Patrones fundamentales

IMPORTANTE
El término abstracto quiere decir que algo o esta definido, en este caso, nos referimos al método de un clase.

Clase abstracta:
Por ello, una clase abstracta es una clase en la cual se instancian uno o más métodos que no están definido en ningún momento y que luego, cuando se EXTIENDA esa clase en otra clase puedan ser definidos de la forma que se quiera para ese caso en concreto.
Estos métodos son parecidos a los métodos de las clases polimórficas.
En la clase abstracta pueden haber métodos que si estén definidos y pueden haber métodos públicos o no.
Una clase puede solo EXTENDER una clase abstracta.

Interface:
Una interface es un "clase" la cual ninguno de sus métodos están instanciados únicamente, no definidos, siendo TODOS esto métodos públicos para las clases que los IMPLEMENTAN.
Una clase puede implementar varias interfaces.

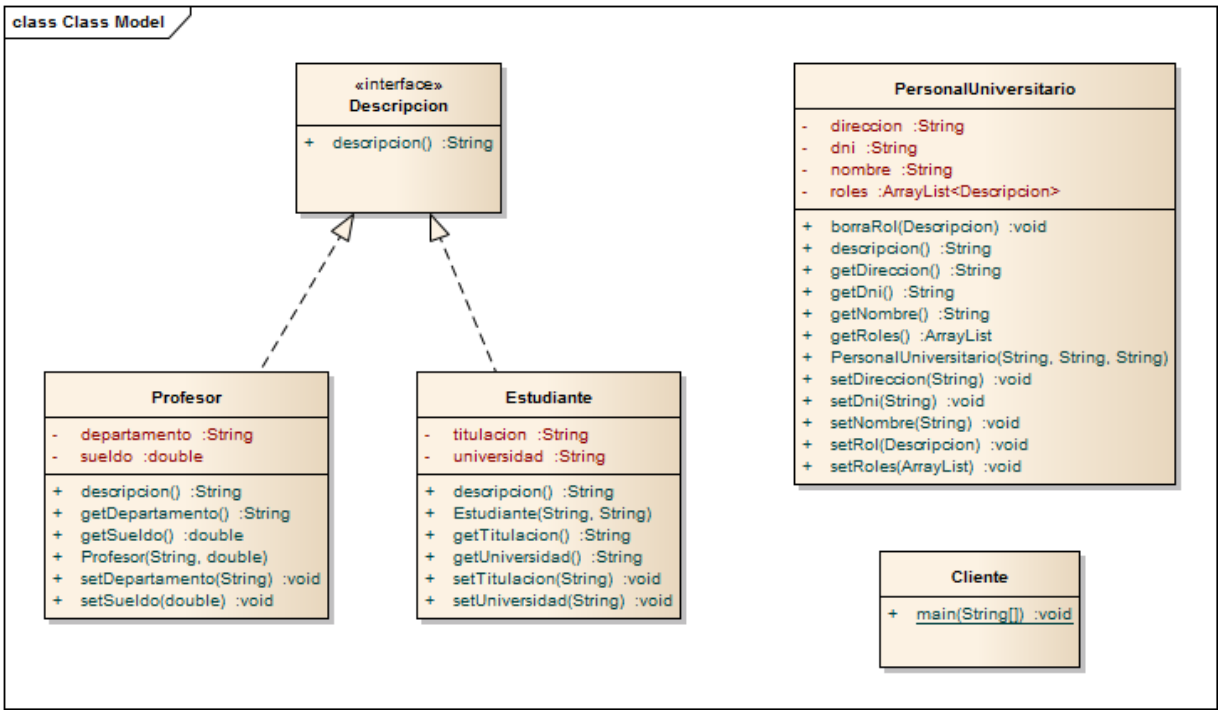
LINEA CONTINUA -> Apunta a una **superclase**
LINEA DISCONTINUA -> Apunta a una **interfaz**

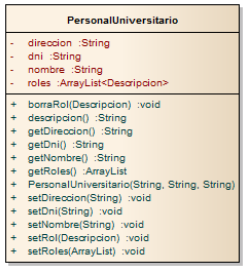
PATRÓN DELEGATION

- ¿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.

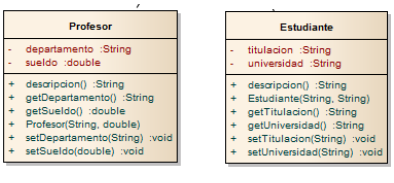
Relación **es un**	Relación **es un rol ejecutado por**
HERENCIA	PATRÓN DELEGATION

- 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.





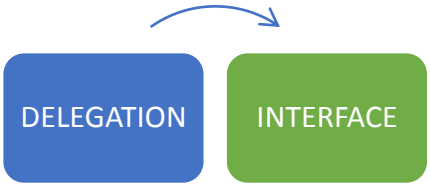
Clase general

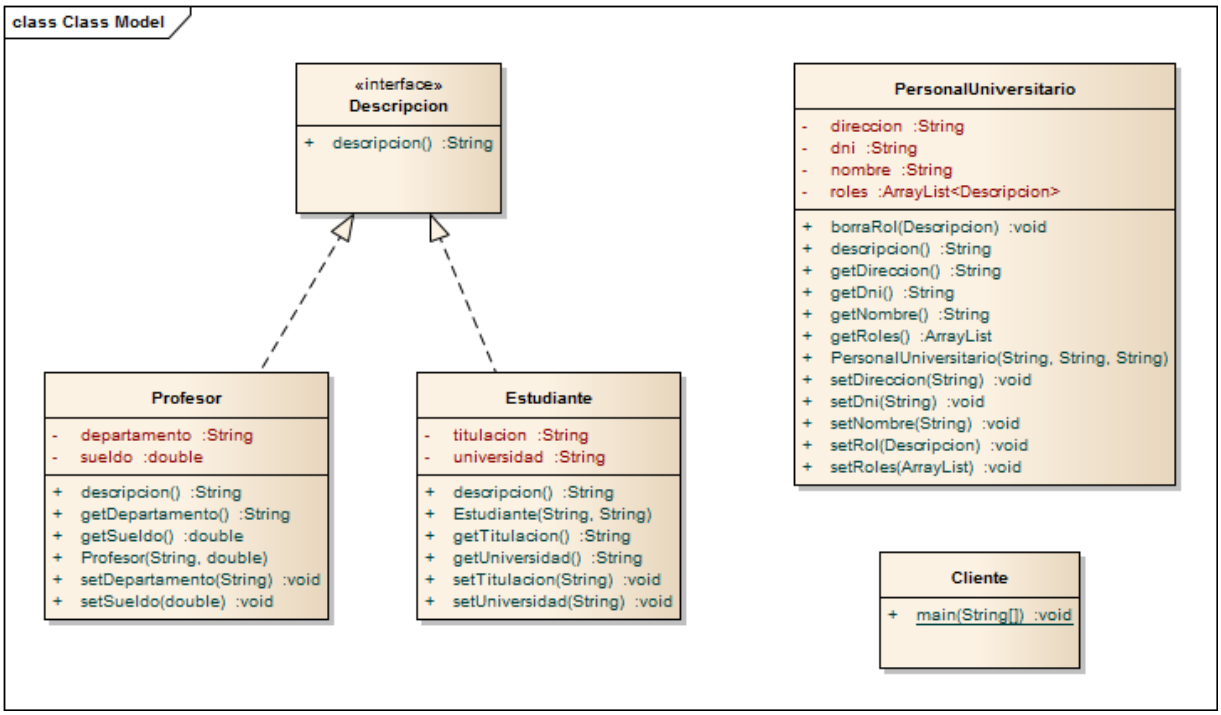


Clases concretas

PATRÓN INTERFACE

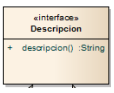
- La clave es la indirección.
- La interfaz provee a sus clases herederas de métodos específicos.
- La idea es que hay un contrato de servicios, es decir, el cliente sabrá la interfaz de los métodos sin necesariamente saber su codificación.
- La terminología a utilizar será: INTERFAZ, DELEGADOR, DELEGADOS.



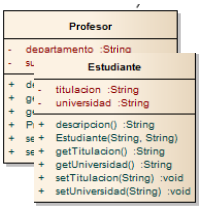


EJEMPLO:

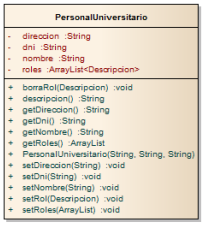
- Interfaz: Descripción (describe las interfaces de los métodos comunes de las clases concretas).
- Delegador: Personal universitario. Esta clase tiene un arreglo de elementos de tipo Descripción. El comportamiento de dichos elementos dependerá de la clase concreta.
- Delegado: Cada una de las clases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



Interfaz



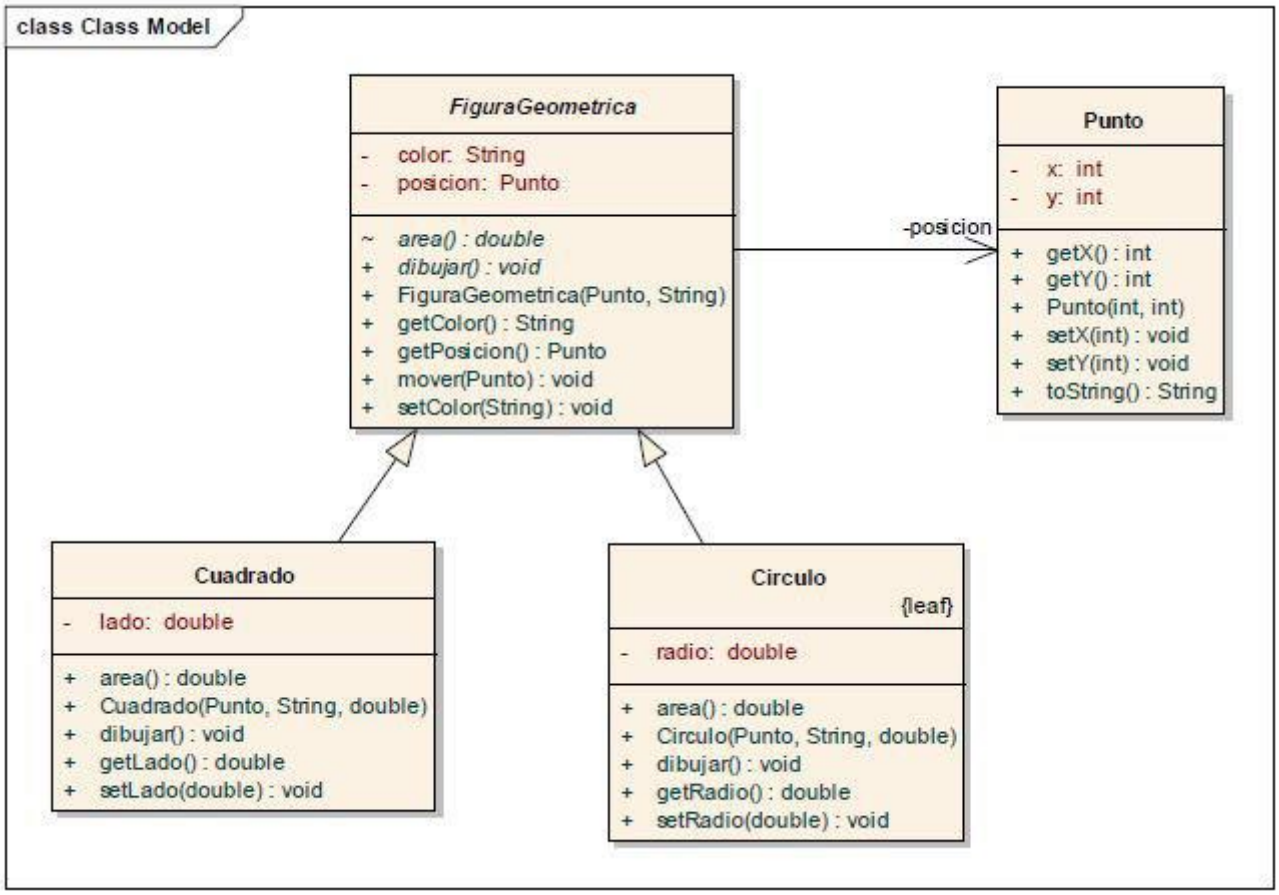
Clases delegadas



Clase delegadora

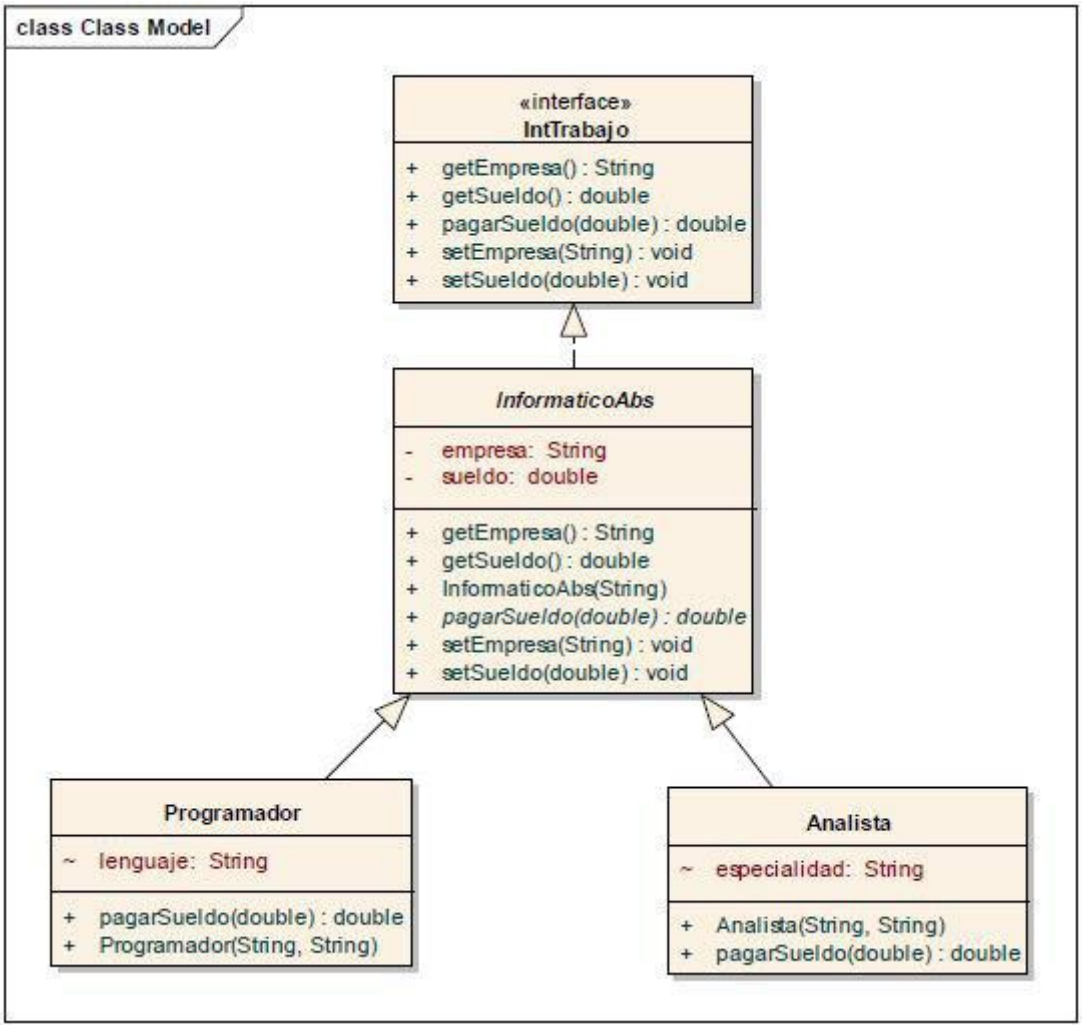
PATRÓN ABSTRACT SUPERCLASS

- 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

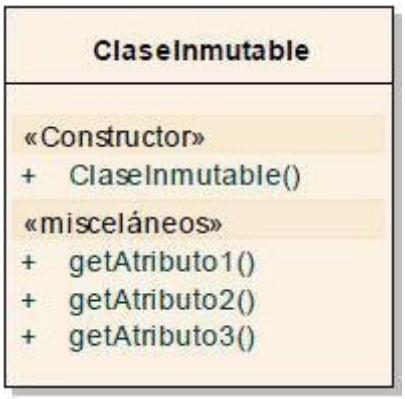
- Se proporciona una clase padre abstracta.
- Dicha clase abstracta es ocultada al cliente utilizando una interfaz de por medio.
- El cliente dispondrá de un "catálogo" de servicios, que son las interfaces puestas en el interface. Además, se tendrá la garantía de que las clases concretas implementan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



PATRÓN INMUTABLE

- 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



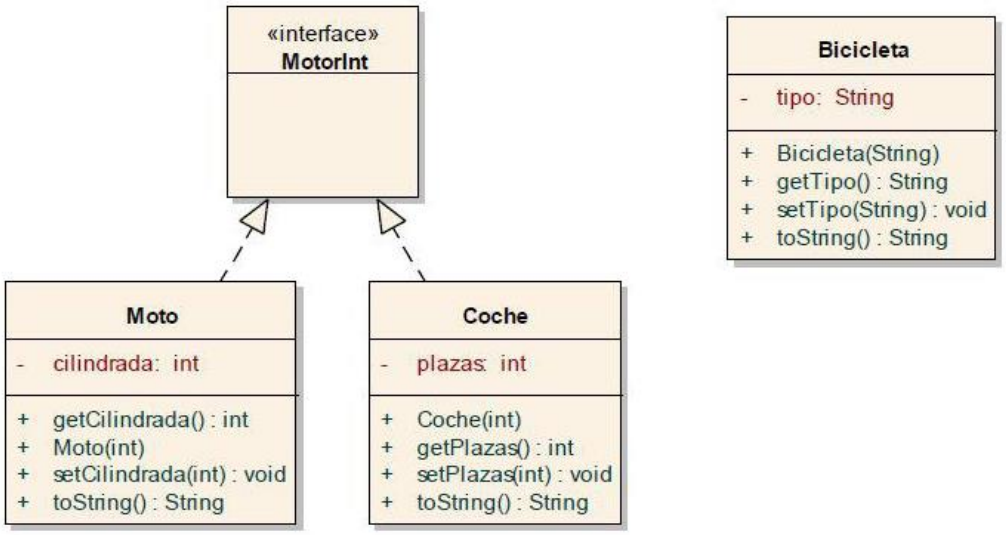

```
class Posicion {

        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);
        }

}
```

PATRÓN MARKER INTERFACE

- 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.

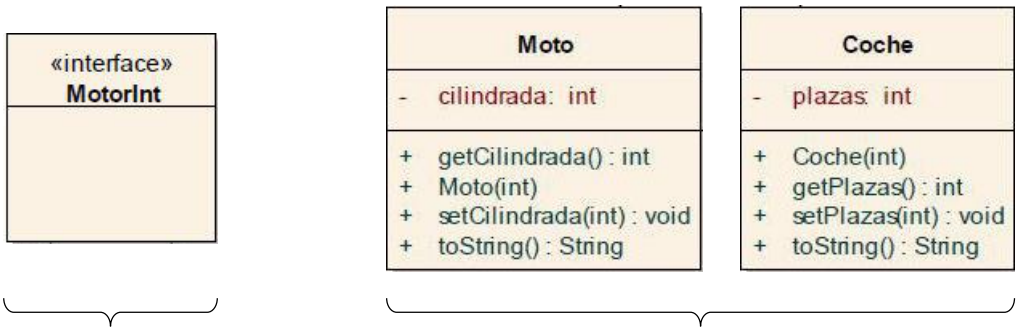
Interfaz Serializable en Java

Interface sin métodos ni atributos → PATRÓN MARKER INTERFACE



EJEMPLO:

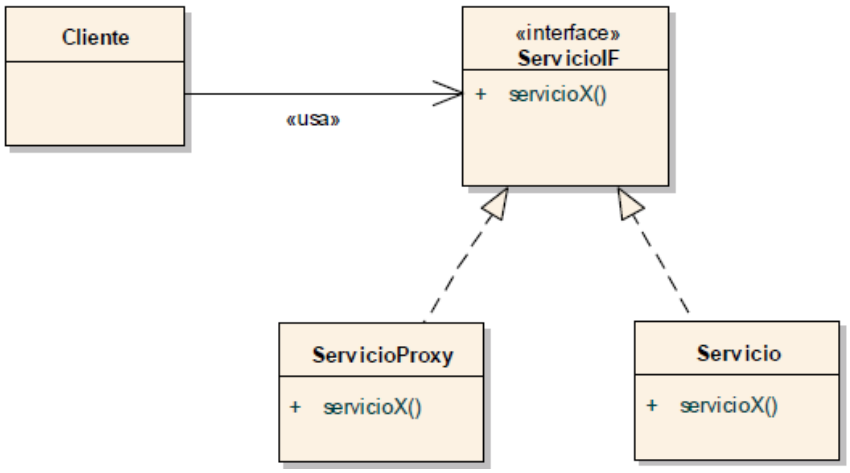
- Clase marcadora: **MotorInt**
- Clases marcadas: **Moto** y **Coche**



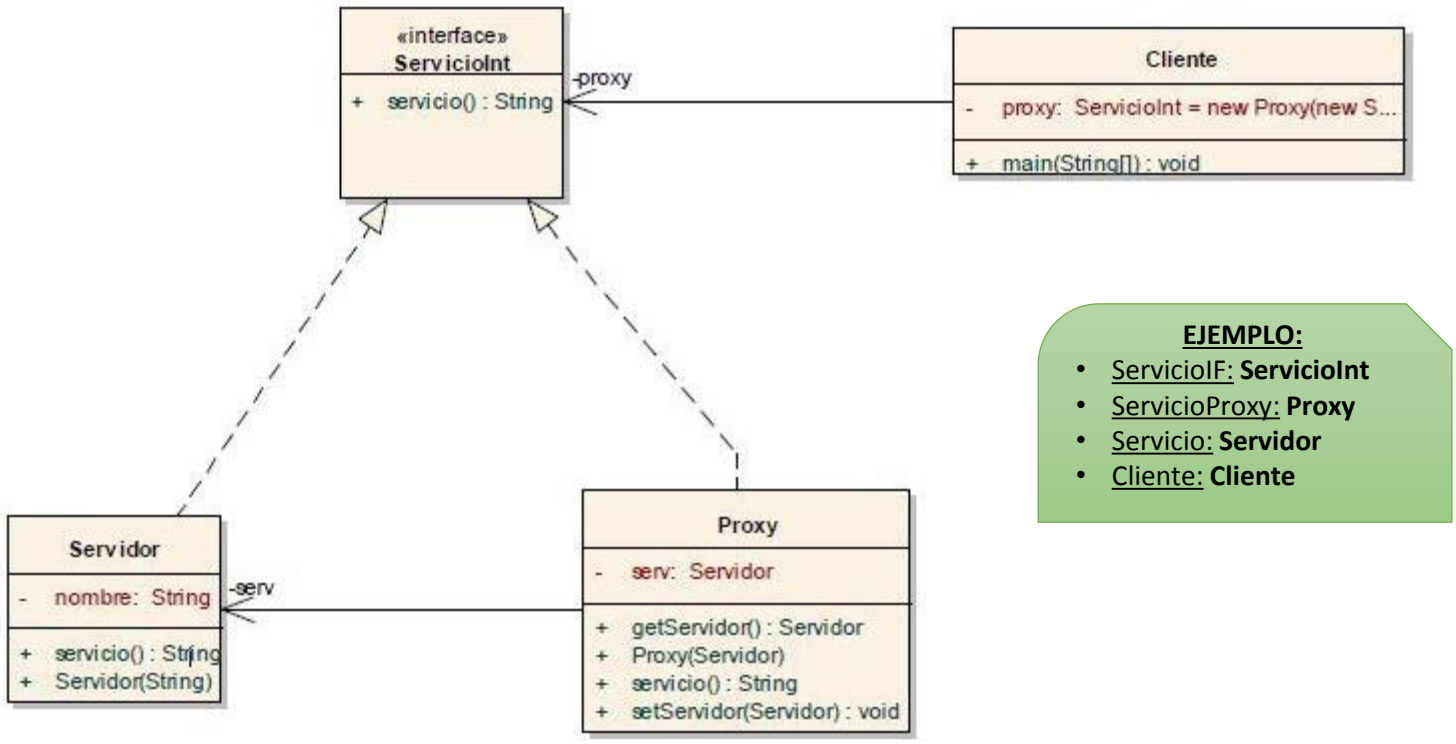
PATRÓN PROXY

- 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:

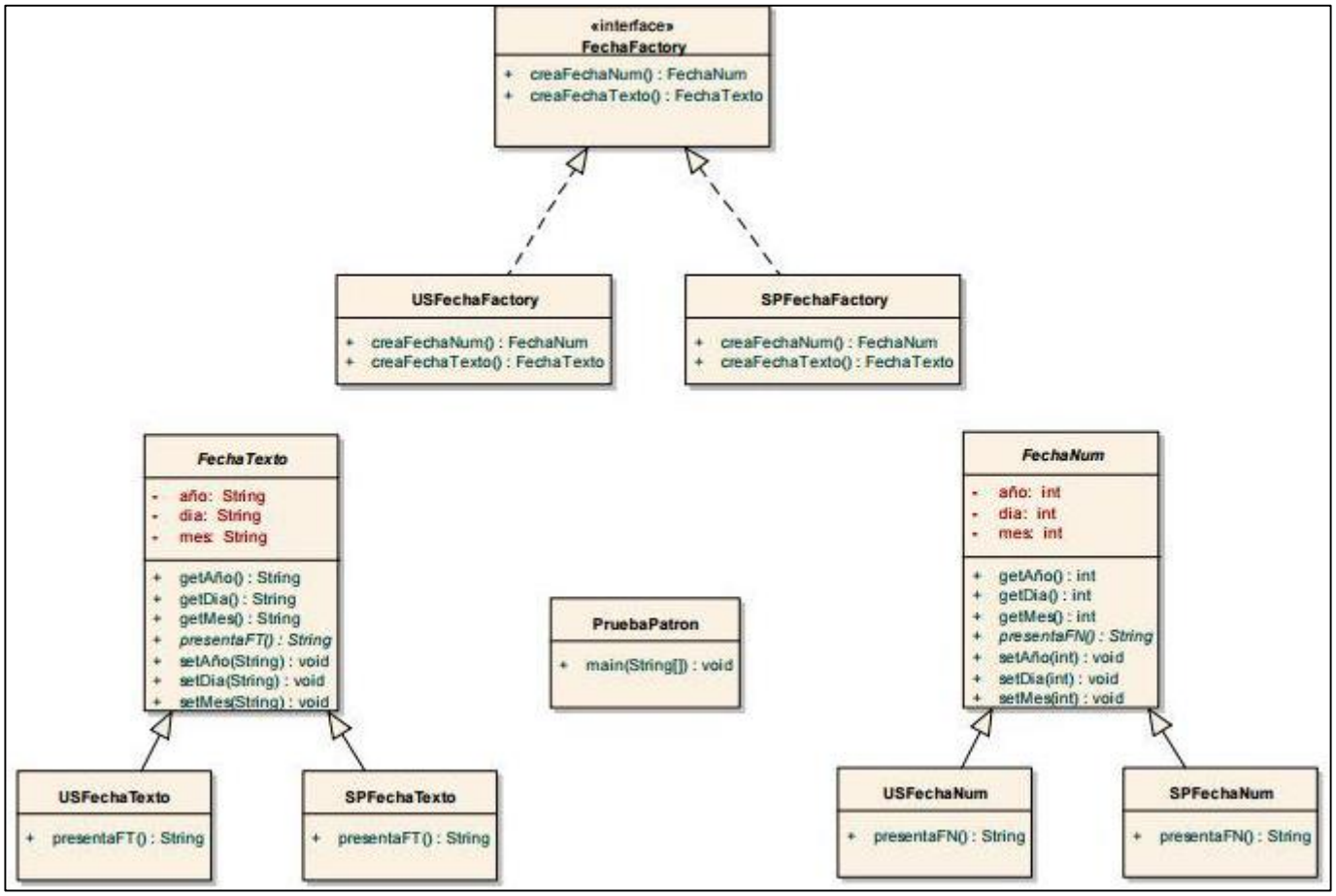
- ServicioIF: **ServicioInt**
- ServicioProxy: **Proxy**
- Servicio: **Servidor**
- Cliente: **Cliente**

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**

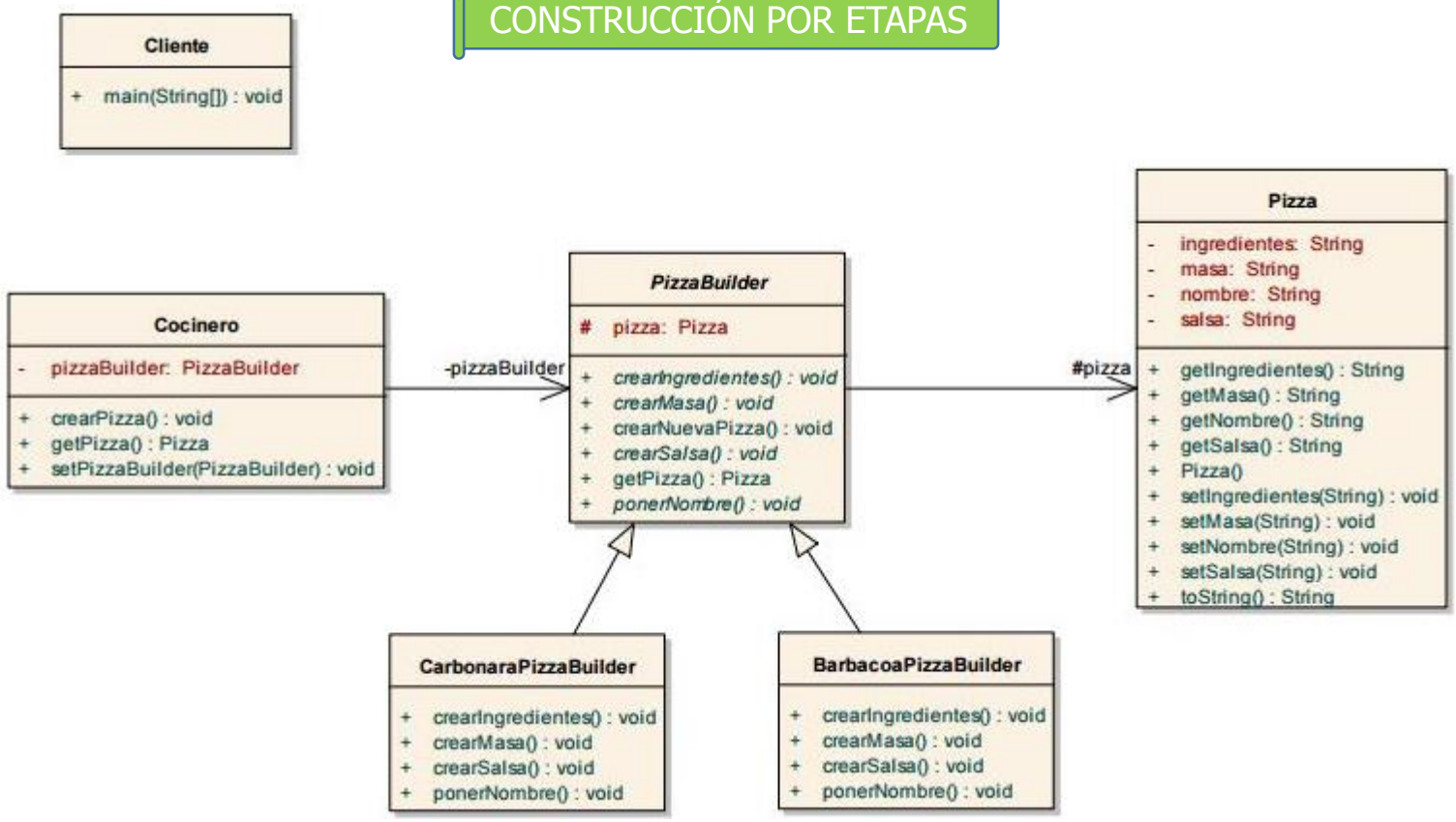
Las consecuencias del patrón son:

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.

CONSTRUCCIÓN POR ETAPAS



EJEMPLO:

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

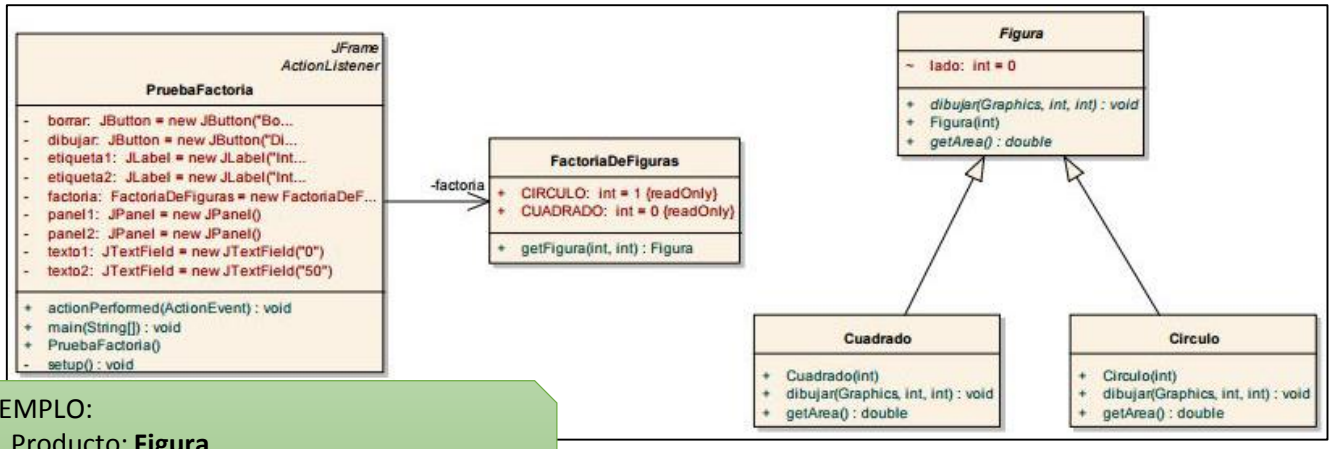
Las consecuencias del patrón son:

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:

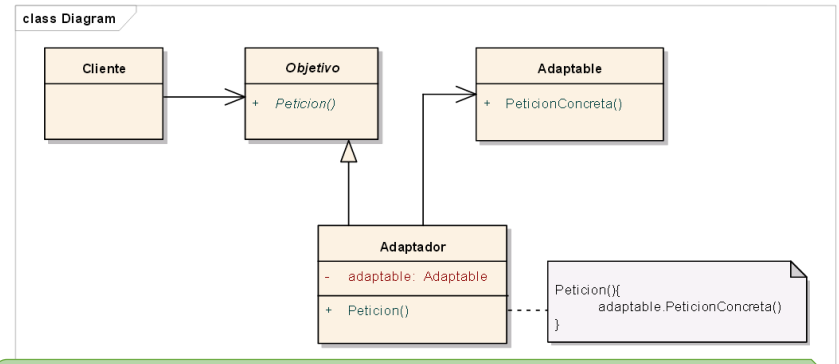
1. Proporciona enlaces para las subclases.
2. Conecta jerarquías de clases paralelas.

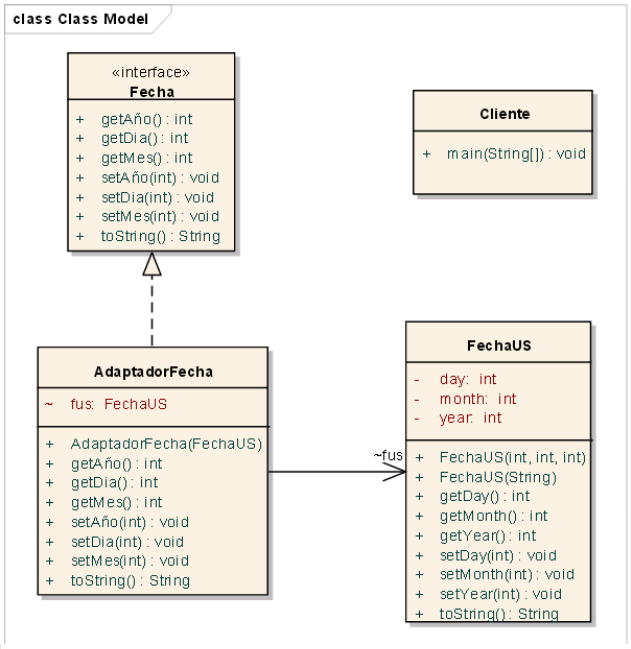
Patrones estructurales

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





EJEMPLO:

- Objetivo: **Define la interfaz especifica del dominio del cliente. - Fecha**
- Cliente: **Colabora con objetos que se ajustan a la interfaz objetivo. –Cliente**
- Adaptable: **Interfaz existente que necesita ser adaptada. – FechaUS**
- Adaptador: **Adapta la interfaz de adaptable a la interfaz objetivo. –Adaptador Fecha**

VARIACIONES

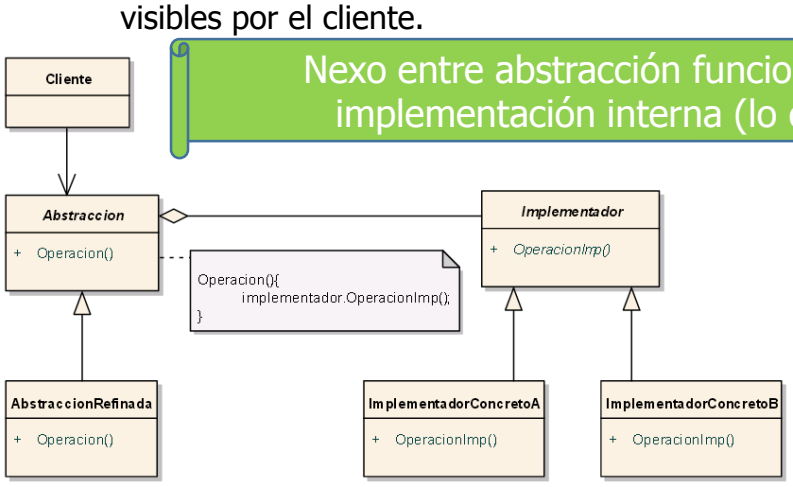
- Un adaptador y múltiples adaptables.
- Adaptadores no basados en interfaz ?.
- Capa de interfaz entre el invocador y adaptador, otra entre el adaptador y el adaptable: La capa entre el invocador y el adaptador permite que se puedan introducir fácilmente nuevos adaptadores en el sistema en tiempo de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

Las consecuencias del patrón son:

1. El cliente es independiente de las clases finales que utiliza.
2. Es posible utilizar la clase adaptador para monitorizar que clases llaman a que métodos de las clases finales.
3. En el **adaptador de clase**, la clase adaptadora…
   1. Puede redefinir y ampliar la interfaz de la clase adaptada.
   2. 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.
4. En el adaptador de objeto, la clase adaptadora…
   1. Permite que un mismo adaptador funcione con muchos adaptables
   2. Puede utilizar todas las subclases de la clase adaptada, ya que tiene constancia de los objetos que instancia.
   3. 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.

PATRÓN BRIDGE (Handle/Body)

- Desacopla un componente complejo en dos jerarquías relacionadas: La abstracción funcional de su implementación interna, así ambas partes pueden variar de forma independiente y cualquier cambio se hace de manera más fácil.
- Se utiliza cuando:
  - Para evitar un enlace permanente entre una abstracción y su implementación, evitando una relación estática entre ambas.
  - La abstracción y la implementación van a formar un árbol de clases derivadas. El patrón permite combinarlas y extenderlas de forma independiente.
  - Evita recompilar los clientes cuando cambia alguna implementación. Estos cambios no deben ser visibles por el cliente.



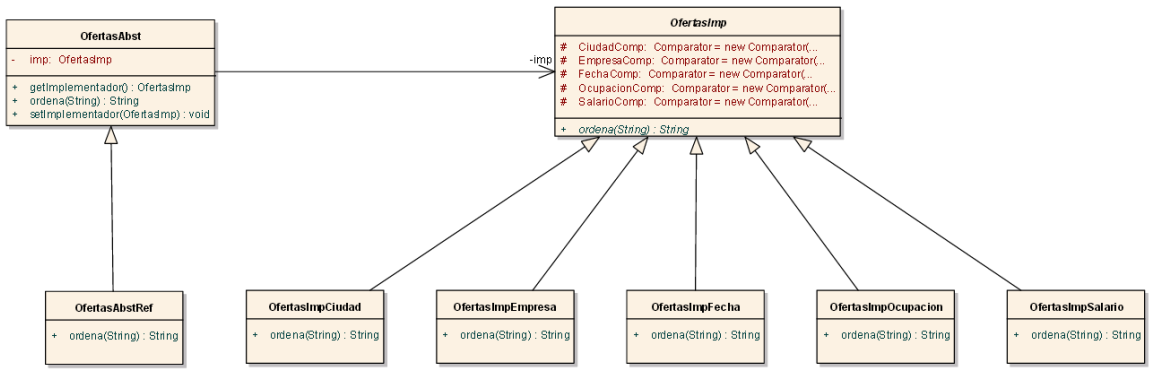
- Abstracción: **Define la interfaz de la abstracción mantiene una referencia al objeto implementador.**
- Abstracción refinada: **Extiende la interfaz definida por Abstracción.**
- Implementador: **Define la interfaz de las clases de implementación, no incluir abstracción.**
- ImplementadorConcreto: **Implementa la interfaz Implementador y define su interfaz concreta.**

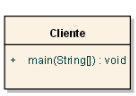
VARIACIONES

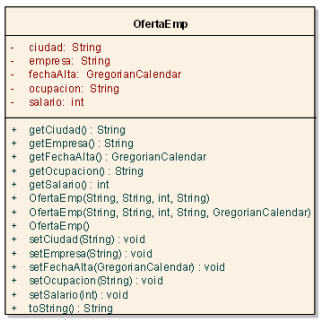
- **Implementador único:** en ocasiones solo existirá una clase implementador que sirve a muchas clases de abstracción. Si hay una única clase implementador, no es necesario definir una jerarquía de clases.

Las consecuencias del patrón son:

1. Se desacopla interfaz e implementación: ya que la implementación no está vinculada permanentemente a la interfaz, y se puede determinar en tiempo de ejecución (incluso cambiar). Se eliminan dependencias de compilación, y se consigue una arquitectura más estructurada en niveles.
2. Se mejora la extensibilidad, las jerarquías de abstracción y de implementación pueden evolucionar independientemente.
3. Se esconden detalles de implementación al cliente (Efecto de caja negra).







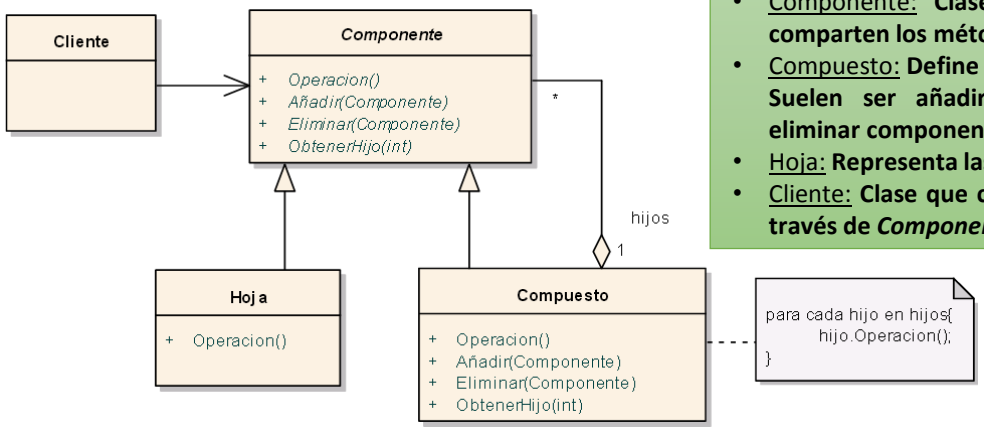
EJEMPLO:

- 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.
  - Desee 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

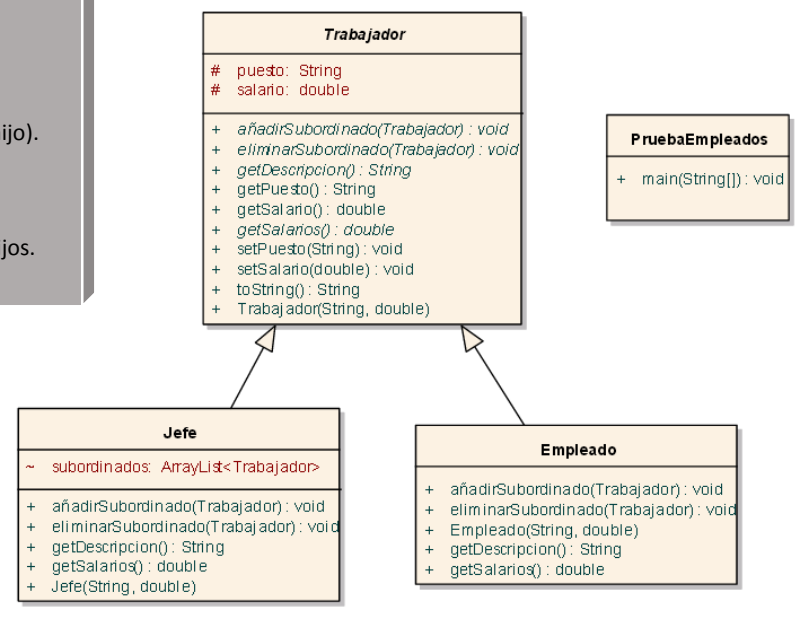
- **Nodo raíz:** Sirve para mejorar la manejabilidad del sistema, en ocasiones se crea un objeto que actúa como base para la jerarquía completa de objetos ***Composite***. Si el objeto raíz se representa como una clase separada, se puede implementar con un ***Singleton***, garantizando su acceso.
- **Ramificación 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:

1. Se crea una jerarquía de objetos básicos y de composiciones de estos, de forma recursiva.
2. Simplifica el cliente, al tratar todos los objetos de la misma forma.
3. 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.
4. **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)
- Destrucción de objetos.
- Utilización de iteradores para recorrer los integrantes.
- ***Visitor*** puede realizar operaciones para evitar distribuirlas por los hijos.

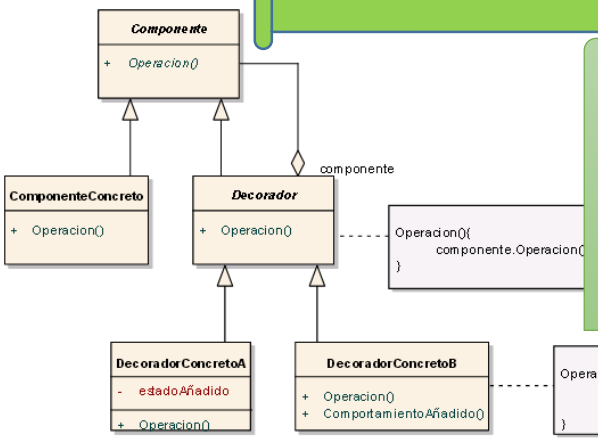


EJEMPLO:
- Componente: **Trabajador**
- Compuesto: **Jefe**
- Hoja: **Empleado**
- Cliente: **PruebaEmpleados**

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.



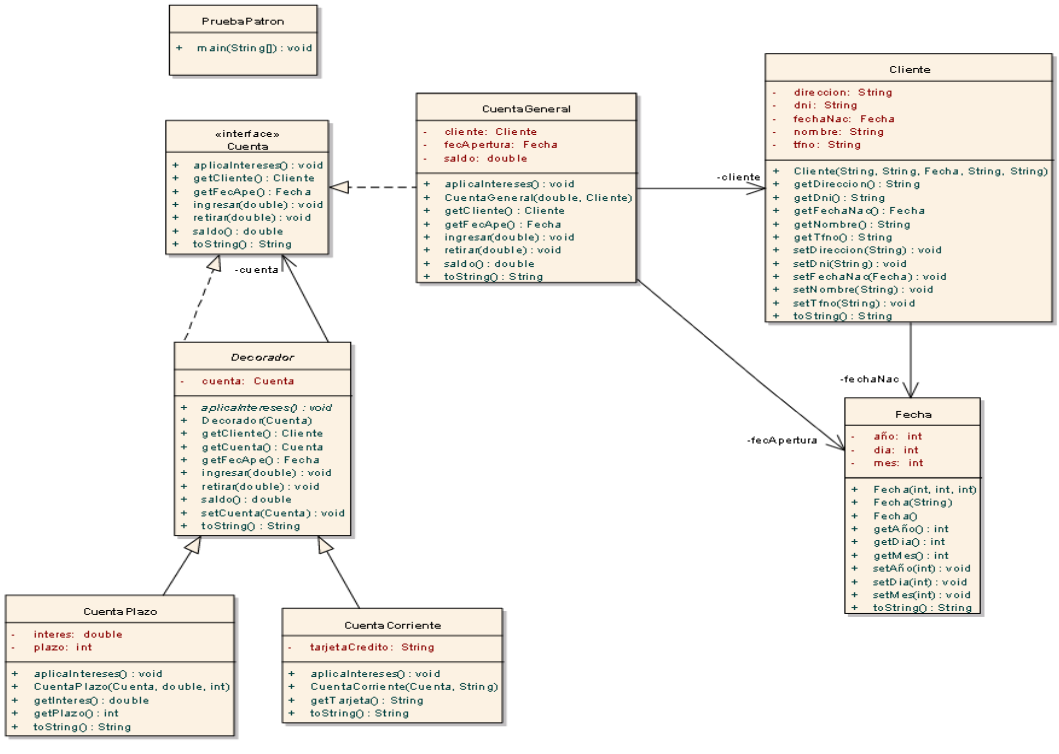
- Componente: **Interfaz común a todas las clases susceptibles de ser ampliadas con el Decorator. Contiene el comportamiento genérico.**
- Componente concreto: **Son las clases cuya interfaz se puede extender y en los que se delega en última instancia para realizar las operaciones propias de la clase.**
- Decorador: **Clase abstracta que declara la clase común a todos los decoradores y declaran (o implementan, según) la responsabilidad de mantener una referencia al objeto que se extiende. E posible que sobrecargue todos los métodos de la clase Componente con llamadas al componente concreto, de forma que aquellos métodos cuya funcionalidad no se extiende simplemente llaman a los originales.**
- Decorador Concreto: **Son clases concretas que heredan de Decorador e implementan las extensiones de funcionalidad de la clase original Componen concreto.**

VARIACIONES

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



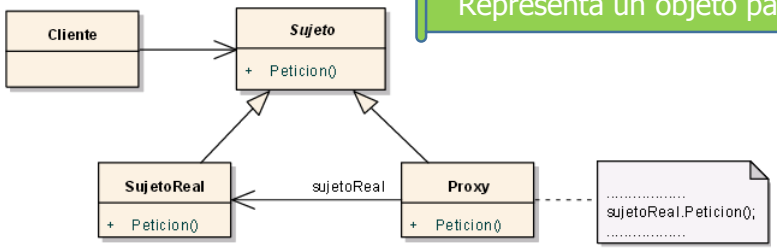
EJEMPLO:

- Componente: **Cuenta**
- ComponenteConcreto: **CuentaGeneral**
- Decorador: **Decorador**
- DecoradorConcreto: **CuentaCorriente, CuentaPlazo**

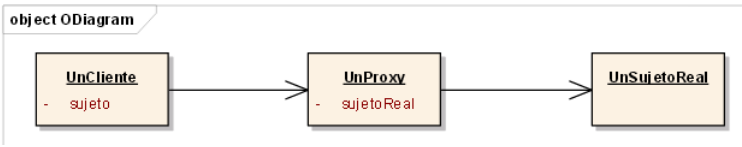
PATRÓN PROXY (Virtual Proxy o Surrogate)

- 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.

Representa un objeto para controlar el acceso.



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:

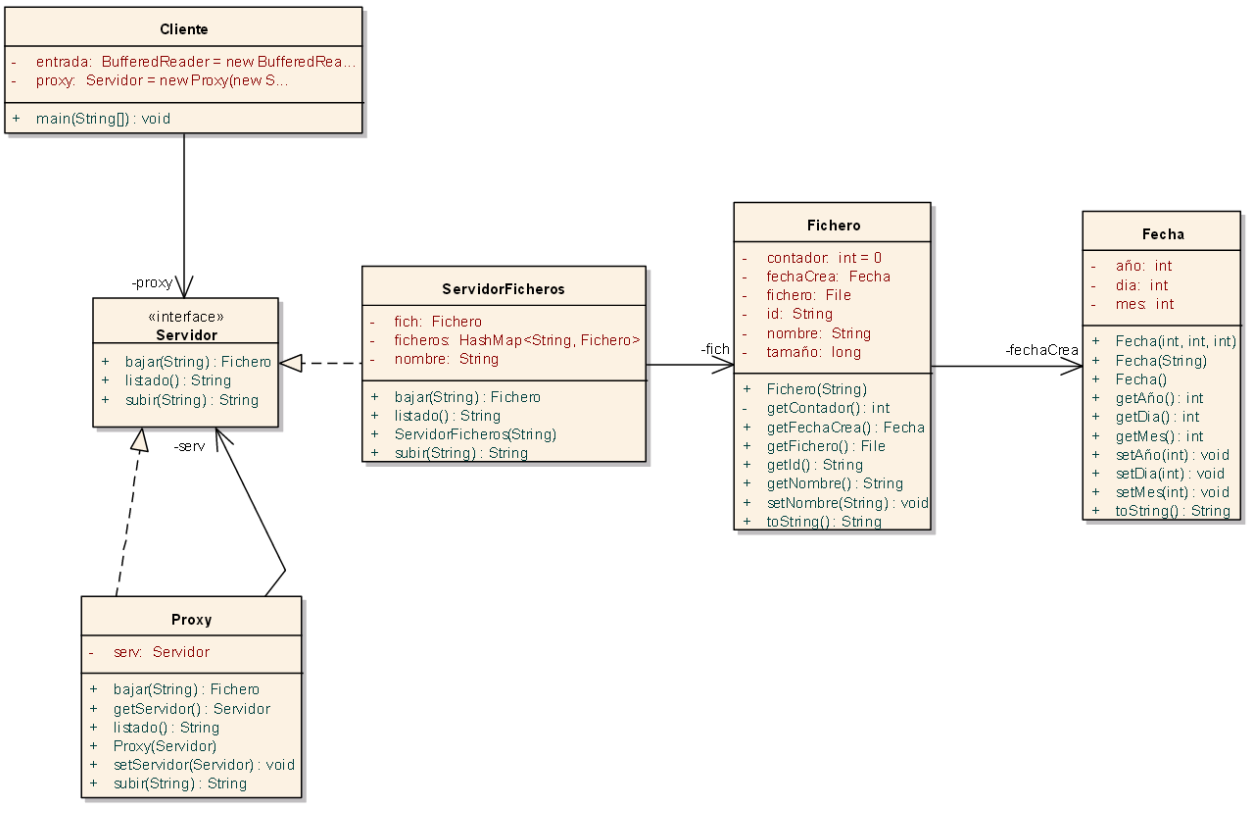
1. El patrón ***Proxy*** introduce un nivel de indirección al acceder a un objeto.
2. Puede mejorar la eficiencia al retrasar la instanciación de un objeto costos hasta que sea necesario utilizarlo. Así el objeto proxy lo sustituye ofreciendo la misma interfaz, solo cuando es necesario le solicita al objeto real la información que necesita.
3. Aumenta la seguridad.
4. 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**
- Sujetoreal: **ServidorFicheros**
- Cliente: **Cliente**

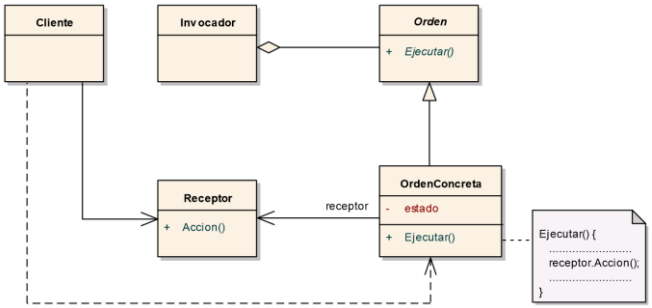


Patrones de comportamiento

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 ldel sistema.
  - Estructurar un sistema mediante operaciones de alto nivel basadas en operaciones más sencillas (primitivas). Sistemas transaccionales.

Permite parametrizar a los clientes con diferentes peticiones.



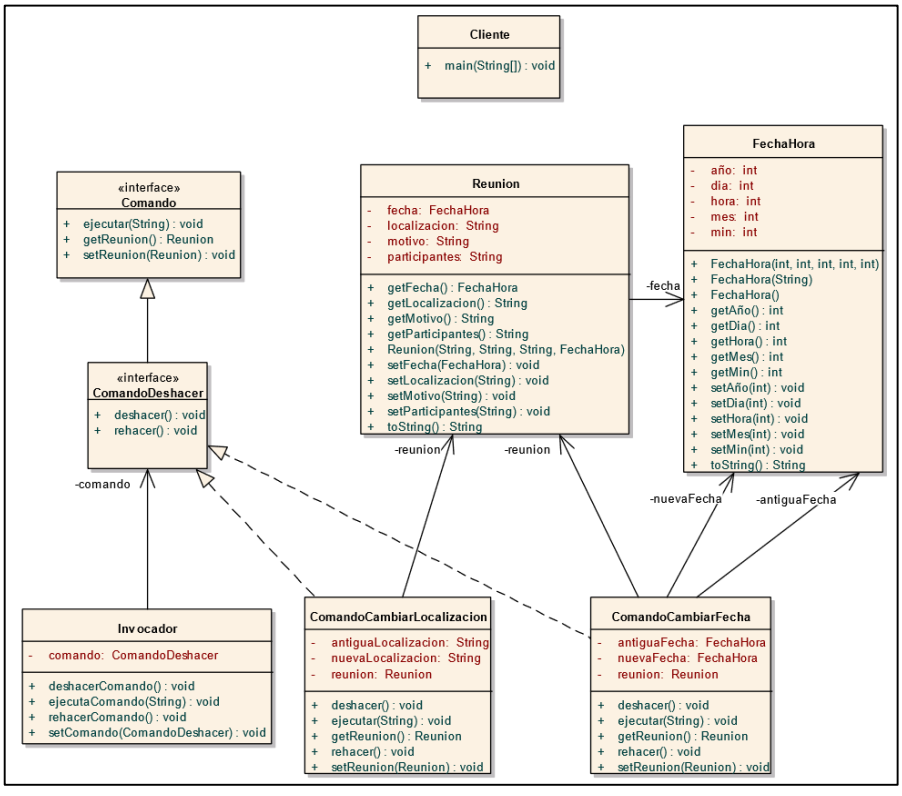
- Orden (comando): **Interfaz en la que se definen los métodos que se usarán por el invocador. Además estos métodos serán implementados por cada uno de los comandos concretos. En el diagrama solo se muestra el método ejecutar, pero se pueden añadir más métodos como deshacer y rehacer.** ***ComandoDeshacer***
- 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.** ***ComandoCambiarFecha, 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:

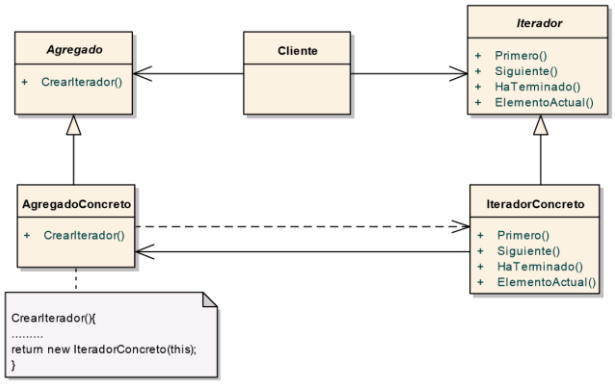
1. Se desacopla la parte de la aplicación que invoca a los comandos de la implementación de los mismos.
2. Al tratarse los comandos como objetos, se puede realizar herencia de los mismos, composiciones de comandos (mediante el patrón ***Composite***).
3. Permite reemplazar objetos comando en tiempo de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



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.

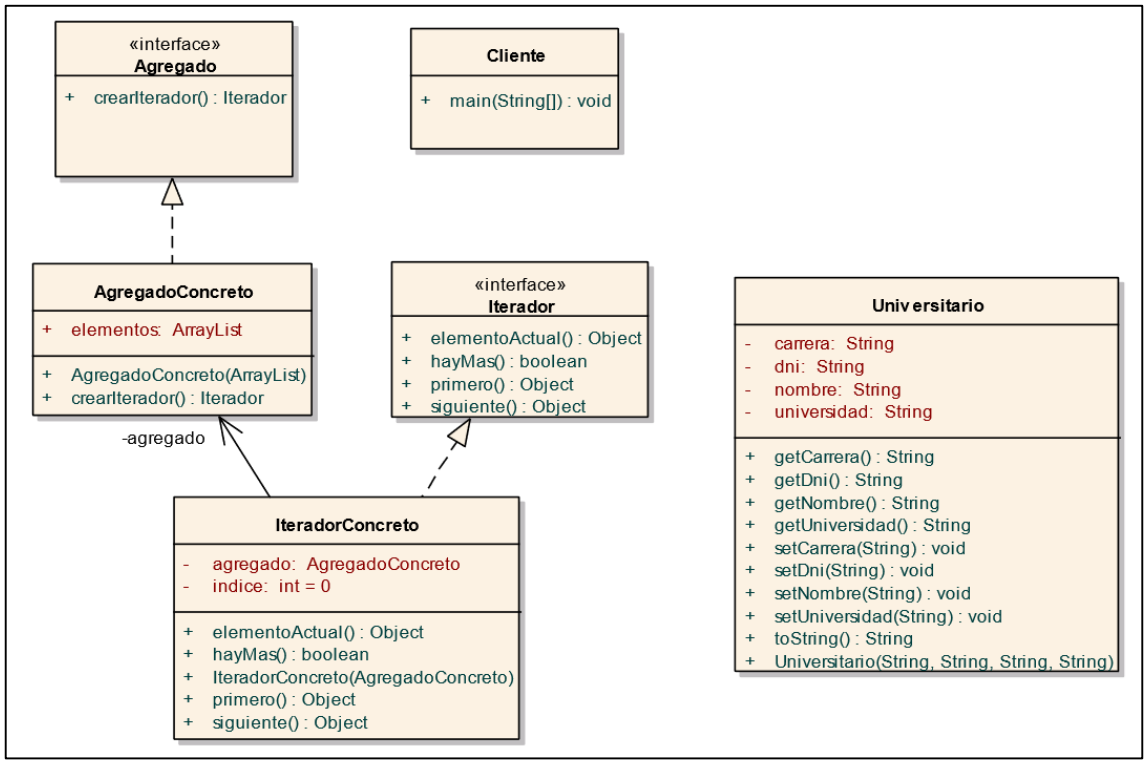
Recorrer una colección de datos indeterminados.



- 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:

1. Podemos recorrer una estructura de objetos sin conocer los detalles internos de sus clases.
2. Al definir una interfaz uniforme para los iteradores concretos, se simplifica el recorrido de las colecciones y se permite utilizar polimorfismo.
3. Al tener que recorrer agregados complejos podemos hacerlo de muchas formas distintas. Facilitan cambiar el algoritmo de recorrido.
4. 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.



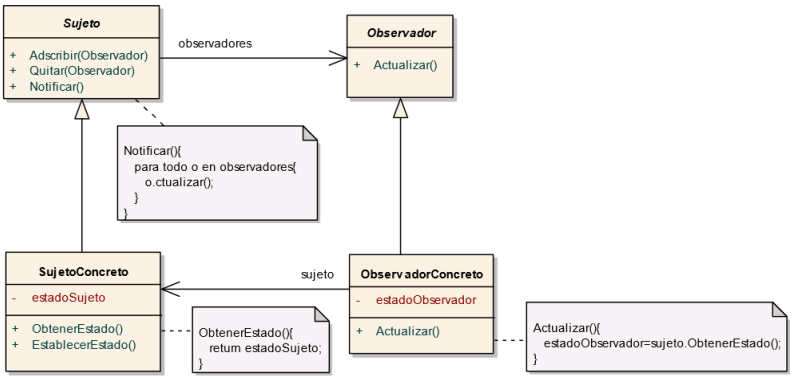
Iteradores en Java

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

PATRÓN 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.

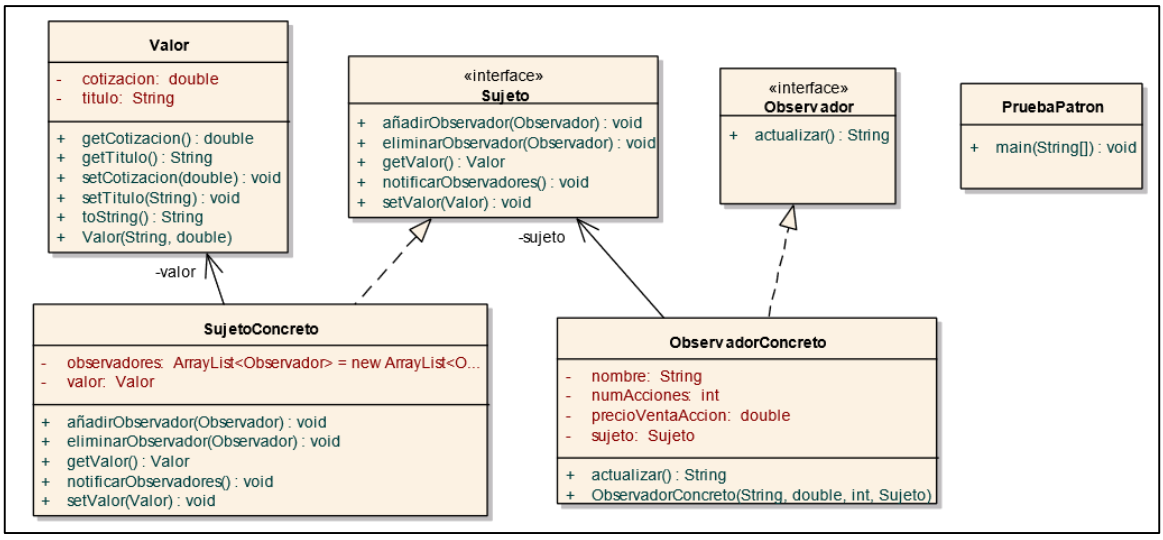
Los objetos suscritos reciben lo que el padre (emisor, publicador) publica.



- 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 lsita 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.



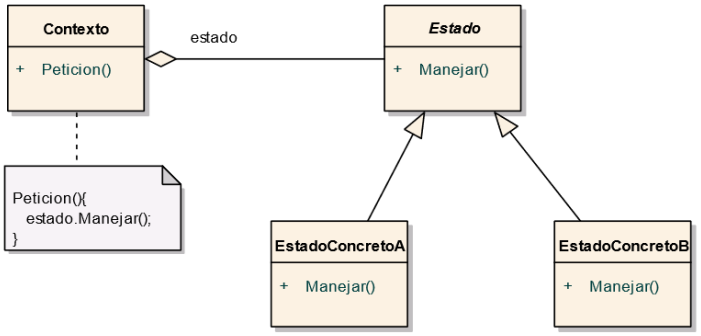
Observadores en Java

Em 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.

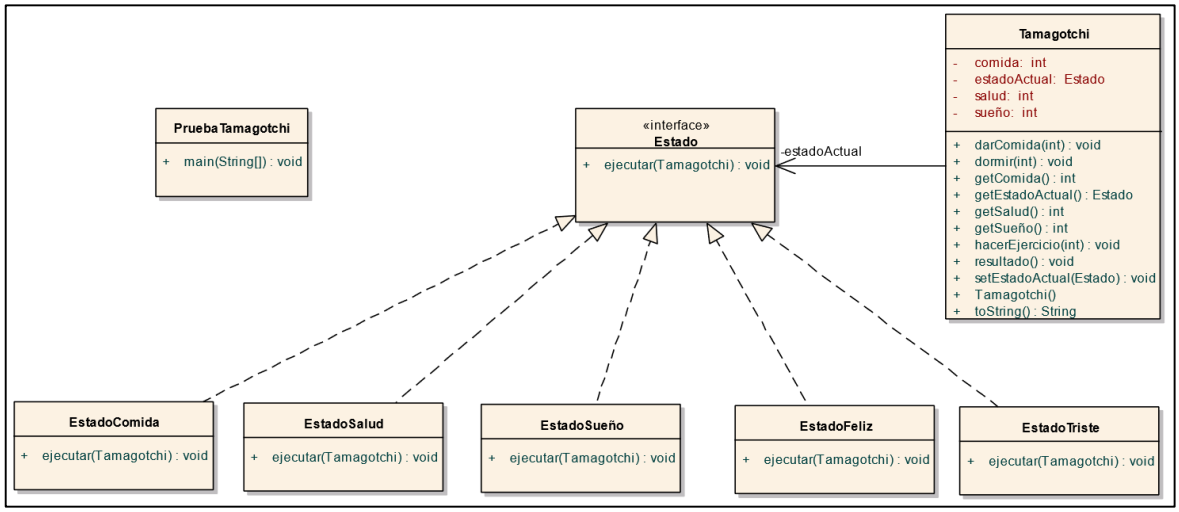
Permite crear clases que modelan comportamiento en vez de datos.



- 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:

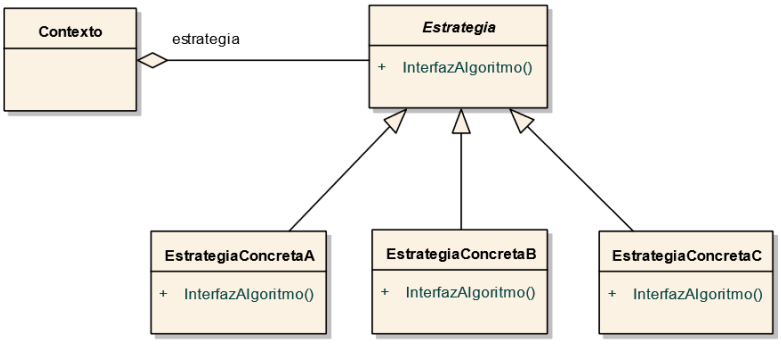
1. Flexibiliza la creación de nuevos estados y transiciones por lo que se obtiene una visión mucho más clara del comportamiento. Para crear un nuevo estado, únicamente hay que crear una nueva clase que implemente la interfaz.
2. El flujo de control del código que depende del estado se hace patente en la jerarquía de clases.
3. Los estados están ahora encapsulados en objetos por lo que se facilita el reconocimiento de un cambio entre estados.
4. **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.

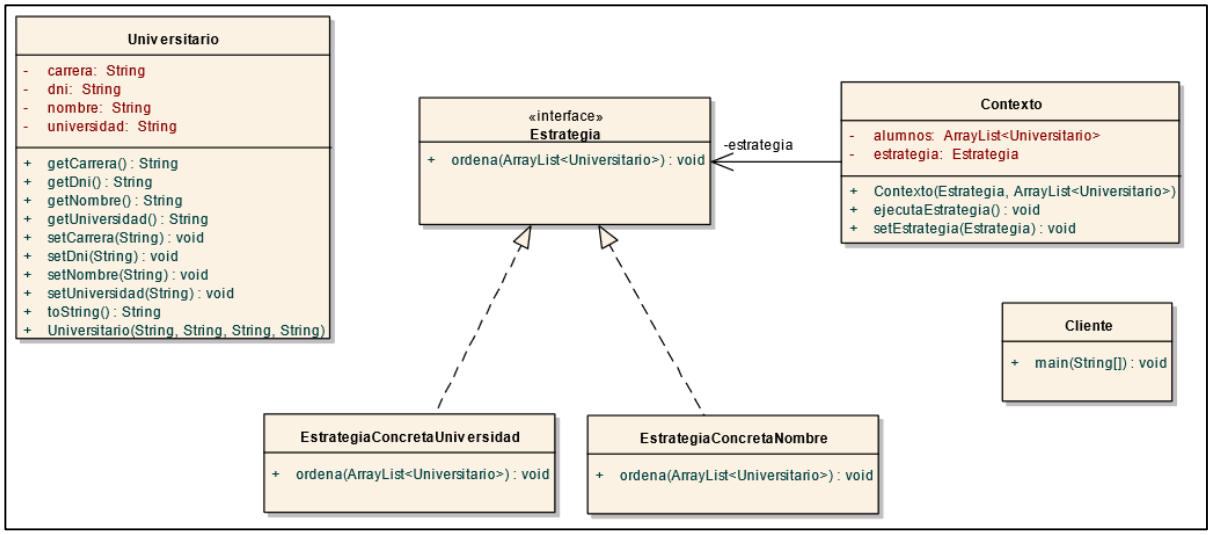
Permite crear clases que modelan comportamiento en vez de datos.



- Contexto: **Clase que utiliza las diferentes estrategias para las distintas tareas. Mantiene una referencia a la instancia *Estretegia* 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:

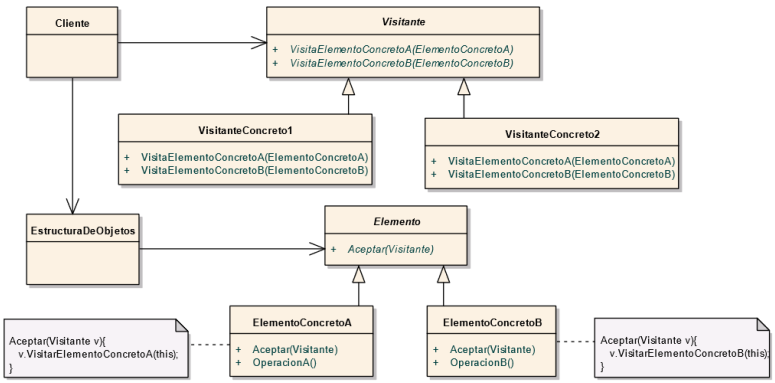
1. Cada comportamiento se define en su propia clase, por lo que se consigue que sean más fáciles de mantener.,
2. Resulta sencillo ampliar el modelo para incorporar nuevos comportamientos sin el alto coste de modificar todo el código de la aplicación o la utilización de la herencia.
3. Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento.
4. 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.

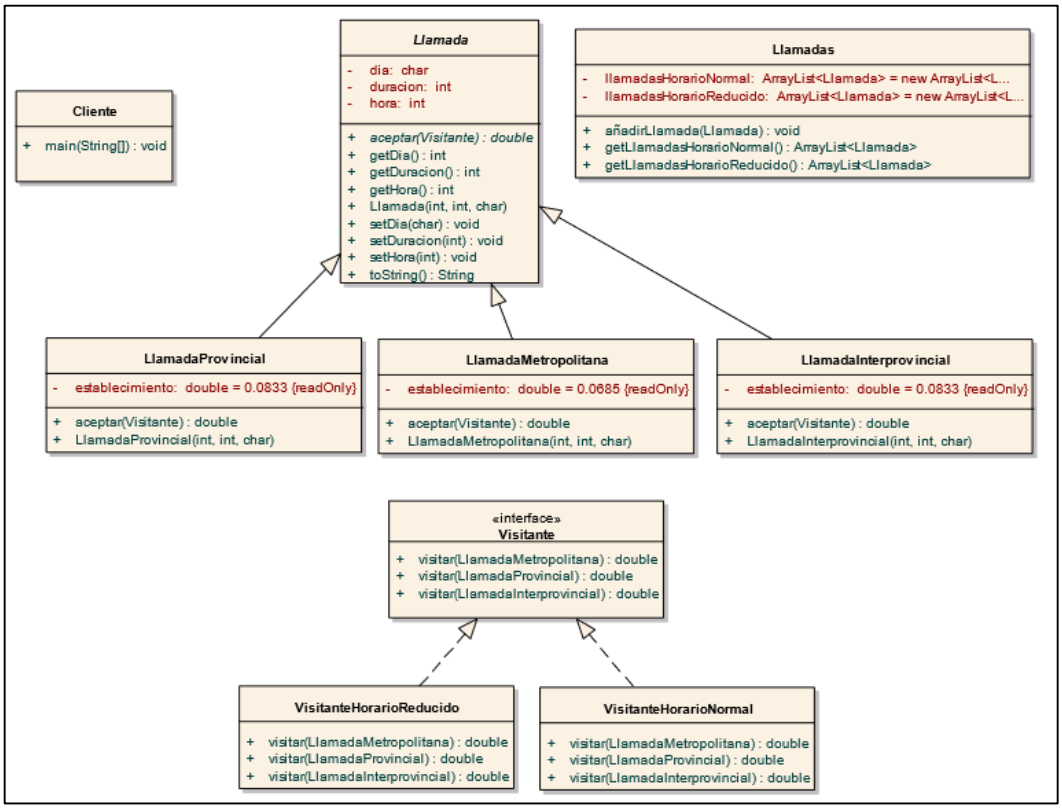
Visitamos una clase y según las circunstancias ciertas acciones se ejecutan o no.



- 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 *aceptar* invocando el método *visitar* apropiado definido en *Visitante*. *LlamadaProvincial, LlamadaMetropolitana y LlamadaInterprovinial.***
- EstructuraDeObjetos: **Para poder enumerar todos los elementos. *Llamadas***

Las consecuencias del patrón son:

1. 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.
2. ***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.
3. Un visitante agrupa operaciones relacionadas y se para el comportamiento no relacionado en las subclases del visitante.
4. **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 componenetes 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
		Spagueti 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 Responsability Assignment Software Pattern)

- GRASP es el acrónimo de General Responsability 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

- 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 cuando 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(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.

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

- **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.

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