PRÁCTICA 5: TIEMPOS MODERNOS[1]

1. Introducción

Los principios de *indirección* e *inversión de dependencias* establecen que cuando un módulo A depende de otro módulo B, dicho módulo A sólo debería depender de los detalles de alto nivel de dicho módulo B, permaneciendo sus detalles de bajo nivel de ocultos. De esta forma, variaciones o cambios realizados en los elementos de bajo nivel del módulo B no deberían afectar al módulo A.

Por ejemplo, supongamos que tenemos un módulo[2] *Agenda* que utiliza un módulo *Date* para manipular fechas. Una fecha podría implementarse, al menos, de dos maneras completamente distintas: (1) almacenando los tres valores que constituyen la fecha, es decir, el día, el mes y el año, de manera separada; o, (2) almacenando el número de días transcurridos desde una fecha determinada, como, por ejemplo, el 1 de enero de 1970.

La primera alternativa permite visualizar muy fácilmente la fecha representada, por lo que es realmente fácil mostrarla en un dispositivo de salida determinado. Por el otro lado, la segunda representación permite calcular con mayor comodidad el número de días transcurridos entre dos fechas. Este cálculo se realiza comúnmente en una Agenda para determinar, por ejemplo, los días que faltan para alcanzar una determinada fecha, o cuánto ha transcurrido desde un determinado evento. Por tanto, dependiendo del uso que la clase *Agenda* vaya a realizar de la clase *Date*, una implementación podría resultar más adecuada que otra.

No obstante, en el caso ideal, un módulo *Agenda* debería depender exclusivamente de la interfaz pública del módulo *Date*, y no de la forma en la cual se implementa ésta última. Es decir, *Agenda* debería depender sólo de *Date* y no implementaciones concretas de *Date* como pueden ser *UserFriendlyDate* (versión donde se almacenan los tres valores de la fecha de manera separada) o *CounterDate* (versión donde se almacenan el número de días transcurridos desde una fecha fija que hace de origen de los tiempos). Además, debería ser relativamente fácil cambiar la implementación concreta del módulo *Date* que queremos que utilice una determinada aplicación.

En los lenguajes orientados a objetos, esta independencia se alcanza mediante la siguiente estrategia: cuando una clase A necesita utilizar otra clase B, se crea una interfaz (o clase abstracta) que contenga la interfaz pública de B. La clase A utiliza esta interfaz como tipo para las referencias a la clase B. De esta forma se consigue que la clase A no dependa de implementaciones concretas de B. En nuestro ejemplo, se crearía una interfaz para la clase *Date* y se haría que la clase *Agenda* contuviese referencias exclusivamente a esta interfaz. A continuación, se crean tantas implementaciones de la interfaz de B como formas de implementar dicho módulo haya.

---

[1] En referencia a una obra maestra del cine mudo, *Tiempos Modernos* (Charles Chaplin, 1936).

[2] Entiéndase como ejemplo de módulo, el concepto de clase, aunque un módulo no tiene porque ser exactamente una clase.

Por ejemplo, en nuestro caso se crearía una interfaz *Date*, especificando las operaciones que este tipo es capaz de realizar, y dos subclases concretas de la *Date*, que denominaríamos *UserFriendlyDate* y *CounterDate*. Cada clase correspondería con una de las estrategias de implementación previamente comentadas.

De esta manera, gracias al polimorfismo y la vinculación dinámica, se pueden utilizar instancias de estas clases concretas allá donde se requiera un objeto del tipo *Date*. De esta forma, la clase *Agenda* puede evitar hacer referencia a las clases concretas de *Date* y trabajar exclusivamente sobre el tipo abstracto, aceptando sin problemas implementaciones de ambos tipos.

Esta técnica, sin embargo, tiene un punto débil. Dado que no se pueden crear instancias ni de clases abstractas ni de interfaces, si la clase A necesita crear una instancia de la clase B, dicha clase A necesitaría hacer referencia a una clase concreta de B. Por ejemplo, si la clase *Agenda* necesitase crear una instancia de *Date*, la clase *Agenda* tendría que hacer referencia a *UserFriendlyDate* o a *CounterDate*, por lo que al final acabaría acoplada a una de sus implementaciones concretas.

Para solventar este problema, existe un conjunto de patrones de diseño, denominados estructurales, entre los que destaca el *Abstract Factory*. Este patrón de diseño a alcanzado un gran éxito y popularidad, utilizándose en la práctica totalidad de aplicaciones creadas hoy en día. Para facilitar su aplicación, en los últimos años se han creado una serie de librerías y *frameworks*, conocidos como *inyectores de dependencias*, cuyo objetivo es permitir la especificación de factorías de manera cuasi declarativa y a un alto nivel, simplificando en gran medida el proceso de desarrollo de factorías abstractas.

El objetivo general de esta práctica es que el alumno aprenda tanto a crear factorías abstractas manualmente como a crearlas con la asistencia de un inyector de dependencias. El siguiente apartado refina esta serie de objetivos genéricos en un conjunto de objetivos concretos.

2. Objetivos

Los objetivos concretos de esta práctica son:

(1) Aprender a utilizar el patrón *Abstract Factory*.
(2) Aprender a utiliza el patrón *Singleton*.
(3) Aprender a utilizar el patrón *Prototype*.
(4) Aprender a utilizar un inyector de dependencias.

Para alcanzar dichos objetivos, el alumno deberá aplicar el patrón *Abstract Factory*, complementado con los patrones *Prototype* y *Singleton* a la situación que se describe a continuación. A continuación, se creará una implementación alternativa basada en inyección de dependencias.

3. Configuración del Sistemas de Archivos *Sparrow*.

El Sistema de Archivos *Sparrow*, de acuerdo con la especificación creada por la organización *La Perla Negra*, puede configurarse mezclando los sistemas de visualización y las estrategias para la conversión de nombres de los modos que se describen a continuación.

En caso de que se opte por utilizar un sistema de visualización compacto, sólo se podrá utilizar la estrategia de visualización de nombres conocida como *Internacional Gallega*. Esta configuración se conoce como la configuración *Básica*.

Si se decide utilizar el sistema de visualización extendido, se puede optar tanto por no usar ninguna estrategia como por utilizar la *Internacional Gallega* o la *Internacional Catalana*. Estas configuraciones se conocen como *Estándar*, *Extendida Gallega* y *Extendida Catalana*, respectivamente.

Además, en el caso de la visualización extendida, el sistema de Archivos *Sparrow* debe soportar la existencia de una configuración personalizable donde la estrategia de visualización de nombres no esté predefinida, dejándose ésta a elección del usuario. Esta estrategia, además, no tendría por qué estar entre las inicialmente definidas en la especificación de la *Perla Negra*. Por ejemplo, podría optarse por utilizar una estrategia donde la letra *ñ* se reemplazase por los caracteres *nn*, tal como ocurría en castellano antiguo. Esta última configuración se conoce como la configuración *Abierta*.

La Tabla 1 resume las diferentes configuraciones soportadas por el sistema de archivos *Sparrow*.

Configuración	Visualización	Estrategia Nombres
Básica	Compacto	Internacional Gallega
Estándar	Extendido	Ninguna
Extendida Gallega	Extendido	Internacional Gallega
Extendida Catalana	Extendido	Internacional Catalana
Abierta	Extendido	Proporcionada por el usuario

Tabla 1. Modos de Funcionamiento del Sistema de Archivos Sparrow.

4. Actividades

El alumno, para alcanzar los objetivos planteados, deberá realizar satisfactoriamente las siguientes actividades, utilizando para ello la implementación creada en prácticas anteriores:

1. Crear una factoría abstracta para soportar las diferentes configuraciones del sistema de archivos *Sparrow*.
2. Crear factorías concretas para las configuraciones *Básica, Estándar, Extendida Gallega* y *Extendida Catalana*.
3. Aplicar el patrón *Singleton* a las factorías creadas en el punto anterior de manera que sólo pueda existir una instancia de la factoría abstracta en tiempo de ejecución.

4. Crear un programa de prueba que visualice el sistema de archivos de la Figura 1 utilizando la configuración *Extendida Catalana*. Dicho programa de pruebas utilizará las factorías creadas anteriormente para la instanciación de las clases que se considere necesario.

```
d Raiz
   d Directorio Vacio
   d Directorio Con Archivo Unico
      f foto001.jpg
   d Directorio Con Archivo Comprimido Simple
      f foto002.jpg
      e foto001.jpg
      c ccSimple.zip
         d Directorio Vacio En Archivo Comprimido
         f foto003.jpg
         e foto001.jpg
   d Directorio con Directorio Anidado
      f foto004.jpg
      e ccSimple.zip
      e Directorio Vacio
      d Directorio con Archivo Comprimido Complejo
         f foto005
         f foto006
         c ccComplejo.zip
            c ccAnidada.zip
               f foto007.jpg
            f foto008.jpg
```

Figura 1. Ejemplo de Sistema de Archivos Sparrow

5. Crear una factoría para la configuración *Abierta*. Dicha factoría utilizará el patrón *Prototype* para permitir la incorporación de estrategias de visualización de nombres distintas a las actualmente definidas.
6. Crear un programa de prueba que visualice el sistema de archivos de la Figura 1 utilizando la configuración *Abierta* personalizada con la estrategia *YourOcre*, donde se sustituye la ñ por los caracteres *ni*. Dicho programa de pruebas utilizará la factoría creada para la configuración *Abierta* para la instanciación de las clases que se considere necesario.
7. En un proyecto aparte, crear módulos *Ninject* para especificar las dependencias a inyectar en cada configuración posible del sistema de archivos *Sparrow*, a excepción de la configuración *Abierta*.
8. En el proyecto del punto anterior, crear un programa de prueba que, utilizando el inyector de dependencias *Ninject* para la creación de los objetos cuando se considere necesario, visualice el sistema de archivos de la Figura 1 utilizando la configuración *Extendida Gallega*.

Para facilitar la realización de la práctica, el alumno tiene disponible en la plataforma *moodle* un pequeño proyecto de ejemplo donde se utiliza *Ninject* para realizar la inyección de dependencia.

Los siguientes apartados contienen una serie de instrucciones básicas para aplicar los patrones *Abstract Factory*, *Singleton* y *Prototype*.

5. Pasos para Instanciar el Patrón *Abstract Factory*

Para aplicar el patrón *Abstract Factory* se recomienda seguir los siguientes pasos:

1. Crear una interfaz (o clase abstracta) que ejerza de *AbstractFactory*. Se recomienda no llamar a esta clase AbstractFactory, sino ponerle un prefijo al término *Factory* que indique qué tipo de objetos crea dicha factoría.
2. Identificar las clases abstractas (o interfaces) de las que es necesario crear objetos concretos.
3. Por cada conjunto de clases concretas de dichas clases abstractas que represente una posible configuración del sistema, crear una factoría concreta. Se recomienda utilizar como prefijo al término *Factory* de cada factoría concreta el nombre de la configuración que representa.
4. Implementar los métodos de las factorías concretas de manera que retornen un objeto del tipo concreto que corresponda conforme a la configuración que representan.

6. Pasos para Instanciar el Patrón *Singleton*

Para aplicar el patrón *Singleton*, se recomienda seguir los siguientes pasos:

1. Proteger el constructor de la clase a la cual queremos aplicar el patrón *Singleton* de manera que dicho constructor deje de ser público.
2. Crear una variable de instancia o atributo estático en dicha clase que represente la instancia única de dicha clase. Se recomienda utilizar como nombre de dicho atributo *instance, theInstance, uniqueInstance* o algún termino similar.
3. Crear un método estático que retorne un objeto de la clase a la que estamos aplicando el patrón *Singleton*. Utilizar como nombre para este método *getInstance*.
4. El método *getInstance* comprobará si el atributo que representa la instancia única que puede existir de la clase a la que se aplica el patrón *Singleton* existe. Si no existe, crearía dicha instancia. Si existe, simplemente la devuelve.

7. Pasos para Instanciar el Patrón *Prototype*.

Para aplicar el patrón *Singleton*, se recomienda seguir los siguientes pasos:

1. Identificar la clase, potencialmente abstracta, que servirá de interfaz o tipo común para los futuros *prototipos*.
2. Añadir, si el lenguaje de programación utilizado no lo soporta por defecto, una operación de clonado, denominada *clone*, a la clase identificada en el punto anterior.
3. Identificar todas las clases que necesiten crear instancias concretas de la clase identificada como prototipo. Añadir una referencia a la clase prototipo a cada una de estas clases.
4. Siempre y cuando una clase identificada en el punto anterior necesite crear una instancia del prototipo, ésta llamará al método *clone* de dicho prototipo.