*
- EstadoConcreto:** Implementa la interfaz *Estado*. Implementa el comportamiento específico de un estado. *EstadoComida, EstadoSalud, EstadoSueño, EstadoFeliz, EstadoTriste*

## Las consecuencias del patrón son:

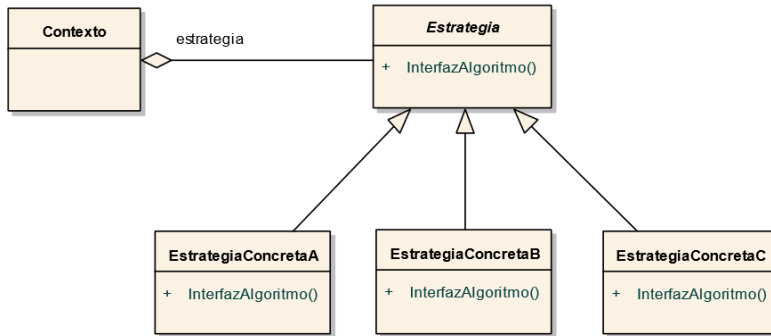
- Flexibiliza la creación de nuevos estados y transiciones por lo que se obtiene una visión mucho más clara del comportamiento. Para crear un nuevo estado, únicamente hay que crear una nueva clase que implemente la interfaz.
- El flujo de control del código que depende del estado se hace patente en la jerarquía de clases.
- Los estados están ahora encapsulados en objetos por lo que se facilita el reconocimiento de un cambio entre estados.
- Desventaja:** Gran número de clases asociadas que suele tener este patrón, pero cuando se compara con las largas cadenas de switch/case, que se generarían si no se utiliza *Estado*, puede ser considerado como una ventaja.



# PATRÓN STRATEGY (POLICY)

- Definir un grupo de clases que representan un conjunto de posibles comportamientos. Estos comportamientos pueden ser fácilmente intercambiados en un aplicación, modificando la funcionalidad en cualquier instante.
- Se utiliza cuando:
  - Muchas clases relacionadas sólo se diferencian en su comportamiento.
  - Se necesitan distintas variables de un algoritmo.
  - Una clase tiene distintos comportamientos posibles que aparecen como instrucciones condicionales en sus métodos.
  - No se sepa qué aproximación utilizar hasta el momento de ejecución.

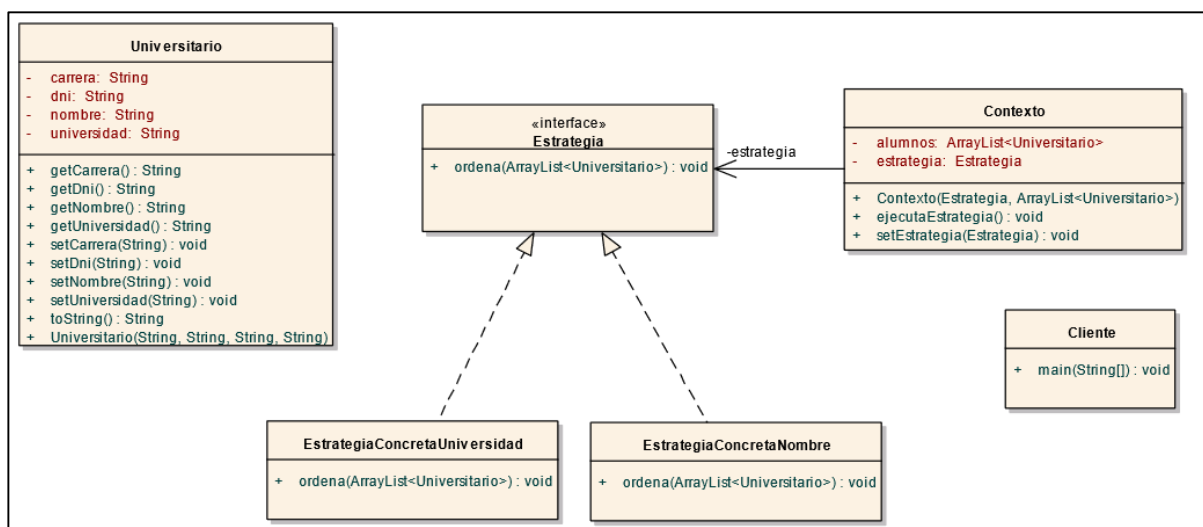
Permite crear clases que modelan comportamiento en vez de datos.



- Contexto:** Clase que utiliza las diferentes estrategias para las distintas tareas. Mantiene una referencia a la instancia *Estrategia* que usa y tiene un método para reemplazar la actual instancia de *Estrategia*. *Contexto*
- Estrategia:** Interfaz en la que se definen todos los métodos disponibles para ser manejados por contexto. *Estrategia*
- EstrategiaConcreta:** Clase que implementa la interfaz *Estrategia* utilizando un conjunto específico de reglas para cada uno de los métodos de la interfaz. *EstrategiaConcretaUniversidad*, *EstrategiaConcretaNombre*

## Las consecuencias del patrón son:

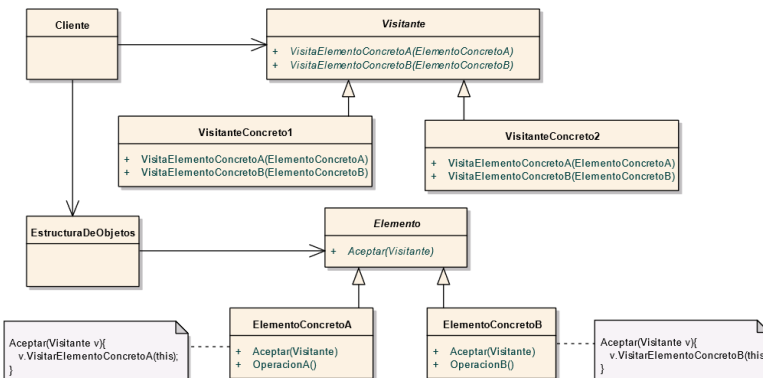
- Cada comportamiento se define en su propia clase, por lo que se consigue que sean más fáciles de mantener.
- Resulta sencillo ampliar el modelo para incorporar nuevos comportamientos sin el alto coste de modificar todo el código de la aplicación o la utilización de la herencia.
- Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento.
- El principal problema es que cada estrategia debe tener la misma interfaz. Debe identificar una interfaz suficientemente genérica como para ser aplicada a distintas implementaciones, pero que, al mismo tiempo, sea suficientemente específica para ser utilizada por las distintas estrategias concretas.



# PATRÓN VISITOR

- Proporcionar una forma fácil y sostenible de ejecutar acciones en una familia de clases. Este patrón centraliza los comportamientos y permite que sean modificados o ampliados sin cambiar las clases sobre las que actúan.
- Se utiliza cuando:
  - Un sistema contenga un grupo de clases relacionadas.
  - Haya que realizar muchas operaciones distintas y no relacionadas sobre algunos o todos los objetos de una estructura de objetos y no queremos contaminar sus clases con dichas operaciones.
  - Las operaciones deban ejecutarse de forma diferente en cada una de las distintas clases.

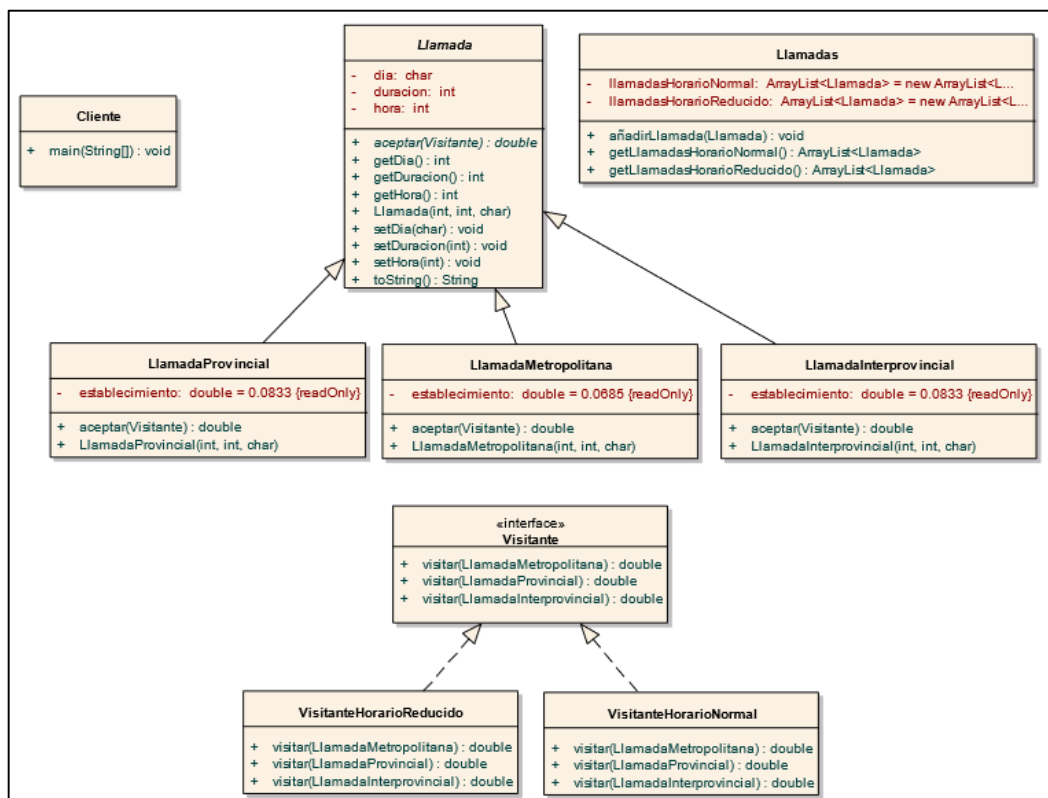
Visitamos una clase y según las circunstancias ciertas acciones se ejecutan o no.



- Visitante:** Interfaz o clase abstracta que define el termino visitar para cada una de las clases *Elemento* concretas. *Visitante*
- VisitanteConcreto:** Clase que representa una operación específica del sistema. Implementa la interfaz *Visitante* para una operación o algoritmo específico. *VisitanteHorarioReducido*, *VisitanteHorarioNormal*
- Elemento:** Clase abstracta o interfaz que representa los objetos sobre los que actúa *Visitante*. Define el método aceptar que recibe un visitante como argumento. *Llamada*
- ElementoConcreto:** Implementa la interfaz *Elemento*. Implementa el método *acceptar* invocando el método *visitar* apropiado definido en *Visitante*. *LlamadaProvincial*, *LlamadaMetropolitana* y *LlamadaInterprovincial*.
- EstructuraDeObjetos:** Para poder enumerar todos los elementos. *Llamadas*

## Las consecuencias del patrón son:

- Facilita en gran medida la introducción de un nuevo comportamiento de un sistema. Para añadir nuevas funciones basta con crear simplemente una nueva clase que implemente la interfaz *Visitante* y escribir el nuevo código para realizar la función.
- Visitante* es útil porque permite centralizar código funcional para una operación. Hace que el código sea más fácil de ampliar y modificar y el mantenimiento es más sencillo.
- Un visitante agrupa operaciones relacionadas y se para el comportamiento no relacionado en las subclases del visitante.
- Desventaja:** Ofrece muy poca flexibilidad en las clases *Elemento*. Cualquier nueva clase *Elemento* hace necesario definir un nuevo método en la interfaz *Visitante* y en cada visitante concreto su implementación.



# Antipatrones y patrones

## GRASP

# ANTIPATRONES

- Los antipatrones son descripciones de situaciones o soluciones, recurrentes que producen consecuencias negativas. Un antipatrón puede ser el resultado de una decisión equivocada sobre como resolver un determinado problema, o bien, la aplicación correcta de un patrón de diseño en el contexto equivocado.
- En el desarrollo de software un antipatrón es una mala práctica, que aunque puede solucionar un problema en nuestra aplicación, también puede generar problemas de mantenimiento, diseño o comportamiento en el software.
- Los antipatrones son una iniciativa de investigación sobre el desarrollo de software que se focaliza en soluciones con efectos negativos, contrario a los patrones de diseño.

## Relación con los patrones

Los antipatrones proveen un vocabulario común que mejora la comunicación en el equipo de desarrollo para poder identificar problemas y discutir soluciones.

Tanto los patrones como los antipatrones documentan conocimiento con el fin de ser distribuido para su utilización. Mientras que los patrones de diseño documentan soluciones exitosas, los antipatrones documentan situaciones problemáticas.

Clasificación de los antipatrones:

- Desarrollo de software:** Se centran en problemas asociados al desarrollo de software a nivel de aplicación.
- Arquitectura de software:** Se centran en la lectura de las aplicaciones y componenets a nivel de sistema y empresa.
- Gestión de proyectos de software:** En la Ingeniería del Software, más de la mitad del trabajo, consiste en comunicación entre personas y resolver problemas relacionados con estas. Los antipatrones de gestión de proyectos software identifican algunos de los escenarios clave donde estos temas son destructivos para el proceso de desarrollo software.

Causas principales que causan los antipatrones son:

- Prisa.
- Apatía.
- Estrechez de miras.
- Pereza.
- Avaricia.
- Ignorancia.
- Soberbia.

- Una refactorización es una transformación controlada del código fuente de un sistema que no altera su comportamiento observable, cuyo fin es hacerlo más comprensible y de más fácil mantenimiento.
- Este proceso permite tomar diseños defectuosos, con código mal escrito y adaptarlo a uno bueno, mejor organizado. El diseño se da a lo largo de todo el ciclo de desarrollo.
- Para poder refactorizar de forma satisfactoria, es indispensable contar con un conjunto suficiente de casos de prueba que validen el correcto funcionamiento del sistema.

| TIPOS DE ANTIPATRONES | DESARROLLO                       | The blob                     |
|-----------------------|----------------------------------|------------------------------|
|                       |                                  | Lava Flow                    |
|                       |                                  | Functional decomposition     |
|                       |                                  | Poltergeists                 |
|                       |                                  | Golden Hammer                |
|                       |                                  | Spaguetti code               |
|                       |                                  | Cut and Paste programming    |
|                       | MINIANTIPATRONES DE DESARROLLO   | Continuous obsolescence      |
|                       |                                  | Ambiguous viewpoint          |
|                       |                                  | Boat anchor                  |
|                       |                                  | Dead end                     |
|                       |                                  | Input kludge                 |
|                       |                                  | Walking through a mine field |
|                       |                                  | Mushroom management          |
|                       | ARQUITECTURA                     | Stovepipe Enterprise         |
|                       |                                  | Vendor Lock-In               |
|                       |                                  | Architecture by implication  |
|                       |                                  | Desing by Committee          |
|                       |                                  | Reinvent the Wheel           |
|                       | MINIANTIPATRONES DE ARQUITECTURA | Autogenerated stovepipe      |
|                       |                                  | Jumble                       |
|                       |                                  | Cover your assets            |
|                       |                                  | Wolf ticket                  |
|                       |                                  | Warm bodies                  |
|                       |                                  | Swiss army knife             |
|                       |                                  | The grand old duke of York   |
|                       | GESTIÓN                          | Analysis paralysis           |
|                       |                                  | Death by planning            |
|                       |                                  | Corncob                      |
|                       |                                  | Irrational management        |
|                       |                                  | Project Missmanagement       |
|                       | MINIANTIPATRONES DE GESTIÓN      | Blowhard                     |
|                       |                                  | Jamboree                     |
|                       |                                  | Viewgraph engineering        |
|                       |                                  | Fear of success              |
|                       |                                  | Intellectual violence        |
|                       |                                  | Smoke and mirrors            |
|                       |                                  | Throw it over the wall       |
|                       |                                  | Fire drill                   |
|                       |                                  | The feud                     |
|                       |                                  | Email is dangerous           |

# PATRONES GRASP

## (General Responsibility Assignment Software Pattern)

- GRASP es el acrónimo de General Responsibility Assignment Software Patterns, y es una de las familias de patrones que propuso Craig Larman, el cual se centró principalmente en proponer los patrones como una codificación de principios básicos ampliamente utilizados para los expertos en objetos, para que parezcan elementales y familiares.
- Por ese motivo los patrones GRASP tienen nombres concisos como Experto en Información, Creador, Variaciones protegidas... para facilitar la comunicación.
- Los patrones GRASP, también conocidos como Patrones de Principios Generales para Asignar Responsabilidades, se basan principalmente en que la asignación de responsabilidades es extremadamente importante en el diseño orientado a objetos, ya que es en esta fase de diseño donde nos encontramos con la tarea de crear las clases y las relaciones entre ellas.
- La decisión a cerca de la asignación de responsabilidades tiene lugar, casi siempre, durante la creación de los diagramas de interacción y durante la programación. Estos patrones GRASP codifican buenos consejos y principios relacionados con frecuencia con la asignación de responsabilidades.
- Resumiendo, podemos decir que los patrones GRASP describen principios fundamentales del diseño de objetos y la asignación de responsabilidades, expresados como patrones.

## CONCEPTOS BÁSICOS Y CLASIFICACIÓN DE TIPOS

- Según Christopher Alexander describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma.
- Beneficios a causa de la reutilización:
  - **Reducción de tiempos.**
  - **Disminución del esfuerzo de desarrollo y mantenimiento.**
  - **Eficiencia.**
  - **Consistencia.**
  - **Fiabilidad.**
- Los Patrones son una forma documentada para resolver problemas de ingeniería del software el objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencias sobre sus problemas y sus soluciones.
- *Un patrón es una información que captura la estructura esencial y la perspicacia de una familia de soluciones probadas con éxito para un problema repetitivo que surge en un cierto contexto y sistema.*
- *Un patrón es una unidad de información nombrada, instructiva e intuitiva que captura la esencia de una familia exitosa de soluciones probadas a un problema recurrente dentro de un cierto contexto.*
- *Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un conjunto de fuerzas que ocurren repetidamente en ese contexto y una cierta configuración software que permite a estas fuerzas resolverse por sí mismas.*
- Características de patrones software:
  - Solucionar un problema.
  - Ser un concepto probado.
  - La solución no es obvia.
  - Describe participantes y relaciones entre ellos.
- Ventajas de la utilización de patrones:
  1. Facilitan la comunicación interna.
  2. Ahorran tiempo y experimentos.
  3. Mejoran la calidad del diseño y la implementación.
  4. Son como "normas de productividad".
  5. En Java facilitan su aprendizaje y comprender como esta diseñado el propio lenguaje.



# PATRONES DE DISEÑO

- El objetivo de los patrones de diseño es guardar la experiencia en diseños de programas orientados a objetos (catálogos de patrones).
- Podemos encontrar patrones de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos. Estos patrones solucionan problemas específicos del diseño y hacen los diseños orientados a objetos más flexibles, elegantes y reutilizables.
- Un diseñador que conoce algunos patrones puede aplicarlos inmediatamente a problemas de diseño sin tener que descubrirlos.
- Los patrones de diseño ayudan a un diseñador a conseguir un diseño correcto rápidamente.

## ELEMENTOS DE UN PATRÓN(ii)

- La **solución** describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones.  
La solución no describe un diseño particular o implementación, proveen una descripción abstracta de un problema de diseño y una disposición general de los elementos (clases y objetos en nuestro caso) que lo soluciona.
- Las **consecuencias** son los resultados de aplicar el patrón. Estas son muy importantes para la evaluación de diseños alternativos y para comprender los costes y beneficios de la aplicación del patrón.

## CLASIFICACIÓN DE LOS PATRONES

- Patrones de creación:**
  - Los patrones de creación muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Éstas se suelen resolver dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades.
  - A menudo hay varios patrones de creación que se pueden aplicar en una misma situación. Otras veces se pueden combinar varios de ellos. En otros casos se debe elegir solo uno de ellos.
- Patrones estructurales:**
  - Los patrones de esta categoría describen las formas comunes en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros.
- Patrones de comportamiento:**
  - Los patrones de este tipo son utilizados para organizar, manejar y combinar comportamientos. Nos permiten definir la comunicación entre los objetos de nuestro sistema y el flujo de la información entre los mismos.

# ELEMENTOS DE UN PATRÓN

- El **nombre del patrón** se utiliza para describir un problema de diseño, su solución, y consecuencias de forma resumida. Nombrar un patrón incrementa inmediatamente nuestro vocabulario de diseño.
- El **problema** describe cundo aplicar el patrón. Se explica el problema y su contexto. Podría describir problemas de diseño específicos tales como algoritmos o como objetos. Algunas veces el problema incluirá una lista de condiciones que deben cumplirse para poder aplicar el patrón.

## ELEMENTOS DE UN PATRÓN(iii)

- Los patrones de diseño identifican las clases y objetos participantes, sus papeles, colaboraciones y comunicaciones y la distribución de responsabilidades.
- Un patrón de diseño nombra, abstrae e identifica los aspectos clave de un diseño común, que lo hace útil para la creación de diseños orientados a objetos reutilizables.
- Los patrones de diseño se pueden utilizar en cualquier lenguaje de programación orientado a objetos, adaptando los diseños generales a las características de la implementación particular.

## CLASIFICACIÓN DE LOS PATRONES(ii)

| CREACIÓN             | ESTRUCTURALES | COMPORTAMIENTO              |
|----------------------|---------------|-----------------------------|
| Abstract Factory (O) | Adapter (C)   | Chain of responsibility (O) |
| Builder (O)          | Bridge (O)    | Command (O)                 |
| Factory Method (C)   | Composite (O) | Interpreter (C)             |
| Prototype (O)        | Decorator (O) | Iterator (O)                |
| Singleton (O)        | Facade (O)    | Mediator (O)                |
|                      | Flyweight (O) | Memento (O)                 |
|                      | Proxy (O)     | Observer (O)                |
|                      |               | State (O)                   |
|                      |               | Strategy (O)                |
|                      |               | Template Method (C)         |
|                      |               | Visitor (O)                 |

**Ámbito: (C) Clase, (O) Objeto.**

**Clase: Relaciones entre clases y subclases. Herencia. Estáticas**

**Objeto: Relaciones entre objetos. Dinámicas.**

**Trabajo realizado por**  
**Daniel Francisco López**  
**Enrique García Cortés**  
**Para la asignatura de Patrones del Software**  
**CURSO 2015/2016**



Universidad de Alcalá