DISEÑO ORIENTADO A OBJETOS

Prof. Esp. Ing. Agustín Fernandez

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
  - Garantiza que una clase solo tenga una única instancia y proporciona un punto de acceso global a ella.
  - La idea del patrón Singleton es proveer un mecanismo para limitar el número de instancias de una clase. Por lo tanto el mismo objeto es siempre compartido por distintas partes del código.
  - Puede ser visto como una solución más elegante para una variable global ya que una variable global no nos previene de crear múltiples instancias de un objeto.

Patrones de Diseño (creación)

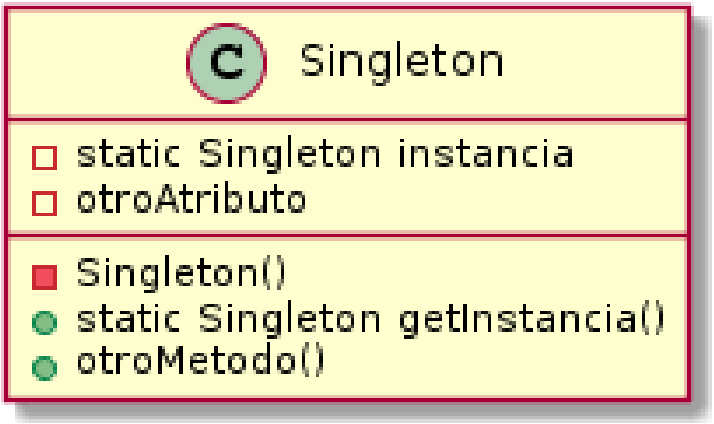
- Patrón Singleton (Instancia única)
- **Debe usarse este patrón cuando:**
  - Debe haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.
  - Se requiere de un acceso estandarizado y conocido públicamente y que no debe ser modificado por los clientes.

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Usos comunes:**
  - Por ejemplo si hay un servidor que necesita ser representado mediante un objeto, este debería ser único (debería existir una sola instancia y el resto de las clases deberían de comunicarse con el mismo servidor).
  - Un calendario, por ejemplo, también es único para todos. (La clase Calendar de Java)
  - No debe utilizarse cuando una clase representa a un objeto que no es único. Ejemplo: la clase Alumno no debería ser Singleton ya que representa a un alumno real y cada alumno tiene su propio nombre, edad, domicilio, DNI, legajo, etc.

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Estructura UML:**



Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Estructura UML (Cont.):**
  - **Singleton**: define una instancia para que los clientes puedan accederla. Esta instancia es accedida mediante un método de clase. Los clientes (quienes quieren acceder a la clase Singleton) acceden a la única instancia mediante un método llamado getInstance().

Patrones de Diseño (creación)

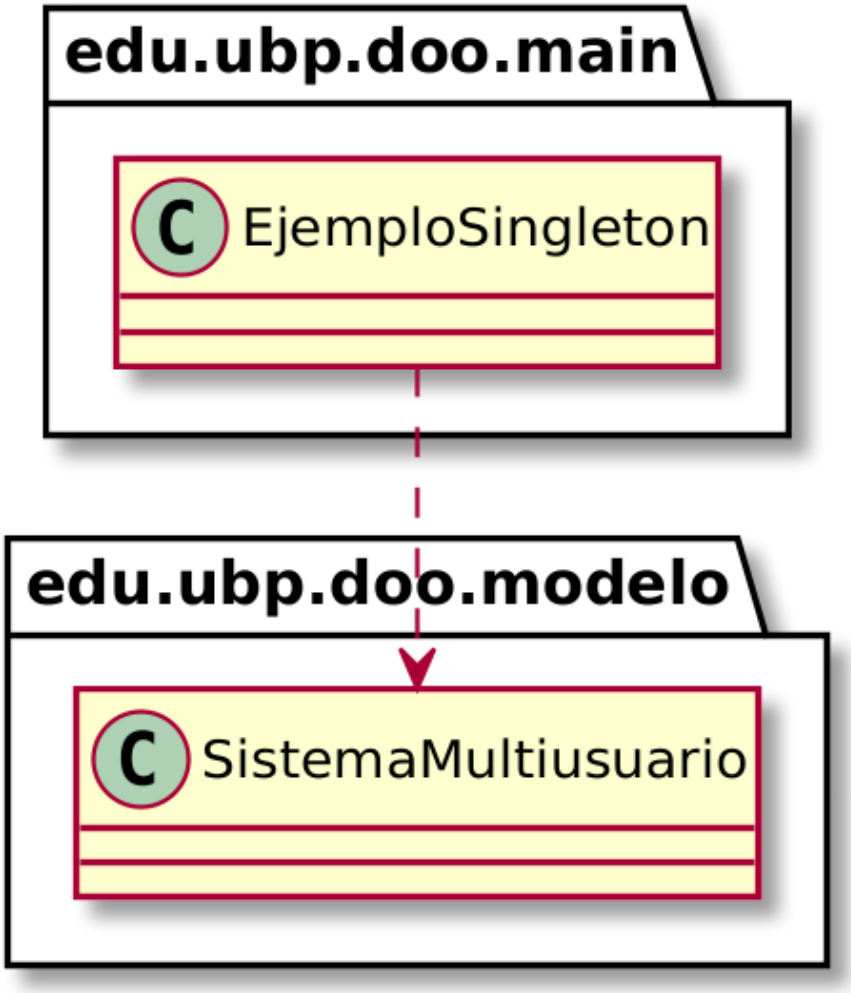
- Patrón Singleton (Instancia única)
- **Implementación:**
  - La propia clase es responsable de crear la única instancia.
  - Permite el acceso global a dicha instancia mediante un método de clase.
  - Declara el constructor de clase como privado para que no sea instanciable directamente.

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Ejemplo:**
  - Supongamos que tenemos un sistema multiusuario en el cual deseamos contar la cantidad de usuarios conectados en un determinado momento.
  - ¿Como sería la solución utilizando este patrón?

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Ejemplo:**



Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Ejercicio:** Según la siguiente implementación de Singleton cual es error qué puede observarse en el fragmento de código:

```
public class SingletonMal {

    private static SingletonMal instancia = null;

    public SingletonMal() {
    }

    /**
     * @return the instancia
     */
    public static SingletonMal getInstancia() {
        if(instancia == null){
            instancia = new SingletonMal();
        }
        return instancia;
    }
}
```

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Cosas a considerar:**
  - El patrón visto hasta aquí esta pensado para una aplicación mono-hilo, por lo que si quiere ser usado en una aplicación multi-hilo debería pensar en sincronizarse la implementación del método getInstancia().
  - Además para asegurar que se cumpla el requerimiento de "única instancia" del singleton la clase debería producir un objeto no clonable. Entonces, se debería impedir la clonación sobrescribiendo el método "clone" en nuestra clase Singleton.
  - Otra buena practica es que los métodos (o la clase) deberían ser declarados como final para forzar a que no puedan ser sobrescritos.

Patrones de Diseño (creación)

- Patrón Singleton (Instancia única)
- **Consecuencias:**
  - Acceso controlado a una única instancia.
  - Espacio de nombres reducidos: el patrón Singleton es una mejora al uso de variables globales. Evitar contaminar el espacio de nombres con el uso de variables globales.
  - Permite un numero variable de instancias: solo se necesitaría cambiar el método que permite acceso a la instancia del Singleton para poder permitir más de una instancia de clase.

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
  - Libera al desarrollador sobre la forma correcta de crear objetos.
  - Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.
  - Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.

Patrones de Diseño (creación)

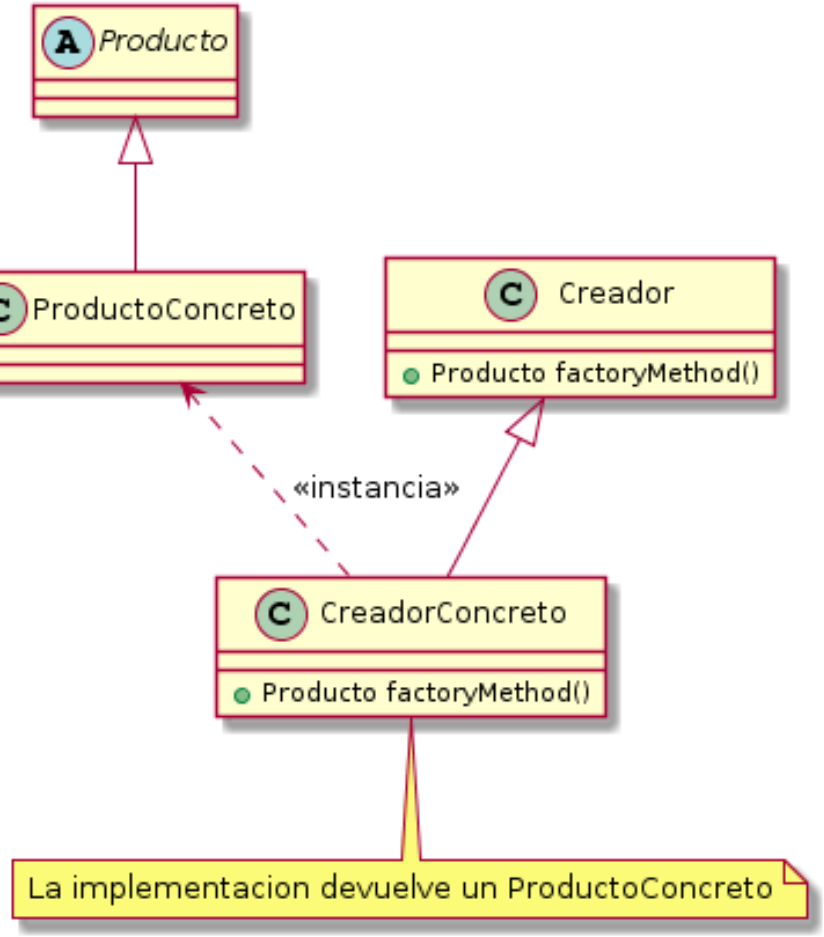
- Patrón Factory Method (Método de Fabricación)
  - Permite un diseño más personalizable y sólo un poco más complicado.
  - Otros patrones de diseño requieren muchas nuevas clases, mientras que éste sólo requiere una nueva operación y unas pocas clases.
  - A menudo se usa éste patrón como la manera estándar de crear objetos.

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Debe usarse este patrón cuando:**
  - Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos.
  - Hay clases que delegan responsabilidades en una o varias subclases.
  - Una aplicación es grande y compleja y posee muchos patrones creacionales.

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Estructura UML:**



Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Estructura UML (Cont.):**
  - **Product:** Define la interfaz de los objetos que de van a crear.
  - **ProductoConcreto (ConcreteProduct)**: es el resultado final. El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.
  - **Creator**: declara la factoría (el método) y devuelve un objeto del tipo producto. Además, puede definir una implementación por defecto para la factoría que devolverá un objeto ConcreteProduct por defecto.
  - **ConcretCreator**: sobrescribe el método de la factoría para devolver una instancia concreta de un objeto ProductoConcreto.

Patrones de Diseño (creación)

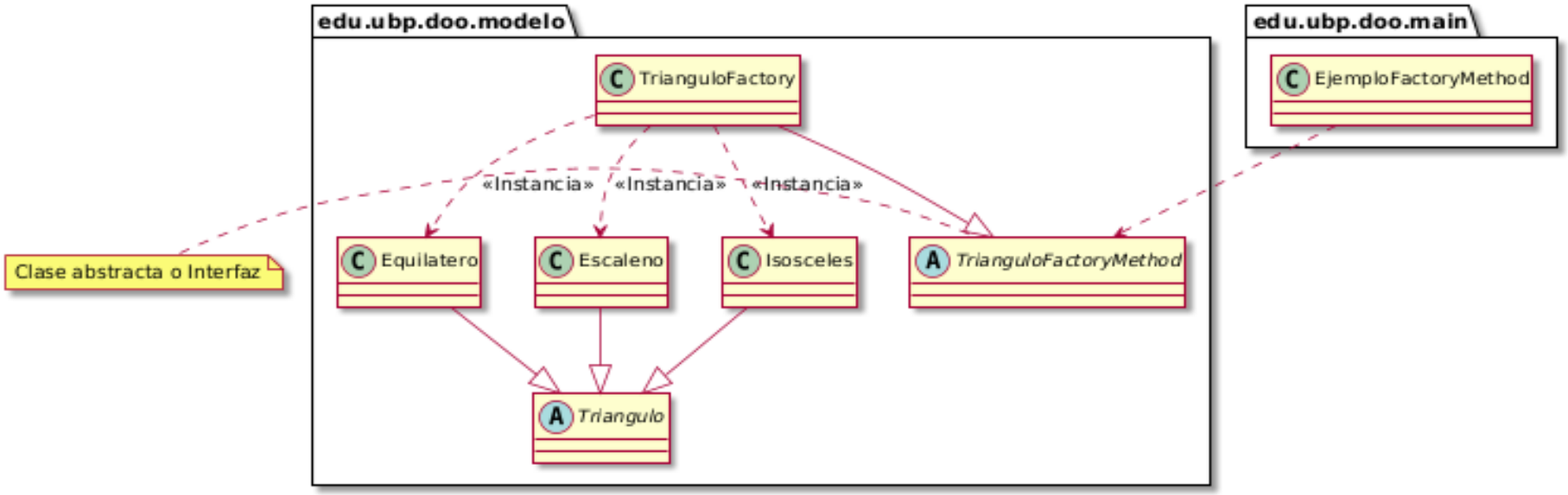
- Patrón Factory Method (Método de Fabricación)
- **Opciones disponibles para la implementación:**
  - El Creator es una clase abstracta y no ofrece ningún tipo de implementación para la factoría que declara.
  - El Creator es una clase concreta y ofrece una implementación por defecto de la factoría.
  - La parametrización de la factoría. Donde la factoría recibe un parámetro con el tipo de objeto que tiene que crear.
  - Todos los objetos disponibles tiene que compartir la interfaz Product.

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Ejemplo:**
  - Supongamos que tenemos una clase abstracta llamada Triangulo, de la cual heredan los 3 tipos de triángulos conocidos. Y si queremos calcular la superficie y determinar el tipo de cada triangulo en tiempo de ejecución.
  - Pero tenemos el siguiente inconveniente: quien se encargue de crear un tipo de triángulo concreto no debería tener que conocer como se compone internamente el mismo.
  - ¿Como hacemos esto utilizando este patrón?

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Ejemplo:**

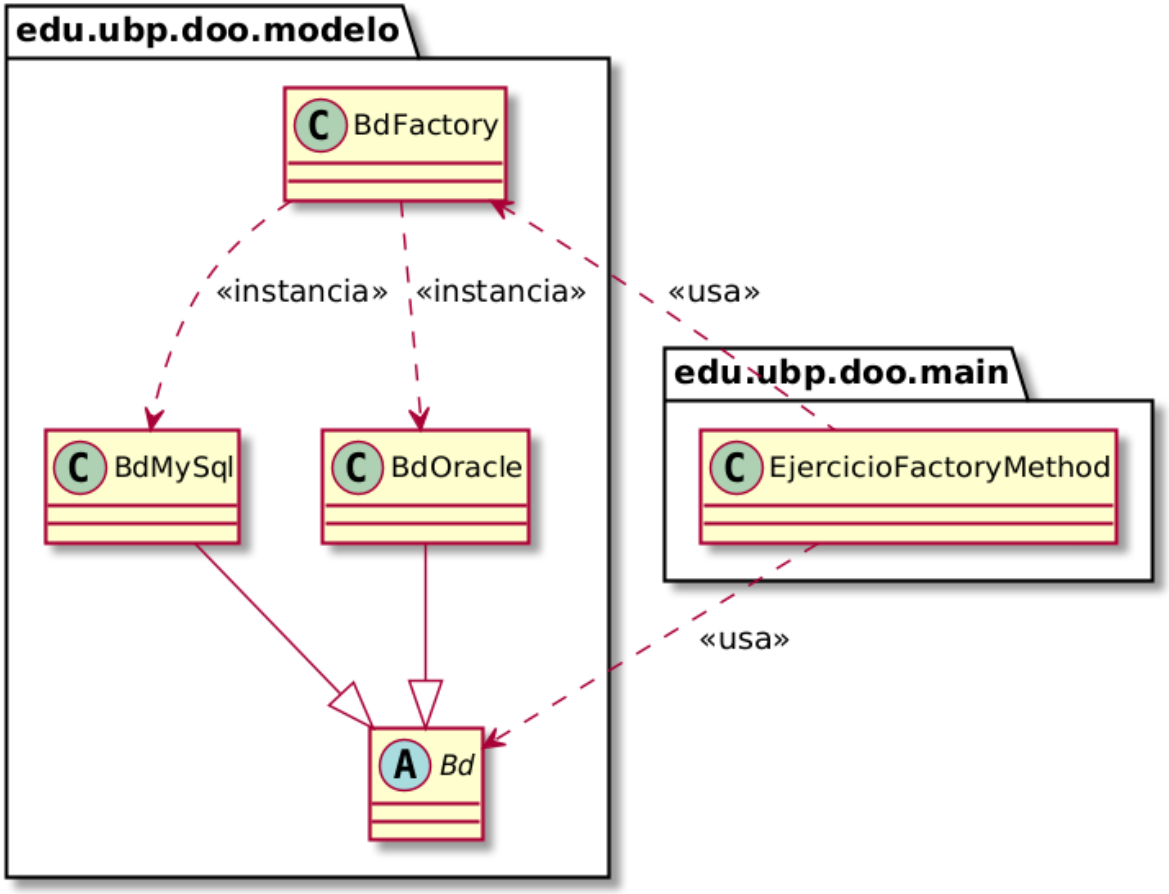


Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Ejercicio:** supongamos que debemos implementar este patrón porque se nos ha solicitado crear las clases necesarias que nos permitan crear las conexiones a dos motores de bases de datos diferentes (Oracle y MySQL) y además que “implementemos” los métodos insert, update y delete para ambos motores. Para nuestro caso y por simplicidad se nos ha pedido, también, que los métodos insert, update y delete no devuelvan nada.
- Utilice las siguientes cadenas de conexión como ejemplo:
  - jdbc:mysql://localhost:3306/miBaseDatos
  - jdbc:oracle:thin:@WIN01:1521:miBaseDatos

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Ejercicio (continua):**



Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Consecuencias:**
  - Como ventaja se destaca que elimina la necesidad de introducir clases específicas en el código del creador.
  - Solo maneja la interfaz Product, por lo que permite añadir cualquier clase ConcretProduct definida por el usuario.
  - Por otro lado, es más flexible crear un objeto con un Factory Method.
  - Un método factoría puede dar una implementación por defecto.

Patrones de Diseño (creación)

- Patrón Factory Method (Método de Fabricación)
- **Para tener en cuenta:**
  - Este patrón es muy versátil.
  - No necesariamente se aplica a una jerarquía de subclases.
  - A una clase que le facilita la creación al cliente también se la suele denominar Factory, aún cuando no se trate de una jerarquía.
  - Entonces podríamos redefinir el término del factory como: **aquella clase que alivia la carga de crear un objeto de manera correcta, cuando el resto de los patrones de creación no aplican.**

Patrones de Diseño (creación)

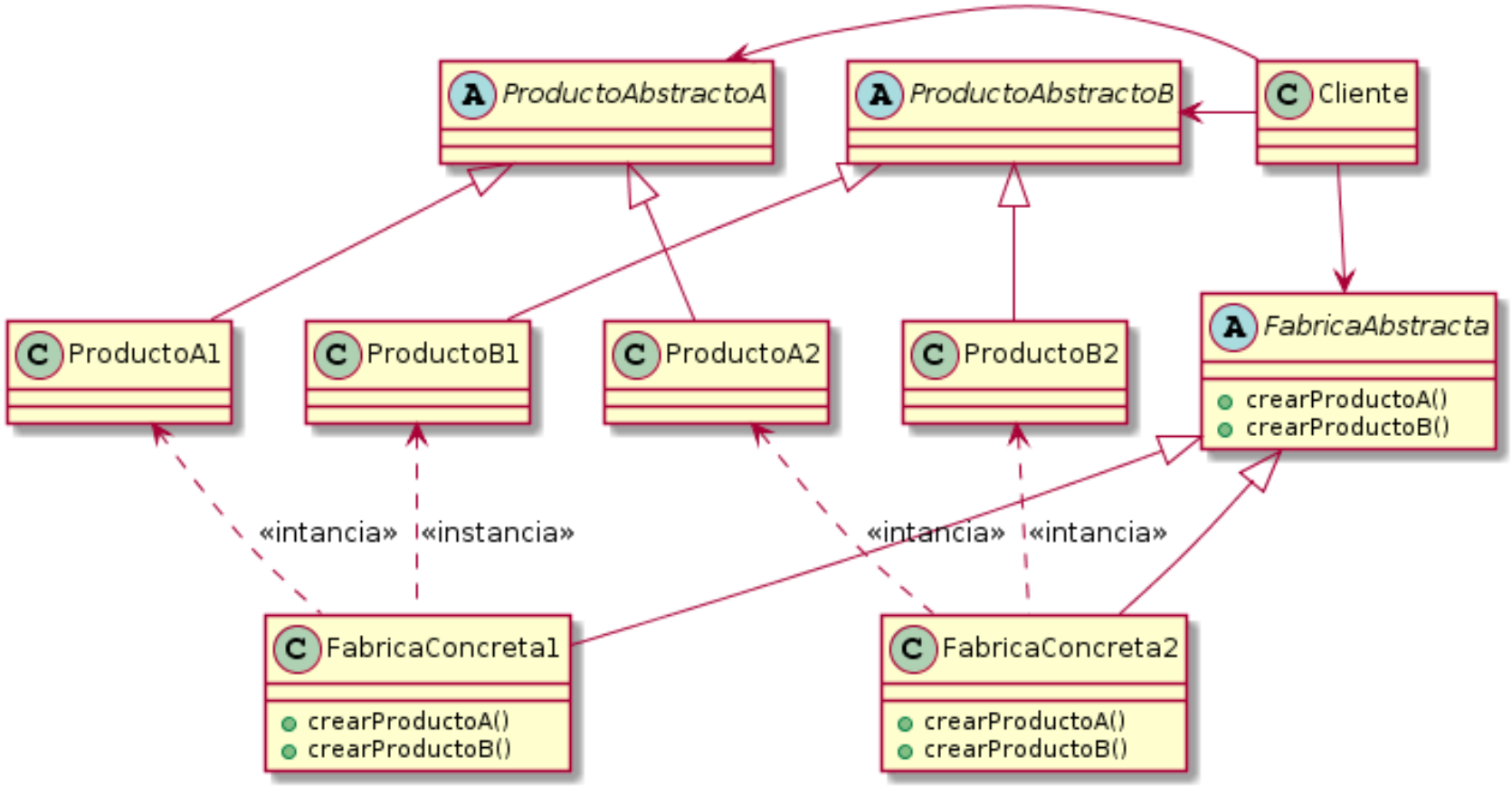
- Patrón Abstract Factory (Fabrica Abstracta)
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
  - Se suele utilizar mucho por ejemplo para la creación de widgets o interfaces gráficas que sean multiplataforma, de esta forma nos podemos abstraer del sistema concreto y realizar el desarrollo de forma genérica.
  - Es similar al Factory Method, sólo que un poco más complejo.
  - Otro nombre que recibe es el de “Kit” [GoF].

Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Debe usarse este patrón cuando:**
  - La creación de objetos debe ser independiente del sistema que los utilice.
  - Los sistemas deben ser capaces de utilizar múltiples familias de objetos.
  - Una familia de productos relacionados están hechos para utilizarse juntos.
  - 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.

Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Estructura UML:**



Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Estructura UML (Cont.):**
  - **Cliente**: La clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría. Lo que quiere es obtener una instancia de alguno de los productos (ProductoA, ProductoB). Sólo usa interfaces declaradas por las clases Fabrica Abstracta y ProductoAbstracto.
  - **Fabrica Abstracta**: Es la definición de la interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear. ("crearProductoA()" y "crearProductoB()")

Patrones de Diseño (creación)

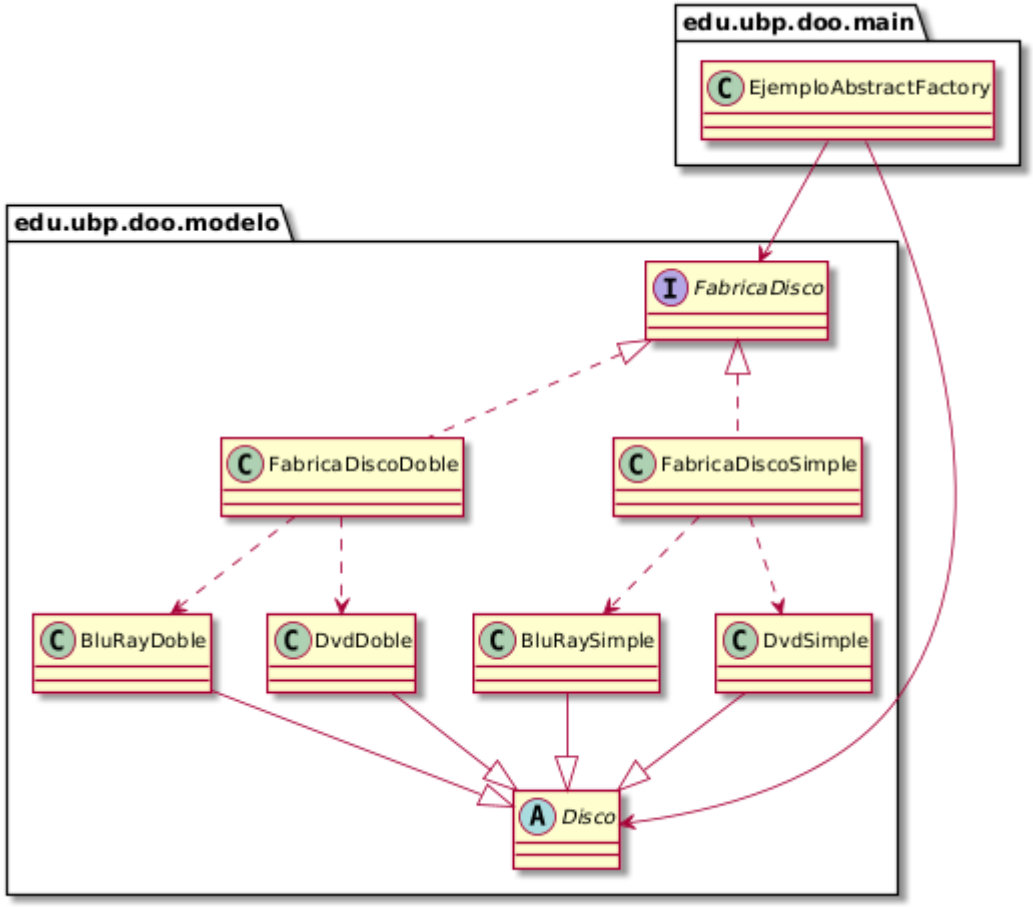
- Patrón Abstract Factory (Fabrica Abstracta)
- **Estructura UML (Cont.):**
  - **Factorías Concretas**: Implementa las operaciones para la creación de objetos de productos concretos.
  - **Producto abstracto**: Definición de las interfaces para la familia de productos genéricos. El cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.
  - **Producto concreto**: Implementación de los diferentes productos que la correspondiente factoría concreta se encargaría de crear.

Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Ejemplo:**
  - Supongamos que tenemos una fabrica de discos (DVD y Blu Ray) y que producimos dos variantes de estos tipos de discos (simple y doble capa) cuyo precio y capacidad de almacenamiento varían según el tipo de disco fabricado.
  - Veamos como sería la implementación de este problema utilizando este patrón.

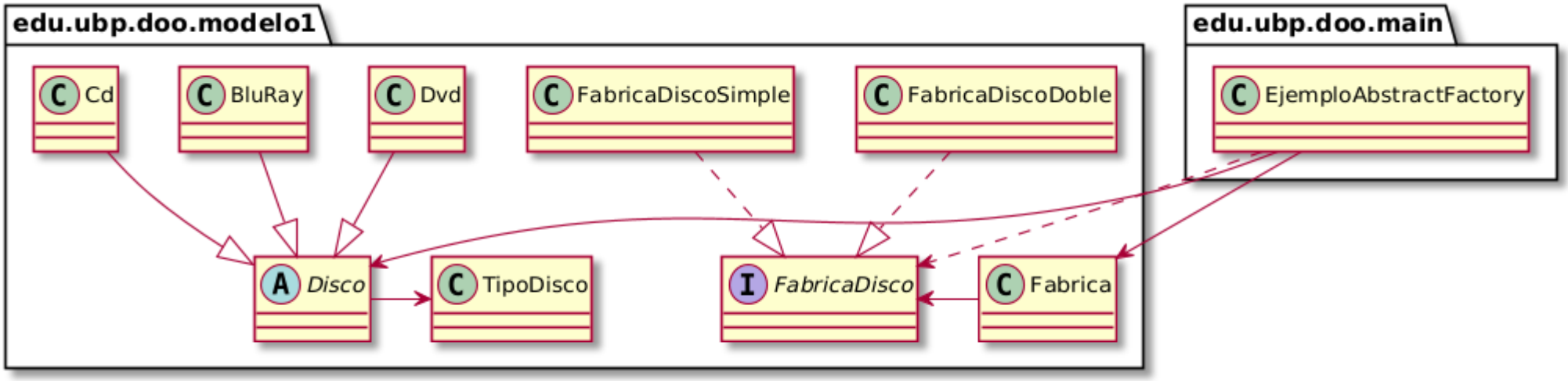
Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Ejemplo: (Con algunas “licencias” en la definición de los Objetos concretos)**



Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Ejemplo: (Con mejoras y un Factory Method)**



Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Ejercicio:** Imaginemos que la empresa de Servicios de televisión por cable “Abstract TV” ofrece 2 productos que sus clientes pueden adquirir:
  - TV por cable.
  - Internet.
- Dentro del producto de **TV por cable** los clientes pueden adquirir el Plan Simple (solo 50 canales) o el Plan Premium (250 canales). Por supuesto cada uno de estos planes tienen precios diferentes:
  - $120 mensuales para el Plan Simple.
  - $300 mensuales para el Plan Premium.
- Además dentro del producto **Internet** los clientes pueden adquirir el Plan Simple (de 3 Mbytes de ancho de banda) o el Plan Premium (de 7 Mbytes de ancho de banda). Por supuesto cada uno de estos planes tienen precios diferentes:
  - $100 mensuales para el Plan Simple.
  - $450 mensuales para el Plan Premium.
- ¿Como desarrollarían la solución aplicando este patrón?

Patrones de Diseño (creación)

- Patrón Abstract Factory (Fabrica Abstracta)
- **Consecuencias:**
  - Se oculta a los clientes las clases de implementación: los clientes manipulan los objetos a través de las interfaces o clases abstractas.
  - Facilita el intercambio de familias de productos: al crear una familia completa de objetos con una factoría abstracta, es fácil cambiar toda la familia de una vez simplemente cambiando la factoría concreta.
  - Mejora la consistencia entre productos: el uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia.
  - Como inconveniente nuevamente podemos decir que no siempre es fácil soportar nuevos tipos de productos si se tiene que extender la interfaz de la Factoría abstracta.