PATRONES DE DISEÑO

FACTORY METHOD

Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Problema:

Imagina que estás creando una aplicación de gestión Rappi. La primera versión de tu aplicación sólo es capaz de manejar la mensajería de Mercados, por lo que la mayor parte de tu código se encuentra dentro de la clase Supermercados

Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de mensajería domicilios para que incorpores la logística por domicilios a la aplicación.

Estupendo, ¿verdad? Pero, ¿qué pasa con el código? En este momento, la mayor parte de tu código está acoplado a la clase Mercados. Para añadir Domicilios a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde decides añadir otro tipo de Mensajería a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

Al final acabarás con un código bastante sucio, plagado de condicionales que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos de transporte.

Solución:

El patrón Factory Method sugiere que, en lugar de llamar al operador Nuevo para construir objetos directamente, se invoque a un método *Pedido* especial. No te preocupes: los objetos se siguen creando a través del operador Nuevo, pero se invocan desde el método Pedido. Los objetos devueltos por el método Pedido a menudo se denominan *Servicios*.

A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método Pedido en una subclase y cambiar la clase de los servicios creados por el método.

No obstante, hay una pequeña limitación: las subclases sólo pueden devolver servicios de distintos tipos si dichos servicios tienen una clase base o interfaz común.

Además, el método Pedido en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

Por ejemplo, tanto la clase Mercado como la clase Domicilios deben implementar la interfaz Pedido, que declara un método llamado Entrega. Cada clase implementa este método de forma diferente: los Mercados entregan varios productos, mientras que los Domicilios entregan un producto. El método Pedido dentro de la clase PedidoMercado devuelve objetos de Mercado, mientras que el método Pedido de la clase PedidoDomicilios devuelve Domicilios.

Problema:

Imagina que estás creando una aplicación de gestión Uber. La primera versión de tu aplicación sólo es capaz de manejar vehículos Particulares, por lo que la mayor parte de tu código se encuentra dentro de la clase Particular

Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de Taxis para que incorpores la flota por taxis a la aplicación.

Estupendo, ¿verdad? Pero, ¿qué pasa con el código? En este momento, la mayor parte de tu código está acoplado a la clase Particulares. Para añadir Taxis a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde decides añadir otro tipo de vehículo a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

Al final acabarás con un código bastante sucio, plagado de condicionales que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos de transporte.

Solución:

El patrón Factory Method sugiere que, en lugar de llamar al operador Nuevo para construir objetos directamente, se invoque a un método *Vehículo*. No te preocupes: los objetos se siguen creando a través del operador Nuevo, pero se invocan desde el método Vehículo. Los objetos devueltos por el método Vehículo a menudo se denominan *Pedidos*.

A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método Vehículo en una subclase y cambiar la clase de los pedidos creados por el método.

No obstante, hay una pequeña limitación: las subclases sólo pueden devolver pedidos de distintos tipos si dichos pedidos tienen una clase base o interfaz común. Además, el método Vehículo en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

Por ejemplo, tanto la clase Particulares como la clase Taxis deben implementar la interfaz Vehículo, que declara un método llamado Vehículo. Cada clase implementa este método de forma diferente: los Particulares pedidos privados, mientras que los Taxis pedidos públicos. El método Vehículo dentro de la clase VehiculoParticular devuelve carros particulares, mientras que el método Vehículo de la clase VehiculoTaxis devuelve carros publicos.

PROXY

Proxy es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

Problema:

¿Por qué querrías controlar el acceso a un objeto? Imagina que tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas de vez en cuando, pero no siempre.

Puedes llevar a cabo una implementación diferida, es decir, crear este objeto sólo cuando sea realmente necesario. Todos los clientes del objeto tendrán que ejecutar algún código de inicialización diferida. Lamentablemente, esto seguramente generará una gran cantidad de código duplicado.

En un mundo ideal, queríamos meter este código directamente dentro de la clase de nuestro objeto, pero eso no siempre es posible. Por ejemplo, la clase puede ser parte de una biblioteca cerrada de un tercero.

Solución:

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

Pero, ¿cuál es la ventaja? Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy

implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real.

1. Una consulta en Datacrédito o Centrales de Riesgo es un proxy de una entidad comercial, que, a su vez, es un proxy de aprobaciones de créditos. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar un Crédito.
2. Una consulta en Simit es un proxy del tránsito, que, a su vez, es un proxy de aprobaciones de deudas en el ministerio de transporte. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar un traspaso o cualquier vuelta en un organismo de tránsito.
3. Una consulta en la procuraduría, policía, es un proxy de algunos puestos laborales públicos, que, a su vez, es un proxy de aprobaciones de personal apto para puestos públicos. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar contratación de personas, aunque ya no está permitido en muchas empresas pedir estas documentaciones.