Clase 11 - DA2 Teórico

25/10/2021

Hoy

- Segunda prueba de actuación en clase
- Patrones
  - Factory method
  - Abstract Factory
  - Visitor
  - Observer

Reglas de prueba de actuación:

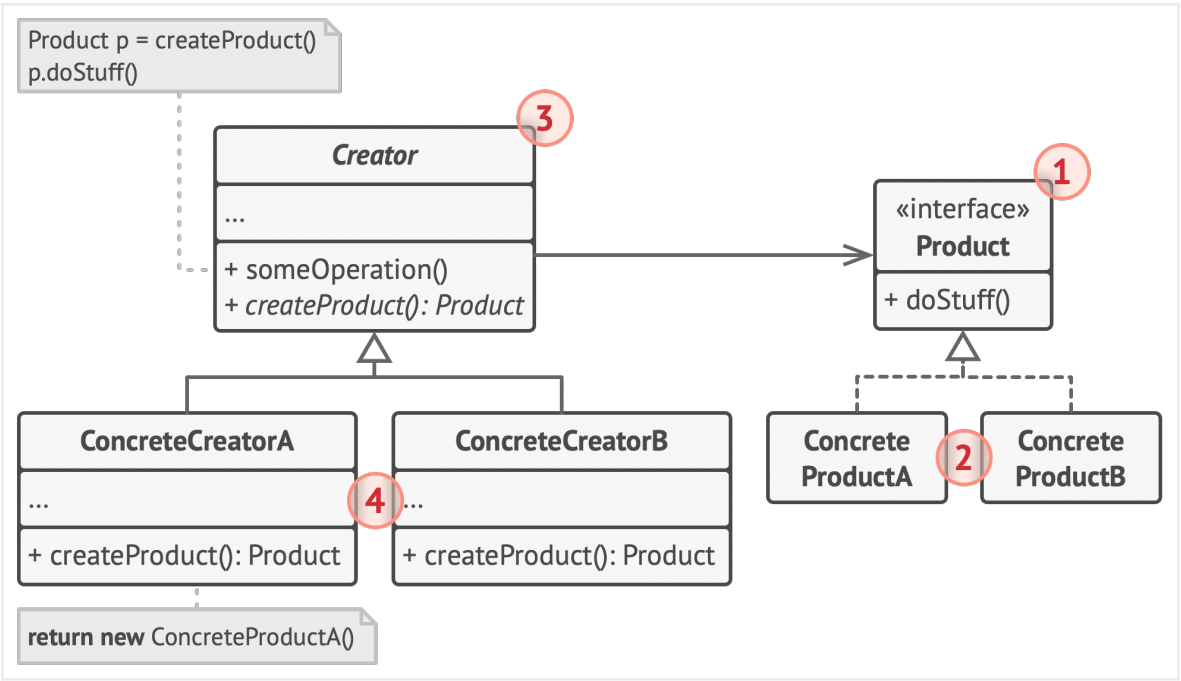
- 20 minutos
- No pueden volver para atrás
- 2 Preguntas multiple opción (única opción correcta)

Factory Method

Clasificación: Creacional

Intención: Proveer una interfaz para crear objetos que heredan de una cierta clase padre, permitiendo que las subclases decidan el tipo de objetos a crear.

Estructura:



1. Clase abstracta/interface: Usuario(declara la interfaz Producto que tendrá que ser cumplida por todos los objetos que queremos crear de la jearquía)
2. Clases: Tester, Desarrollador y Administrador (son los objetos concretos que implementan la interfaz Producto)
3. Clase UsuarioFactory (clase abstracta/inteface que declara el factory method que devuelve nuevos tipos de productos, en este caso devuelve los tipos de Usuario
4. Clases: DesarrolladorFactory, TesterFactory, AdministradorFactory. (las fabricas concretas que devuelven cada tipo de producto en concreto. Tenemos una factory por tipo de producto que queremos construir).

Aplicabilidad:

- Cuando una clase no puede anticipar que tipo de objetos debe crear.
- Una clase desea que sus subclases especifiquen los objetos que crean.
- Para no tener que acoplarme a las clases concretas (al crearlas), el patrón delega la responsabilidad de creación del objeto a quien sepa crearlos, que es cada implementación particular del creador (**Creator**). Cambia el foco de creación del objeto, pasando a ser un problema de cada clase

concreta.
- Factory Method elimina la necesidad de que se tenga que conocer clases específicas: el código solo trabaja con la interfaz / clase abstracta del producto, pudiendo trabajar con cualquier ProductoConcreto que el usuario haya definido.

Ventajas:
- Tengo el new en un solo lado.
- Tengo control sobre la creación del objeto, tal vez lo pueda crear una vez y luego cachearlo.
- A diferencia de un constructor, los factory methods pueden tener nombres diferentes y más descriptivos (e.g. **Color.MakeRGBColor(float red, float green, float blue) vs Color = new Color(r, g, b);**
- El cliente esta totalmente desacoplado de los detalles de implementación de las clases derivadas. Nuestro código solo maneja la interfaz “Product” y la creación se realiza de forma polimórfica.

Desventajas:

- Nuestro sistema se vuelve más complejo, tenemos que generar siempre una subclase del “Creator” (es decir, que crearme factories concretas siempre) para poder crear un tipo de “Product” nuevo. Es decir. so quiero crearme un ConcreteProductC tengo que crearme un ConcreteCreatorC que permite crear el objeto en cuestión.

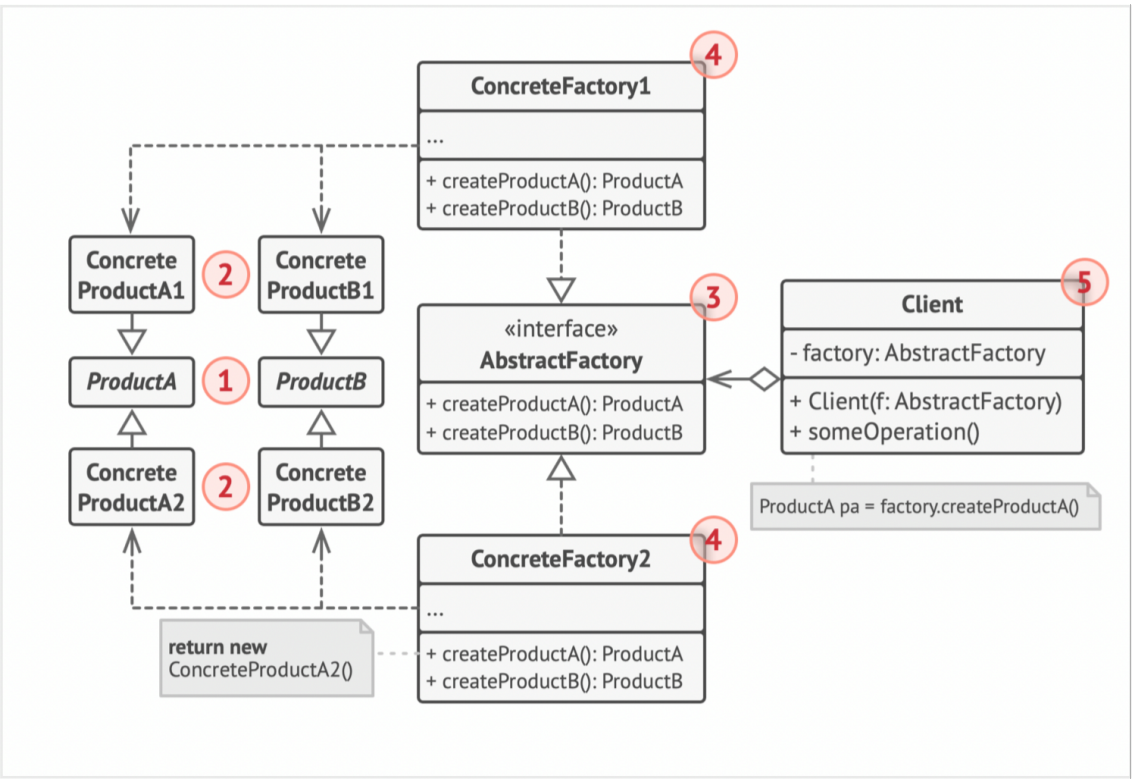
Abstract Factory

Clasificación: Creacional

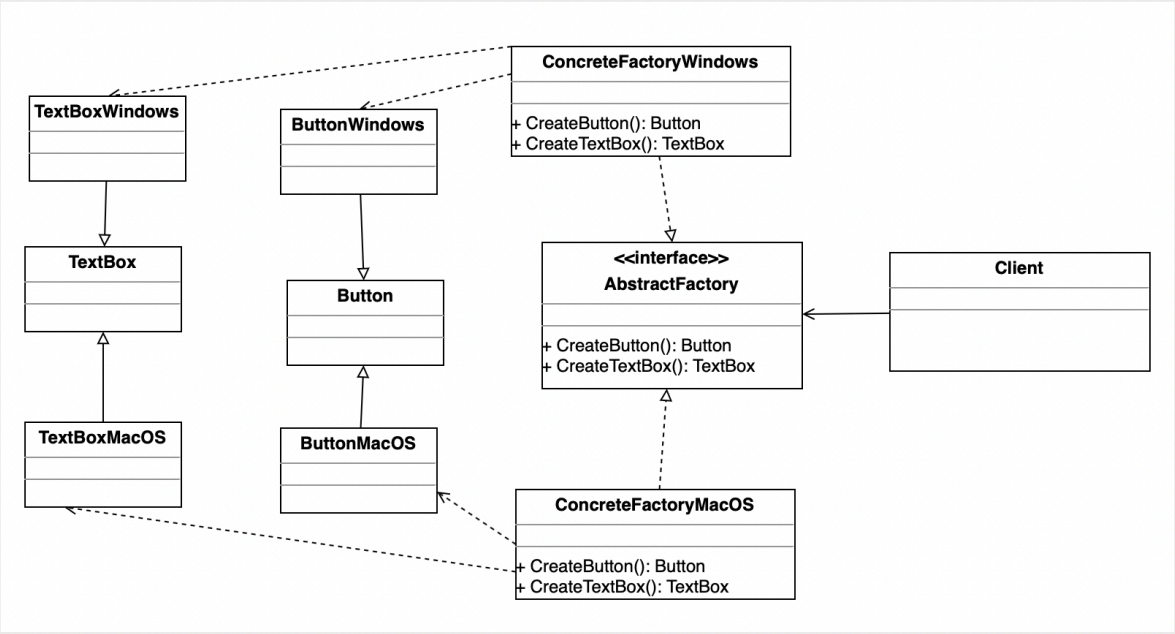
Intención: Permitir crear familias de objetos relacionados sin tener que especificar clases concretas. Lo usamos cuando tenemos varios objetos relacionados entre si que pueden ser agregados o modificados en runtime.

Definición del GOF: *"Provee una forma de encapsular un grupo común de factories **que tienen un tema concreto** sin tener que especificar sus clases concretas". Provee una interfaz para crear familias de objetos dependientes sin tener que especificar sus clases concretas.*

Estructura:



Estructura aplicada



1. Productos abstractos: Tipos de productos a construir, estos son abstractos/interfaces. En nuestro ejemplo son el Button y el Textbox.
2. Productos concretos: Son las diferentes implementaciones de nuestros productos abstractos/interfaces/ Cada producto abstracto, tiene que ser implementado para todas las familias.
3. Abstract Factory: interfaz/abstracta que declara un conjunto de metodos, uno por ada uno de los producots abstractos a construir.
4. ConcreteFactory1 y ConcreteFactory2: son las fabricas concretas que implementan los métodos para crear los objetos de cada producto abstracto, pero siempre asegurandonos que sean instancias de una familia particular. Voy a tener tantas Factories concretas como familias de productos tenga.

Aplicabilidad

- Cuando un sistema debe ser independiente de como sus objetos son creados
- Cuando quiero esconder la implementación de cada familia de productos desacoplando la implementación de cada operación.
- Una familia esta diseñada para funcionar junta y hay que

asegurar que esto sea así, tenemos diferentes familias de objetos/productos.

Ventajas:

- Aísla las clases de implementación: ayuda a controlar que los objetos se creen y encapsula la responsabilidad y el proceso de creación de objetos producto.
- Hace fácil el intercambio de familias de productos. Solo necesitaremos cambiar de factory.
- Fomenta la consistencia entre productos.

Desventajas:

- Para añadir un nuevo producto, se requiere la implementación de la interfaz y todos sus metodos.

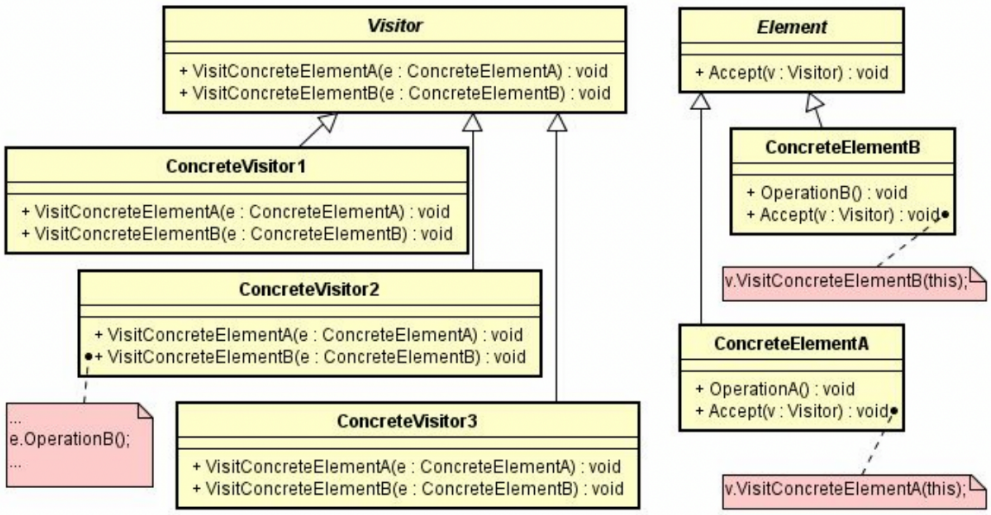
Visitor

Clasificación: Comportamiento

Intención: Representar un método (una operación) que necesitemos aplicar sobre los elementos de una jerarquía de objetos, permitiendo a su vez deifinir nuevos elemtnos sin cambiar las clases que estan en la estructura.

Queremos agregar nuevas operaciones a una jerarquía sin tocar la misma.

Estructura



Cada visitor concreto es una operación. Se utiliza:

```
Element anElement = new ConcreteElementB();
Visitor anOperation = new ConcreteVisitor1();
anElement.Accept(anOperation);
```

Ventajas:

- Es fácil añadir nuevas operaciones a la jerarquía, solo hay que crear un nuevo “Visitante” en vez de cambiar todas la clases agregandole la operación que cumple este visitante.

Desventajas:

- Añadir un nuevo elemento a la jerarquía implica un cambio o impacto en muchas clases, tantas como “visitantes” tengamos.