PROGRAMACIÓN DE APLICACIONES MÓVILES NATIVAS

Semana 7: Patrones de Diseño – Factory Method

Realizado por: Daniel Betancor Zamora

Curso 2023-2024

Contenido

1. Introducción

A lo largo de este informe se abordarán diferentes conceptos relacionados con el uso de los patrones de diseño, específicamente, con el patrón de diseño ***Factory Method***. Para ello, se diseñará una estructura de clases para una aplicación de notas en Kotlin utilizando dicho patrón de diseño y, además, se plantearán distintos tipos de escenarios posibles en el futuro sobre los cuales se deberá reflexionar.

Los patrones de diseño son herramientas fundamentales que permiten abordar problemas comunes de diseño de manera elegante y efectiva. Este tipo de patrones son soluciones probadas y documentadas para desafíos recurrentes en el desarrollo de software, y ofrecen un enfoque sistemático para abordar problemas específicos.

Uno de los patrones de diseño más influyentes y versátiles es el *Factory Method*, que pertenece a la categoría de patrones de creación. El *Factory Method* resuelve el problema de crear objetos sin especificar explícitamente su clase concreta. A través de este informe, exploraremos a fondo este patrón, su funcionamiento, y los beneficios que aporta a la ingeniería de software.

2. Diseño de la estructura de clases

En este apartado, se planteará un diseño básico para la estructura de clases de la aplicación de notas. Es decir, se expondrán las diferentes clases identificadas y necesarias para el desarrollo de la aplicación, aplicando, por supuesto, el patrón *Factory Method*. Además, se hará una pequeña descripción de cada una de las clases, logrando entender de esta manera qué función cumple cada una de ellas y qué papel tienen con respecto al patrón de diseño usado.

Para todas las clases, se adjuntarán imágenes de como sería el código para cada una de ellas, sin entrar en muchos detalles de la implementación. Cada uno de los códigos que se incluyan en este informe serán muy sencillos y representativos, pues la idea es que se entienda de qué manera se ve reflejado el *Factory Method* en la implementación de las clases.

Por último, también se mostrará un pequeño diagrama de clases, para que se vea de una forma más visual la estructura de la aplicación, por qué clases está formada, de qué manera interactúan entre ellas, y en qué partes de la aplicación se aplica el *Factory Method*.

2.1. Descripción y código de las clases

Antes de comenzar con la presentación de las distintas clases, se deberá comprender que elementos o componentes son claves para el correcto funcionamiento del *Factory Method*:

- ***Product***: Este es el tipo de objeto que se va a crear. Puede ser una clase abstracta o una interfaz que define las operaciones que todos los productos deben implementar.
- ***Creator***: Esta es una clase (también puede ser una interfaz) que contiene un método abstracto llamado "*factoryMethod*" que devuelve un objeto del tipo "*Product*". Las subclases del *Creator* implementan este método para crear instancias concretas de productos. El *Creator* puede tener otros métodos que utilizan el factory method para crear objetos.
- ***ConcreteProduct***: Estas son las clases concretas que implementan la interfaz del Producto. Cada *ConcreteProduct* proporciona una implementación específica de los métodos definidos en el Producto.
- ***ConcreteCreator***: Estas son las subclases del *Creator* que implementan el *factory method* para crear objetos concretos del *Product*. Cada *ConcreteCreator* puede producir un tipo específico de *ConcreteProduct*.

Partiendo de estos elementos, los cuales componen el patrón de diseño que va a ser usado, podemos ir identificando las diferentes clases necesarias para el desarrollo de la aplicación de notas.

Nota (*Product*)

El *Product* en este contexto es la interfaz ***Nota***, que define la operación común que todas las notas deben proporcionar. En este caso, la operación común es obtener el contenido de la nota, que es un texto. La interfaz Nota se define de la siguiente manera:

```
interface Nota {
    fun contenido(): String
}
```

NotaTexto (*ConcreteProduct*)

Este es un *ConcreteProduct* que implementa la interfaz *Nota*. En este caso, la clase ***NotaTexto*** representa una nota de texto y proporciona una implementación concreta del método "*contenido()*":

```
class NotaTexto(private val texto: String) : Nota {
    override fun contenido(): String {
        return texto
    }
}
```

FabricaNotas (*Creator*)

El *Creator* será la interfaz ***FabricaNotas*** que declara un método abstracto "*crearNota()*", que se utiliza para crear objetos de tipo Nota. La interfaz del *Creator* permite desacoplar la creación de objetos de su uso. Aquí se define la interfaz *FabricaNotas*:

```
interface FabricaNotas {
    fun crearNota(texto: String): Nota
}
```

FabricaNotasTexto (*ConcreteCreator*)

La clase ***FabricaNotasTexto*** es una implementación concreta del *Creator* que se encarga de crear notas de texto. Su método "*crearNota()*" toma un texto como argumento y devuelve un objeto de tipo *NotaTexto*:

```
class FabricaNotasTexto : FabricaNotas {
    override fun crearNota(texto: String): Nota {
        return NotaTexto(texto)
    }
}
```

Aplicación del *Factory Method*

El patrón *Factory Method* se aplica en esta estructura de clases para permitir la creación de objetos *Nota* de manera flexible y desacoplada. A través de este patrón, la creación de objetos se delega a las clases concretas de fábrica, como *FabricaNotasTexto*, sin que el cliente necesite conocer los detalles de cómo se crean las notas.

En el *main* o en cualquier otro lugar de la aplicación, puede utilizarse la fábrica de notas de texto de la siguiente manera:

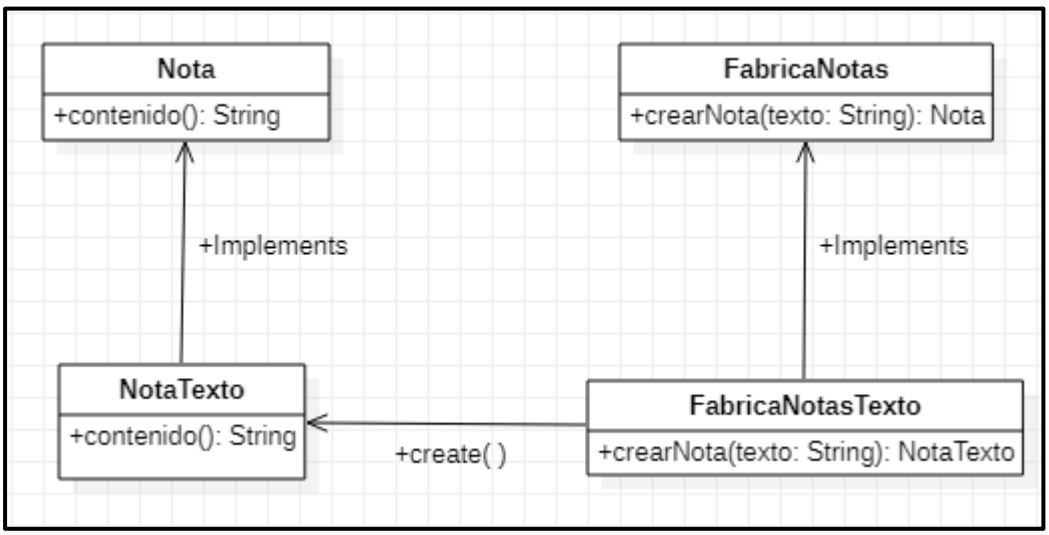
```
fun main() {
    val fabrica = FabricaNotasTexto()

    println("Introduce el contenido de la nota de texto: ")
    val texto = readLine() ?: ""

    val nota = fabrica.crearNota(texto)
    println("Contenido de la nota de texto: ${nota.contenido()}")
}
```

El patrón *Factory Method* permite que la aplicación sea extensible, ya que pueden agregarse nuevas implementaciones de fábricas para crear diferentes tipos de notas (por ejemplo, notas de imagen y notas de audio) sin afectar el código existente. Esto promueve la flexibilidad y el mantenimiento en el desarrollo de la aplicación de notas.

2.2. Diagrama de clases



Este diagrama ilustra cómo el patrón *Factory Method* permite la creación de objetos *Nota* sin que el cliente conozca los detalles de implementación de las clases concretas. Además, muestra cómo la creación de notas de texto se ha encapsulado en *FabricaNotasTexto*, lo que facilita la expansión de la aplicación para incluir otros tipos de notas (como notas de imagen y audio) sin afectar al código existente.

3. Reflexión sobre cambios futuros

En este último apartado se responderán varias preguntas relacionadas con el futuro de la aplicación, proponiendo diferentes escenarios que se podrían llegar a materializar y de qué manera podríamos afrontarlos.

Las preguntas, concretamente, son las siguientes:

- ¿Qué pasa si en un futuro se quisiera añadir un nuevo tipo de notas?
- ¿Qué partes de tu aplicación tendrías que modificar?
- ¿Qué nuevas clases tendrías que añadir?

Debido a que las tres preguntas están muy relacionadas, ya que la respuesta a la primera pregunta implica la inclusión de las otras dos, se desarrollará una explicación común para las tres.

Si en el futuro se desea agregar un nuevo tipo de nota, como "Nota de Video", sería necesario extender la aplicación para acomodar este nuevo tipo de nota. Para hacerlo, se requerirían los siguientes cambios:

CREAR UNA NUEVA FÁBRICA CONCRETA

Actualmente, tenemos una fábrica concreta llamada *FabricaNotasTexto* que se encarga de crear notas de texto. Para soportar un nuevo tipo de nota, como "Nota de Video", se tendría que modificar o crear una nueva fábrica concreta que implemente el *Factory Method* para crear objetos de ese nuevo tipo. Por ejemplo, podríamos tener una clase *FabricaNotasVideo* que implemente el método "*crearNota()*" para crear notas de video.

CREAR UNA NUEVA CLASE DE NOTA CONCRETA

Para el nuevo tipo de nota, en este caso, "Nota de Video", se tendría que crear una nueva clase que implemente la interfaz *Nota*. Por ejemplo, podríamos crear la clase *NotaVideo* que proporcionaría su propia implementación de la operación "*contenido()*" específica para las notas de video.

MODIFICAR EL CÓDIGO CLIENTE

El código cliente, es decir, las partes de la aplicación que utilizan las fábricas para crear notas, deberían ser actualizadas para interactuar con la nueva fábrica concreta (*FabricaNotasVideo*) en lugar de la fábrica de notas de texto (*FabricaNotasTexto*). En el caso de que se quisieran usar ambas notas, se dejaría la implementación de ambas fábricas.

En resumen, para añadir un nuevo tipo de nota en el futuro, deberíamos crear una nueva fábrica concreta para ese tipo de nota, implementar la clase de nota correspondiente y asegurarnos de que el código cliente pueda interactuar con esta nueva fábrica. El patrón *Factory Method* facilita la extensibilidad de la aplicación al permitirnos agregar nuevas clases de fábrica y tipos de notas sin perturbar el funcionamiento de las partes existentes de la aplicación.