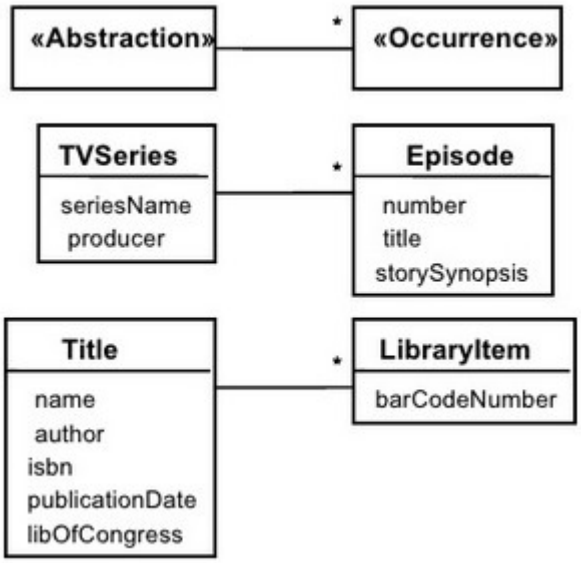
Abstracción-caso (Abstraction-occurrence)

Contexto: frecuentemente en un modelo de dominio se pueden encontrar un conjunto de objetos relacionados que llamaremos casos.
Los miembros de este conjunto comparten una información común, pero también se diferencian entre ellos de manera importante.

Fuerzas: representación de conjuntos de casos sin duplicar información común; prever inconsistencias cuando se cambia la información en algunos objetos pero no en otros; "flexibilizar" la posible solución al máximo.

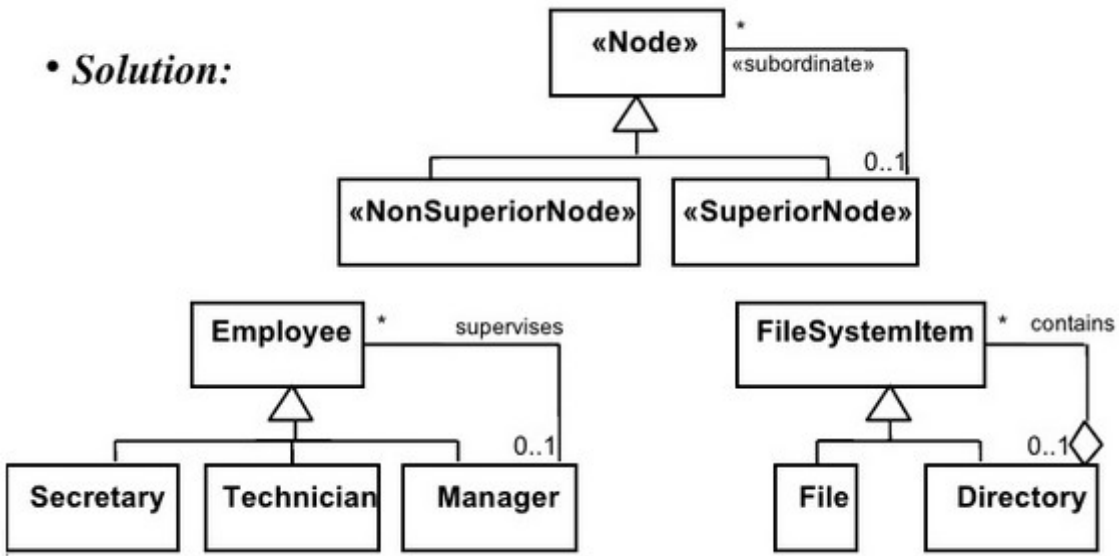


Jerarquía General (General Hierarchy)

Contexto: cada uno de los objetos en dicha jerarquía puede tener uno o más objetos encima de él (superiores) y cero o más elementos debajo (subordinados). Algunos objetos puede que no posean subordinados.

Problema: ¿Cómo se puede dibujar un diagrama de clases para representar una jerarquía de objetos en la que algunos objetos no puedan tener subordinados?

Fuerzas: se quiere encontrar una forma flexible de representar la jerarquía que, de manera natural, impida a determinados objetos tener subordinados. También hay que tener en cuenta que todos los objetos de la jerarquía comparten características comunes.

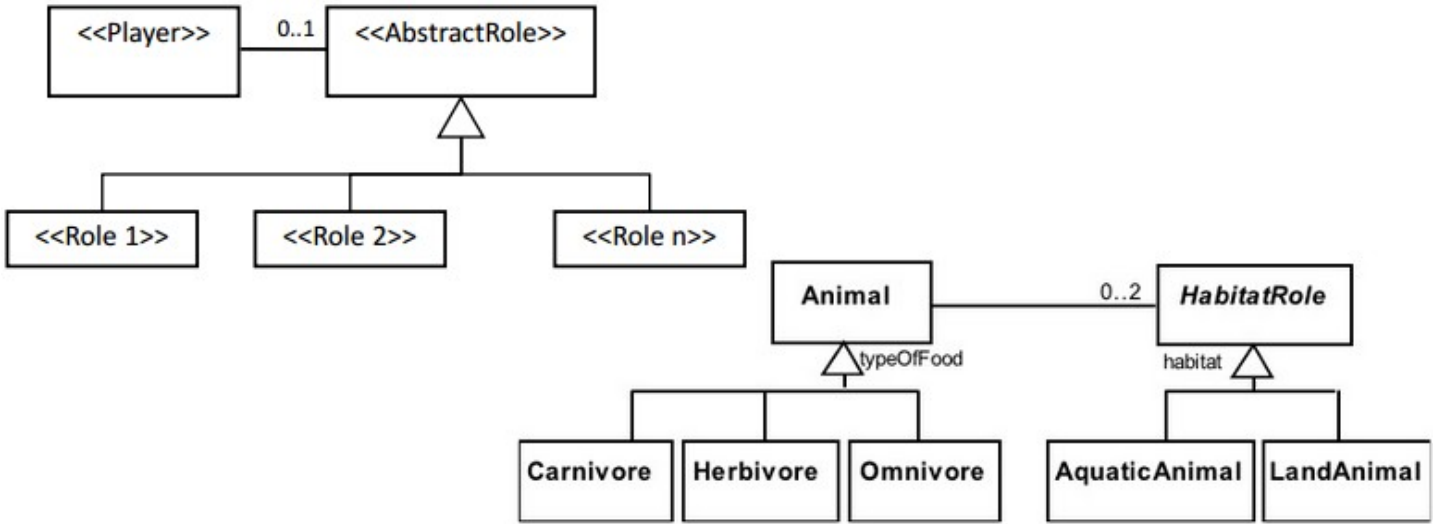


Actor-Rol (Player-Role)

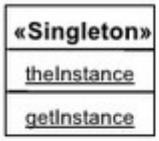
Contexto: Un rol es un conjunto particular de características asociadas a un objeto en un contexto particular. Un objeto actor puede jugar diferentes roles en distintos contextos.

Problema: ¿Cómo se pueden modelar actores y roles de tal manera que un actor pueda cambiar de rol o asignársele múltiples roles?

Fuerzas: es aconsejable mejorar la encapsulación capturando la información asociada a cada rol diferente que puede desempeñar un actor dentro de una clase. Pero esto se debe conseguir sin que sea necesario recurrir a la herencia múltiple ni tampoco permitir que una instancia pueda cambiar una clase.

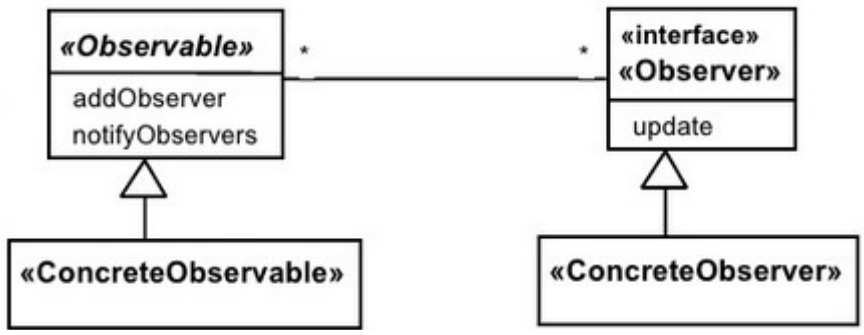


Singleton



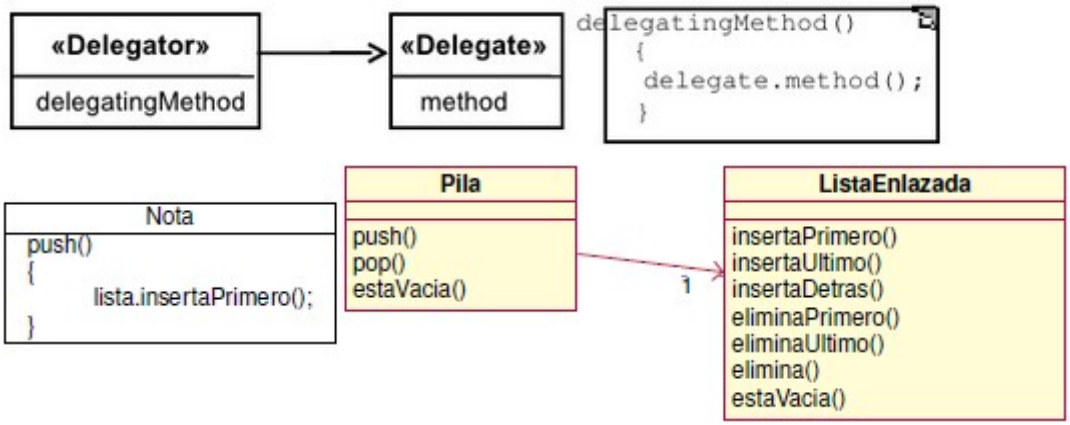

```
public class Singleton {
        static Singleton theInstance
                                = new Singleton();
        private Singleton() { }
        public static Singleton getInstance()  {
                return theInstance;
        }
}
```

Observable/Observador (Observer)



Delegación (Delegation)

Contexto: se necesita una determinada operación y nos damos cuenta que otra clase tiene ya una implementación de la citada operación, sin embargo, para heredar la operación que necesitamos, no nos conviene hacer a la primera una subclase de la segunda ya que, o bien no se puede aplicar la relación "IS A" con propiedad o no queremos heredar todos los métodos de la otra clase.



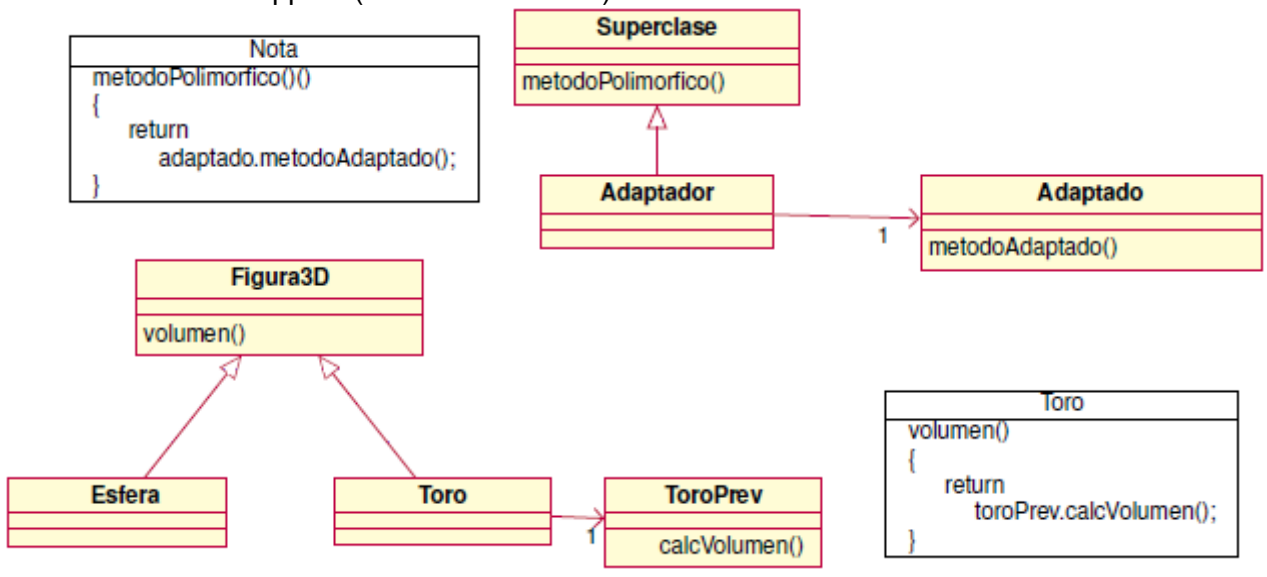
Patrones relacionados: Adaptador y Proxy.

Adaptador (Adapter)

Contexto: Se trata de reutilizar una clase no-relacionada y pre-existente a las contenidas en una jerarquía que estamos construyendo. Normalmente, los métodos de la clase reutilizada no tienen los mismos tipos de argumentos que los de la clase en la jerarquía que estamos creando.

Problema: ¿Cómo obtener la potencia del polimorsmo cuando reutilizamos una clase cuyos métodos poseen la misma función pero no queremos que aparezca en la jerarquía de clases que estamos diseñando?

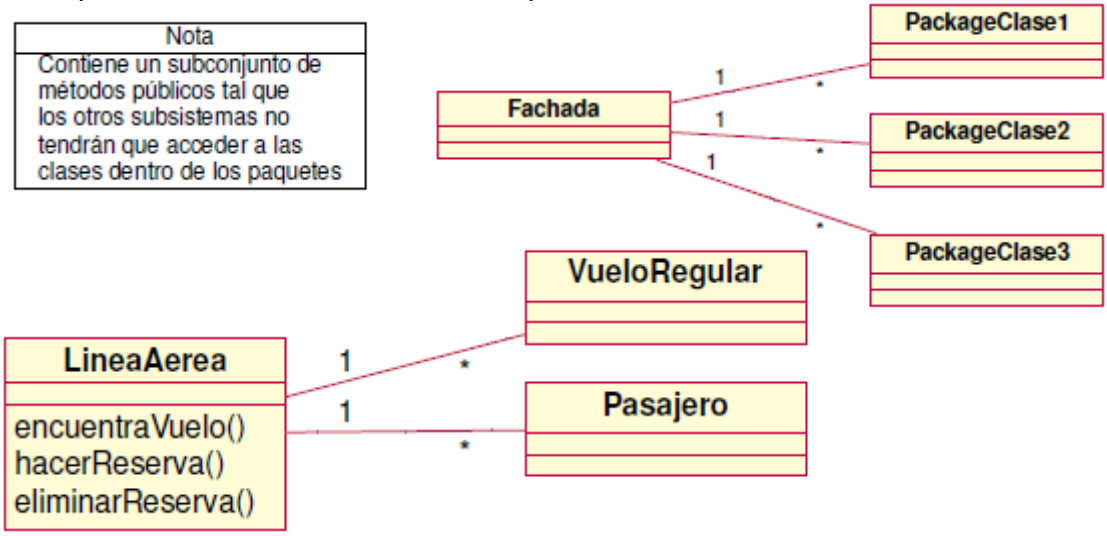
Fuerzas: no se tiene acceso a la herencia múltiple o no se la quiere utilizar. A los adaptadores se les llama a veces wrappers ("clases envoltorio").



Fachada (Façade)

Contexto: A menudo, una aplicación contiene varios paquetes complejos, un programador trabajando con tal cantidad de paquetes, tiene que trabajar con clases muy diferentes.

Problema: ¿Cómo podemos simplificar a los programadores la vista de un paquete complejo? Hay que tener en cuenta que si varias clases en los programas de aplicación llamasen a métodos del paquete complejo, cualquier modificación que se hiciera en dicho paquete necesitara de una revisión completa de todas estas clases de la aplicación.



Inmutable (Immutable)

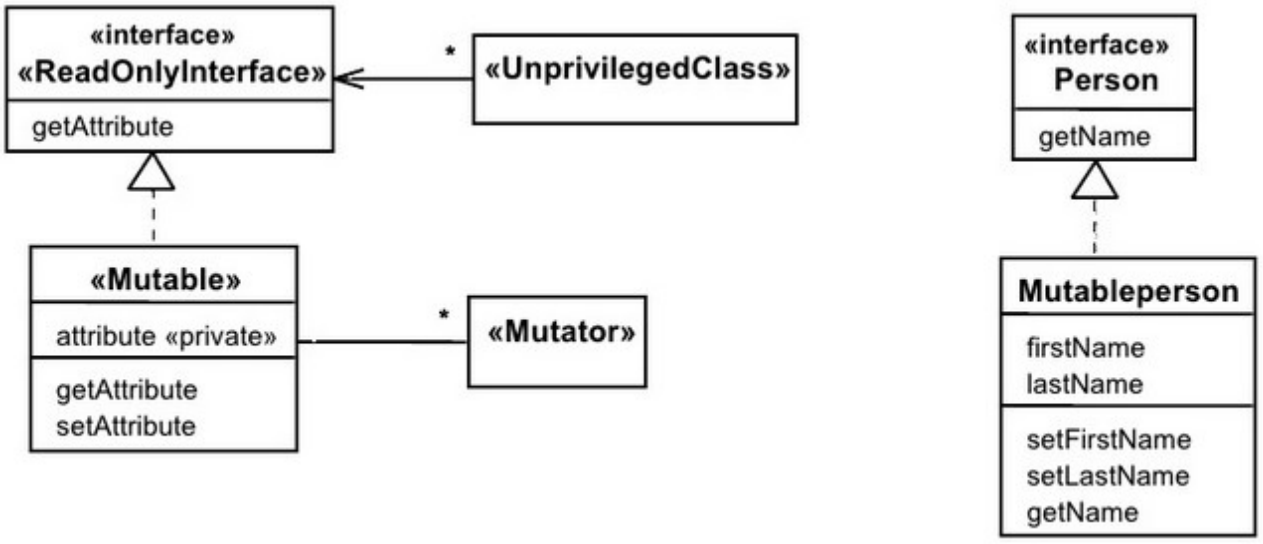
Contexto: Un objeto inmutable es aquel que posee un estado que nunca cambia después de ser creado. Una razón importante para utilizar un objeto inmutable es que otros objetos pueden confiar en que su contenido no cambiará inesperadamente.

Solución: Asegurar que el constructor de la clase inmutable es el único lugar donde los valores de las variables de instancia son asignados o modificados.
Hay que asegurarse, por tanto, que la ejecución de cualquier método de instancia no tenga efectos colaterales derivados de cambiar el valor de variables de instancia o de llamar a métodos que las puedan cambiar.
Un método que necesitara modificar el valor de una variable de instancia de una clase inmutable tendrá que devolver una nueva instancia de esta clase.

Interfaz Sólo-Lectura (Read-Only Interface)

Contexto: A veces, es necesario que sólo determinadas clases privilegiadas sean capaces de modificar los atributos de objetos que de otra manera serán inmutables.

Fuerzas: lenguajes de programación tales como Java permiten controlar los accesos utilizando las palabras reservadas public, protected y private. Sin embargo, la conversión del alcance de una entidad a público ocasiona que esta pueda ser accedida tanto en lectura como en escritura por otras clases.



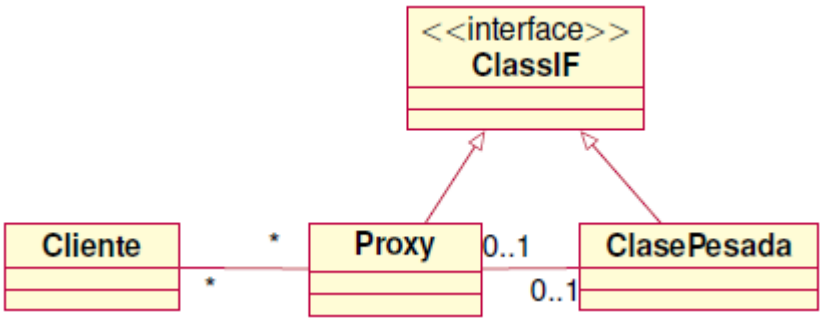
Proxy

Contexto: A menudo, se tarda mucho tiempo y resulta complicado obtener acceso a las instancias de una clase remota dentro de un programa. A estas clases se las suele denominar clases pesadas y suelen estar ubicadas en una base de datos separada de la aplicación.
Los métodos "constructor" deberán cargar previamente, quizás junto con todos los datos de la citada base de datos, las clases pesadas que vayan a utilizar.
De manera análoga, puede que el cargar un objeto pesado, puede hacer más lento el programa. Existe un retardo y un mecanismo complejo involucrado en la creación del objeto en memoria, sin embargo, el sistema software puede incluir objetos que necesiten referirse o utilizar instancias de clases pesadas con rapidez.

Problema: ¿Cómo se puede reducir la necesidad de cargar en memoria un gran número de objetos pesados cuando no se necesitan todos a la vez inmediatamente?

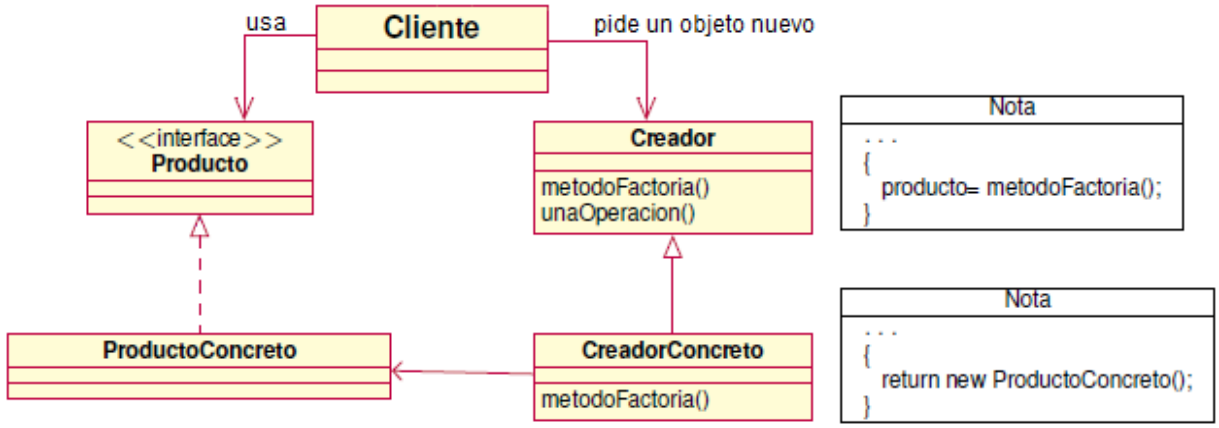
Fuerzas: se necesita que todos los objetos de un modelo de dominio estén disponibles en memoria, pero sólo es imprescindible su carga cuando desempeñan determinadas responsabilidades del sistema software.

Solución: crear una versión más simple de la clase pesada, de esta forma, los programadores pueden declarar variables en sus programas sin preocuparse acerca de si estas se asignan a una instancia de un "Proxy" o de la clase pesada inicial.



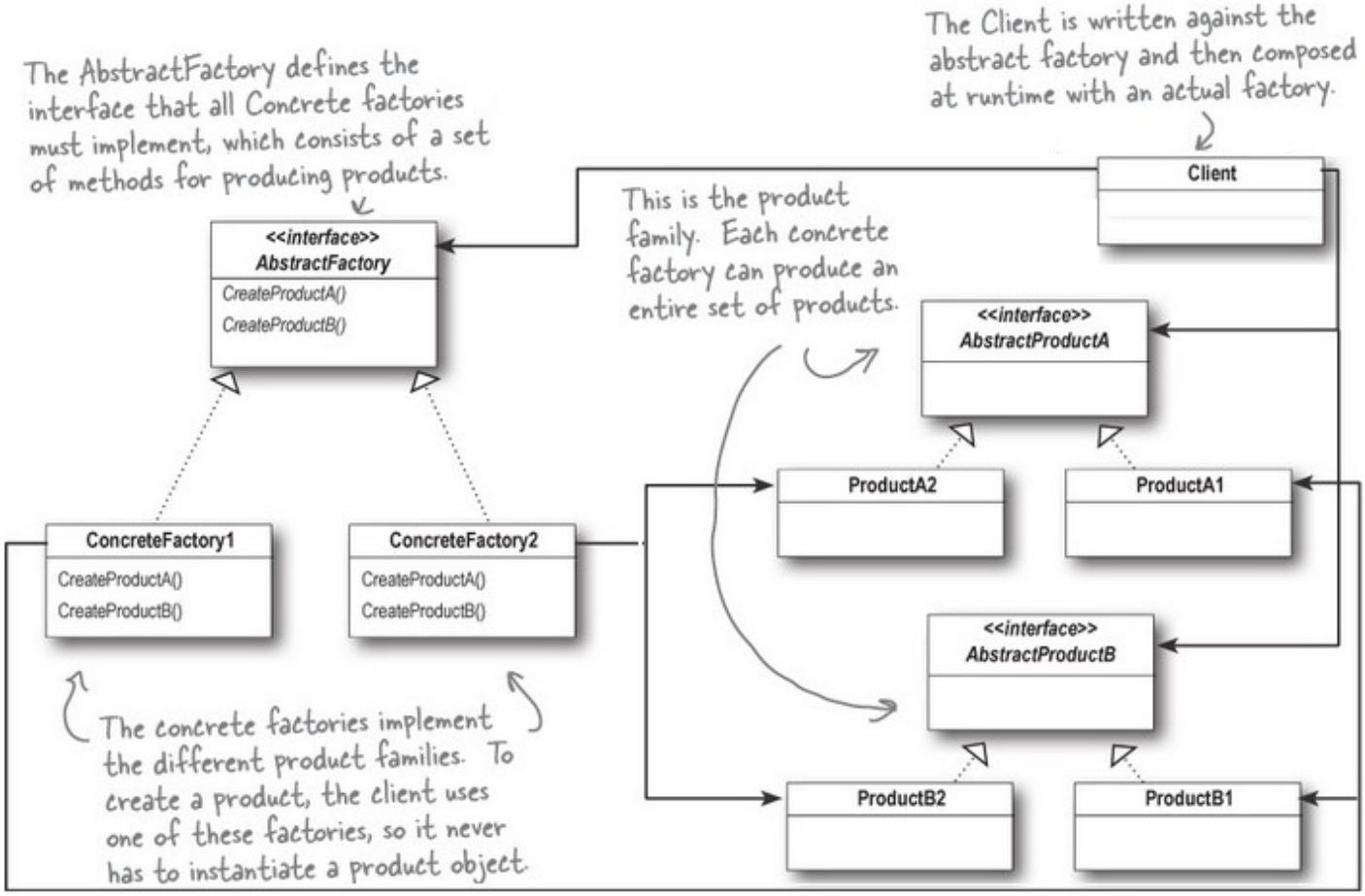
Factoría (Factory Method)

Definición y contexto: Define una interfaz para crear un objeto, pero dejando que sean las subclases quienes decidan qué clase instanciar. Permite así definir la instanciación a las subclases.
Algunos de los procesos que se necesitan en la creación de un objeto incluyen el determinar qué objeto concreto se va a crear, gestionar el tiempo de vida del objeto, y gestionar los asuntos de construcción y destrucción de dicho objeto. El patrón Método Factoría depende del mecanismo de herencia, ya que la creación de objetos se delega a las subclases que implementan el método factoría para crearse a los objetos.



Factoría abstracta (Abstract Factory)

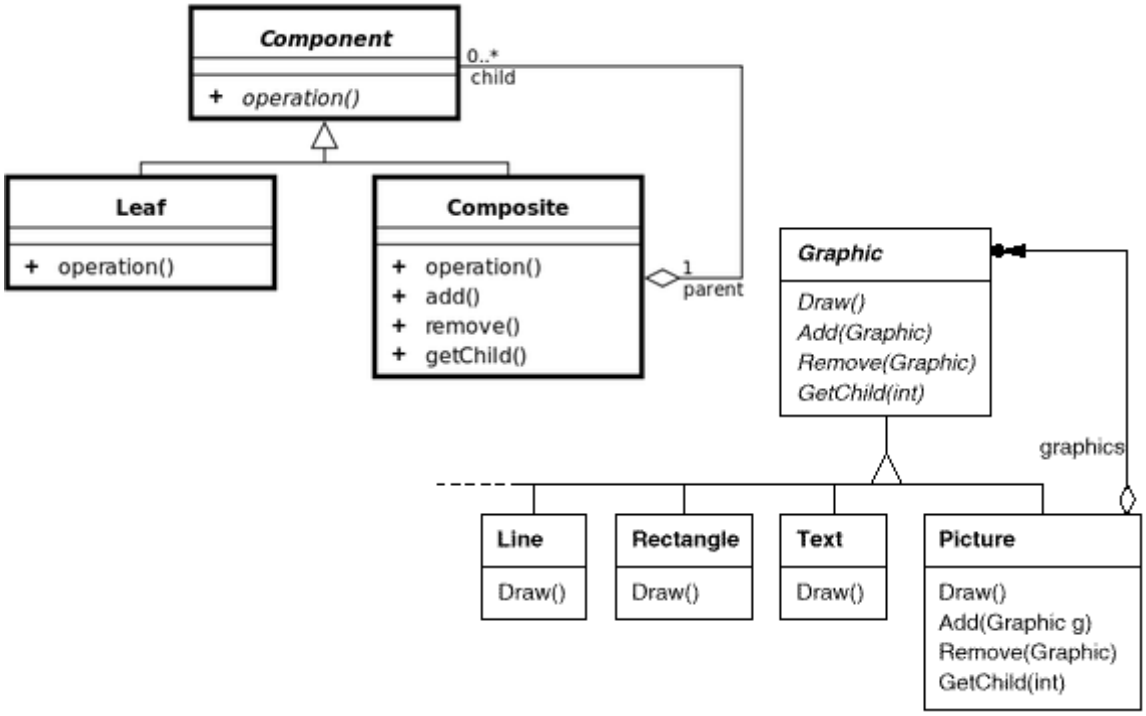
Contexto: Debemos crear diferentes objetos, todos pertenecientes a la misma familia.
El problema que intenta solucionar este patrón es el de crear diferentes familias de objetos.
El patrón factoría abstracta está aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes, puesto que afectaría a todas las familias creadas.



Composite

Definición y contexto: Se trata de un patrón de diseño para el particionamiento. Mediante este patrón, un grupo de objetos es tratado de la misma forma que un solo objeto.
“Componer" objetos en estructuras en forma de árbol abstracto para representar jerarquías todo–parte. Con este tipo de representación las aplicaciones–cliente pueden procesar de manera uniforme los objetos individuales y las composiciones de objetos.

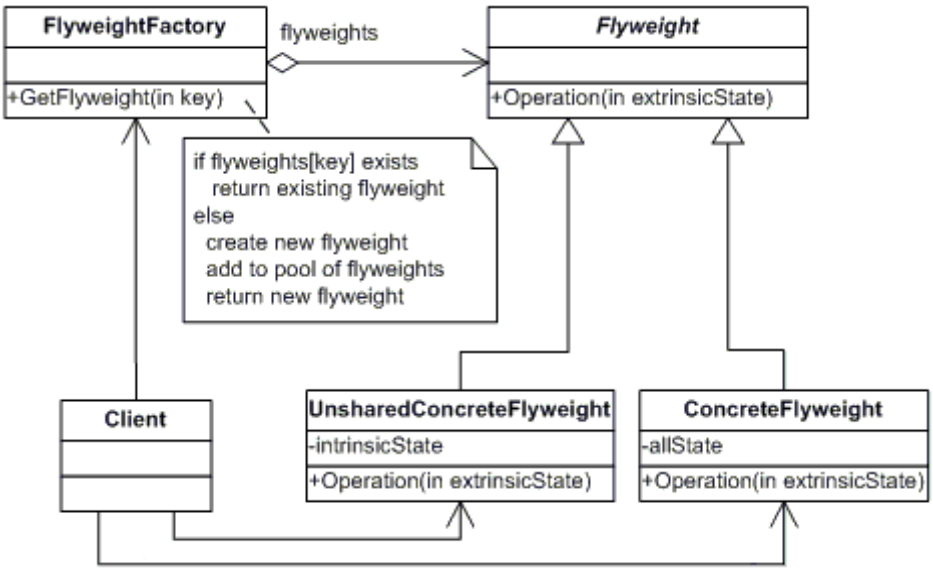
Fuerzas: Se utiliza cuando es necesario componer objetos para formar árboles que representen jerarquías, cuando hay que representar (abstractamente) objetos primitivos (leaf) y sus contenedores. Los contenedores pueden ser agregados de otros contenedores o de objetos hay operaciones del contenedor que aplican a todos los descendientes (children).
Se usa para conseguir que los clientes ignoren la diferencia entre objetos individuales y agregaciones de objetos, para hacer más fácil la inclusión de nuevos componentes y para hacer el diseño más general.



Flyweight (u objeto ligero)

Descripción y contexto: sirve para eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos que contienen información idéntica, además de lograr un equilibrio entre flexibilidad y rendimiento (uso de recursos).
Se usa para ahorrar espacio, mayor cuanto más estado intrínseco (elementos que se puedan compartir o son comunes) de los objetos se pueda compartir.
Se debe crear una fábrica que pueda almacenar y reutilizar las instancias existentes de clases, el cliente debe usar la fábrica en vez de utilizar el operador new si requiere de creación de objetos.

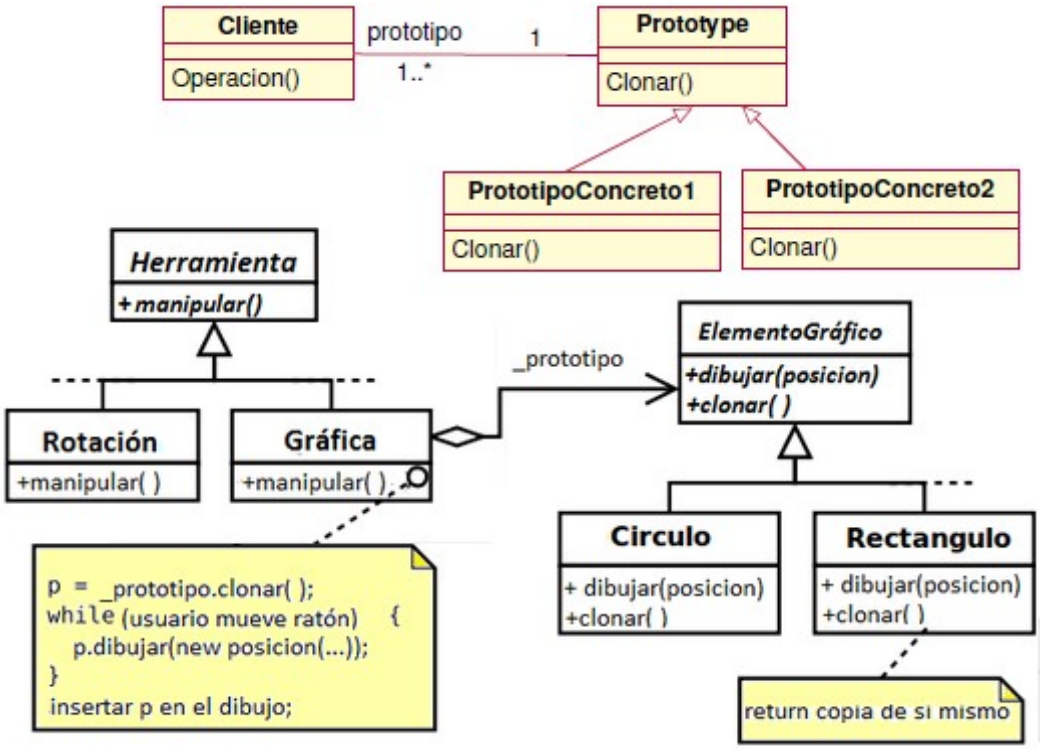
Condiciones para aplicar el patrón:
La aplicación utiliza un número grande de objetos (baja granularidad) con altos costes de almacenamiento debido a la cantidad de objetos.
La mayoría del estado de un objeto se puede convertir en extrínseco (elementos particulares a cada tipo). Después de encapsular el estado extrínseco, muchos grupos de objetos pueden ser reemplazados por un número relativamente pequeño de objetos compartidos.
No se depende de la identidad de los objetos.



Prototype

Descripción y contexto: tiene como finalidad crear nuevos objetos duplicándolos (se crean en tiempo de ejecución), clonando una instancia creada previamente. Se utiliza cuando el sistema ha de ser independiente de cómo se crean, componen y representan sus productos.

La clase de los objetos que servirán de prototipo deberá incluir en su interfaz la manera de solicitar una copia, que será desarrollada luego por las clases concretas de prototipos.
Se pueden añadir y eliminar productos (objetos) en tiempo de ejecución.



Visitante (Visitor)

Descripción y contexto: se utiliza para representar a una operación que ha de ser realizada sobre los elementos de una estructura de objetos.
Las aplicaciones crearán objetos VisitanteConcreto y después atravesarán la estructura de objetos.
Al ser visitados los objetos, el visitante llamará al método adecuado (aceptar) para cada objeto.
Se ha de evitar cambiar las clases de los elementos sobre los que opera.

La idea básica es que se tiene un conjunto de clases elemento que conforman la estructura de un objeto.
Cada una de estas clases elemento tiene un método aceptar (accept()) que recibe al objeto visitante (visitor) como argumento.
El visitante es una interfaz que tiene un método visit diferente para cada clase elemento; por tanto habrá implementaciones de la interfaz visitor de la forma: visitorClase1, visitorClase2 … visitorClaseN.
El método accept de una clase elemento llama al método visit de su clase.
Clases concretas de un visitante pueden entonces ser escritas para hacer una operación en particular.