Práctica 05 - Patrón Abstract Factory

Tiempos Modernos [1]

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

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 `Fecha` 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;
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 mostrarla en un dispositivo de salida cualesquiera resulta bastante trivial. Por otro lado, la segunda representación permite calcular con mayor comodidad el número de días transcurridos entre dos fechas. Este cálculo se podría realizar 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 `Fecha`, 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 `Fecha`, y no de la forma concreta en la cual se implementa ésta última. Es decir, `Agenda` debería depender sólo de `Fecha` y no de implementaciones concretas de `Fecha` como podrían 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). Además, cambiar la implementación concreta del módulo `Fecha` que queremos utilizar en la clase `Agenda` debería ser una operación relativamente fácil de realizar.

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`. Esta interfaz es común a todas las implementaciones concretas de `B` y está libre de detalles ed bajo nivel. De esta forma, la clase `A` sólo utiliza esta interfaz como tipo para las referencias a la clase `B`, evitando así depender de implementaciones concretas de la clase `B`.

En nuestro ejemplo concreto, crearíamos una interfaz para la clase `Fecha`. Esta interfaz contendría sólo la declaración de las operaciones que se pueden realizar sobre un objeto del tipo `Fecha`, cualesquiera que sea su implementación. A continuación haríamos que la clase `Agenda` sólo contuviese referencias a esta interfaz y crearíamos tantas implementaciones o clases concretas de la interfaz `Fecha` como formas de implementar dicha interfaz conozcamos. En nuestro caso, crearíamos como clases concretas `UserFriendlyDate` y `CounterDate`. Cada clase concreta implementaría una de las dos formas de almacenar fechas descritas con anterioridad. Ambas clases heredarían de `Fecha`.

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

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 implementación concreta de `B`. Por ejemplo, en nuestro ejemplo, si la clase `Agenda`

necesitase crear una instancia de la clase `Fecha`, la clase `Agenda` tendría que hacer referencia a las clases concretas `UserFriendlyDate` o `CounterDate`, por lo que al final acabaría acoplada a alguna de estas implementaciones concretas.

Para solventar este problema, existe un conjunto de patrones de diseño particulares denominados *creacionales*. Dentro de este conjunto de patrones destaca especialmente el *Abstract Factory*, el cual ha alcanzado gran éxito y popularidad, utilizándose en la práctica totalidad de las aplicaciones creadas hoy en día. Tal es su popularidad, que, 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 a alto nivel, de manera cuasi declarativa, simplificando en gran medida su desarrollo.

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

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

Objetivos

Los objetivos concretos de esta práctica son:

1. Aprender a utilizar el patrón *Abstract Factory*.
2. Aprender a utilizar 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. Además, se creará una implementación alternativa basada en *inyección de dependencias*.

Configuración del Sistema de Archivos Sparrow

El *Sistema de Archivos Sparrow*, de acuerdo con la especificación original creada por la organización *La Perla Negra*, puede distribuirse en cinco configuraciones diferentes, conocidas como *básica*, *estándar*, *extendida gallega*, *extendia catalana* y *abierta*. Cada configuración utiliza un mecanismo de impresión y una estrategia de visualización de caractereres castellanos específicos. Los mecanismos de impresión y estrategias de visualización de caracteres castellanos utilizados por cada configuración se resumen en la Tabla 1.

Tabla 1. Posibles configuraciones del Sistema de Archivos Sparrow

Configuración	Impresión	Estrategia Caracteres
Básica	Compacta	Internacional Gallega
Estándar	Extendida	Ninguna
Extendida Gallega	Extendida	Internacional Gallega
Extendida Catalana	Extendida	Internacional Catalana
Abierta	Extendida	Proporcionada externamente

Las configuraciones *basica*, *estándar*, *extendida gallega* y *extendida catalana* representan diferentes combinaciones de los vistantes de impresión con las estrategias de conversión de los caracteres propios del castellano.

Además, en el caso de la configuración *abierta*, el *Sistema de Archivos Sparrow* debe soportar la existencia de una configuración personalizable donde la estrategia de conversión de los caracteres propios del castellano pueda ser configurada a gusto del usuario. En este caso, por ejemplo, podría optarse por utilizar una estrategia donde la letra ñ se reemplazase por los caracteres `nn`, tal como ocurría en el castellano antiguo.

Actividades

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

1. Crear una *factoría abstracta* para soportar la creación de estrategias y visitantes de impresión de acuerdo con las diferentes configuraciones existentes dentro de la especificación original del *Sistema de Archivos Sparrow*.
2. Crear *factorías concretas* que den soporte a 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. Modificar el programa de prueba creado en prácticas anteriores para que al inicializar la aplicación se inicie las factoría abstracta con la configuración *extendida gallega*.
5. Modificar la clase `SparrowView` creada en prácticas anteriores para que haga uso de las factorías abstractas para la creación de los visitantes de impresión.
6. Crear una factoría concreta para la configuración *abierta*. Dicha factoría utilizará el patrón *Prototype* para permitir la incorporación de estrategias de cambio de caracteres propio del castellano que puedan ser distintas a las actualmente definidas.
7. Crear una nueva estrategia de reemplazo de caracteres propios del castellano denominada *YourOcre* [3]. En dicha estrategia se preservan las vocales con tilde y se sustituye el carácter `ñ` por los caracteres `ni`.
8. Modificar el programa de pruebas para que al iniciarse la aplicación se inicialice la factoría abstracta a la configuración abierta utilizando la estrategia *YourOcre* como estrategia de reemplazo de caracteres castellanos.
9. Crear módulos *Ninject* para especificar las dependencias a inyectar en las configuraciones *básica*, *estándar*, *extendida gallega* y *extendida catalana*.
10. Modificar la clase `SparrowView` para que utilice el inyector de dependencias para la creación de los visitantes de impresión.

Para facilitar la realización de la práctica, se pone a disposición del alumno el siguiente proyecto de *Visual Studio* donde se hace utilización de la inyección de dependencias mediante *Ninject*.

Proyecto Ejemplo Ninject

[3] En homenaje a la gran tuitera @LecheConHiel

Criterios de Autoevaluación

Para verificar que el alumno ha implementado correctamente los patrones *Abstract Factory*, *Singleton* y *Prototype* se aconseja verificar los siguientes puntos:

1. Existe una única factoría abstracta para toda la aplicación.
2. Existe un método *create* en la factoría abstracta por cada clase que pueda variar dentro de una configuración concreta.
3. Los métodos *create* de la factoría abstracta no contiene parámetros.
4. Existe una factoría concreta por configuración posible de la aplicación.
5. La factoría abstracta se inicializa sólo al iniciar la aplicación, dentro del *main*.
6. Los constructores de las factorías no son públicos.
7. La creación de nuevos objetos a partir del prototipo proporcionado se realiza mediante el clonado de dicho prototipo.

Apéndice: Cómo instalar Ninject en un proyecto VS

Para instalar *Ninject* dentro de un proyecto de Visual Studio se deben realizar las siguientes acciones:

1. En el *Explorador de Soluciones* (workspace), seleccionar el proyecto donde se desea añadir *Ninject*.
2. Pulsar en el ratón boton derecho, y sobre el menú que se despliega seleccionar *Administrador paquetes NuGet*.
3. Dentro de la ventana que se abre sobre el editor, seleccionar la pestaña *Examinar*.
4. Escribir en la caja de búsqueda *Ninject*.
5. Seleccionar entre los resultados *Ninject*.
6. A la derecha de la lista de resultados, teniendo *Ninject* seleccionado, pulsar sobre el botón *Instalar*