LA UNIVERSIDAD EN INTERNET

PATRONES DE CREACIACÓN

Comprender los patrones de creación

Profesor: Alfonso Ortega De La Puente

Ernesto González Pradas
ernesto.gonzalez023@comunidadunir.net

Índice

Introducción

El objetivo de esta actividad será comprender los patrones de creación, mediante su aplicación con un ejemplo real de uso. Asimismo, buscaremos presentar un análisis crítico de las ventajas e inconvenientes de dichos patrones a la hora de resolver el problema que se nos presenta.

Realizaremos un análisis justificado de las ventajas e inconvenientes que aporta el uso del patrón de creación "Factoría Abstracta" y, además, llevaremos a cabo el diseño media diagramas UML y su correspondiente implementación en Java. Por último, mostraremos la facilidad de evolución del sistema que nos aporta el patrón seleccionado.

El dominio del caso seleccionado son skins de personajes de videojuegos, inicialmente para verano e invierno y posteriormente para Halloween y una prenda añadida.

Ejercicio 1: Análisis ventajas e inconvenientes del patrón

Para poder realizar un análisis de ventajas e inconvenientes del patrón “Factoría Abstracta” primero explicaremos cuando debemos usar este patrón. Y es que, siempre que se nos presente un problema en que tengamos que modelar familias de objetos relacionados sin especificar su clase concreta, pensaremos en usar el patrón de creación “Factoría Abstracta”.

Sin embargo, otros patrones de creación como puede ser “Builder” los asociaremos cuando tengamos un problema en el que tengamos que crear objetos complejos paso a paso. Con este patrón podemos crear distintas representaciones de un objeto utilizando la misma implementación de construcción.

Siguiendo con el análisis, el problema que se nos plantea es la de crear familias de objetos ya que se van a crear, por un lado, skins de verano (esto sería una familia ya que está compuesta por “CubrePiesVerano”, “InferiorVerano” y “SuperiorVerano”), y por otro, skins de invierno (formadas por “CubrePiesInvierno”, “InferiorInvierno” y “SuperiorInvierno”).
Con la “Factoria Abstracta” la principal ventaje que tenemos frente a cualquier otro patrón de creación, para este problema, es que, con crear una interfaz de fábrica abstracta con los objetos abstractos después se pueden implementar todas las familias de fábricas concretas que se necesiten o se quieran ampliar.

Si esto lo hiciéramos con el patrón “Builder”, tendríamos que crear por cada uno de los tipos de skin un constructor o builder concreto, por lo que no solo no nos facilitaría la ampliación a un futuro si se quisiera implementar una nueva familia (como es en nuestro caso la skin de Halloween) si no que sería muy tedioso.

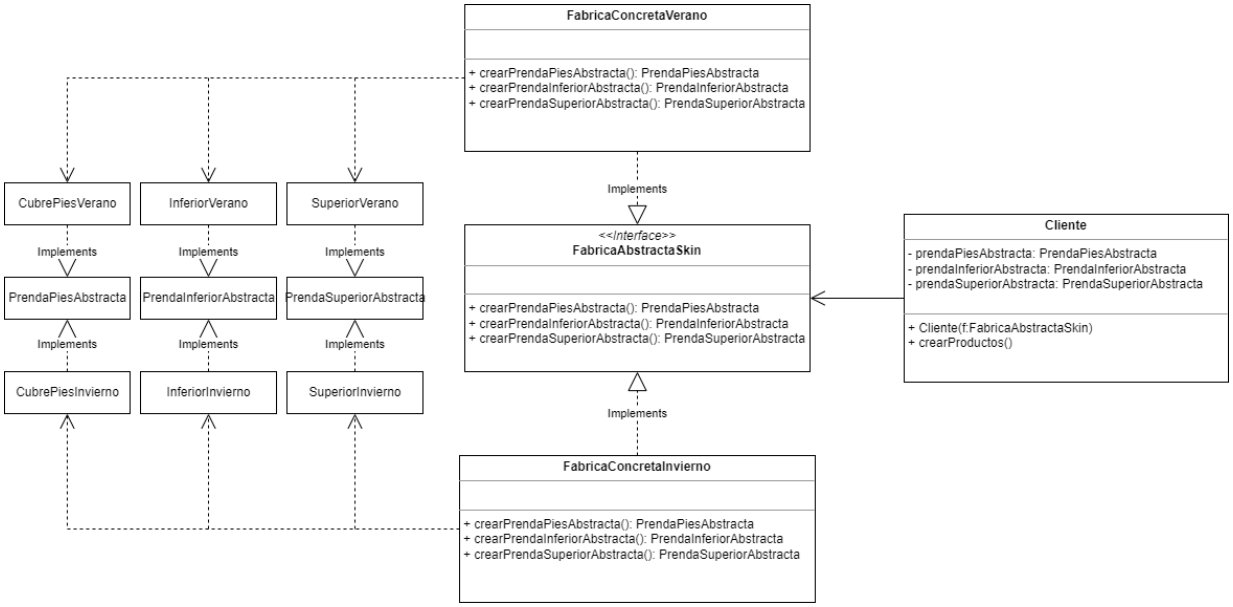
Gracias al patrón “Fabrica Abstracta”, para la primera ampliación, solo tendríamos que crear una clase “FabricaConcretaHallowen” que implementase de la “FabricaAbstractaSkin” del planteamiento base y tres clases (“ZapatosCalabaza”, “MallasEsqueleto” y “CamisetaEsqueleto”) que implementen las interfaces “PrendaPiesAbstraca”, “PrendaInferiorAbstracta” y “PrendaSuperiorAbstracta”. Si usásemos otro patrón de creación, como el “Builder” no podríamos reutilizar las interfaces ya que tendríamos un director con el que iríamos creando paso a paso cada una de las clases sin poder reutilizar dichas interfaces.

Hasta ahora solo hemos visto las ventajas de este patrón con respecto nuestro problema que se nos planea, pero vamos a ver cuál es el principal inconveniente. Y, es que, cuando decidimos usar el patrón “Factoría Abstracta” dependiendo del problema

se nos puede complicar mucho la lógica al tener tantas interfaces y clases que tienen que implementarse unas a otras por lo que debemos tener muy claro desde el principio del diseño como vamos a modelar nuestro sistema.

Ejercicio 2: Implementación y diagrama UML del patrón

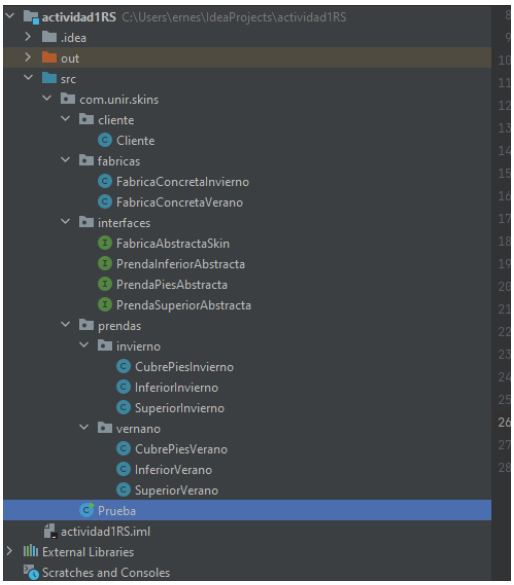
A continuación, vamos a ver el diseño en UML de nuestro problema y posteriormente veremos la implementación en código Java. Este sería el diseño UML básico de nuestro problema:



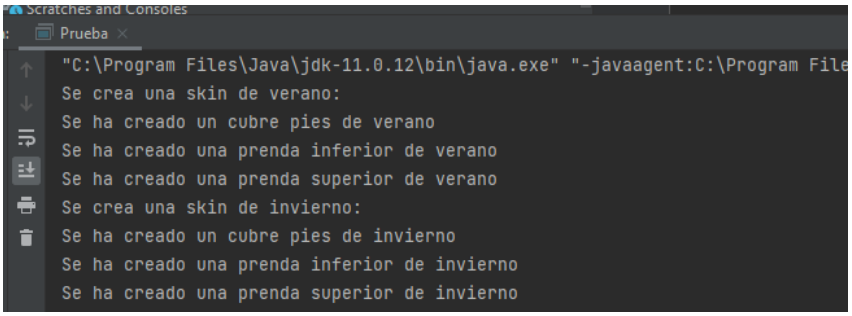
Como se puede observar tenemos:

- La interfaz “FabricaAbstractaSkin” que declara un grupo de métodos para crear cada uno de los productos concretos.
- Las fábricas concretas, “FabricaConcretaVerano” y “FabricaConcretaInvierno”, que implementan los métodos de creación de la fábrica abstracta. Cada una de estas fábricas concretas se identifica con un tipo específico de los productos y crea solo sus variantes.
- Los productos abstractos, “PrendaPiesAbstracta”, “PrendaInferiorAbstracta” y “PrendaSuperiorAbstracta”, que declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.
- Y finalmente los productos concretos, “CubrePiesVerano/Invierno”, “InferiorVerano/Invierno” y “SuperiorVerano/Invierno”, que son implementaciones distintas de los productos abstractos vistos en el párrafo anterior.

Ahora veremos, como ha quedado la implementación base de nuestro sistema. Nuestro proyecto ha quedado de la siguiente forma:



Si ejecutamos la clase main que se encuentra en nuestra clase llamada “Prueba”, tendremos el siguiente resultado por pantalla:

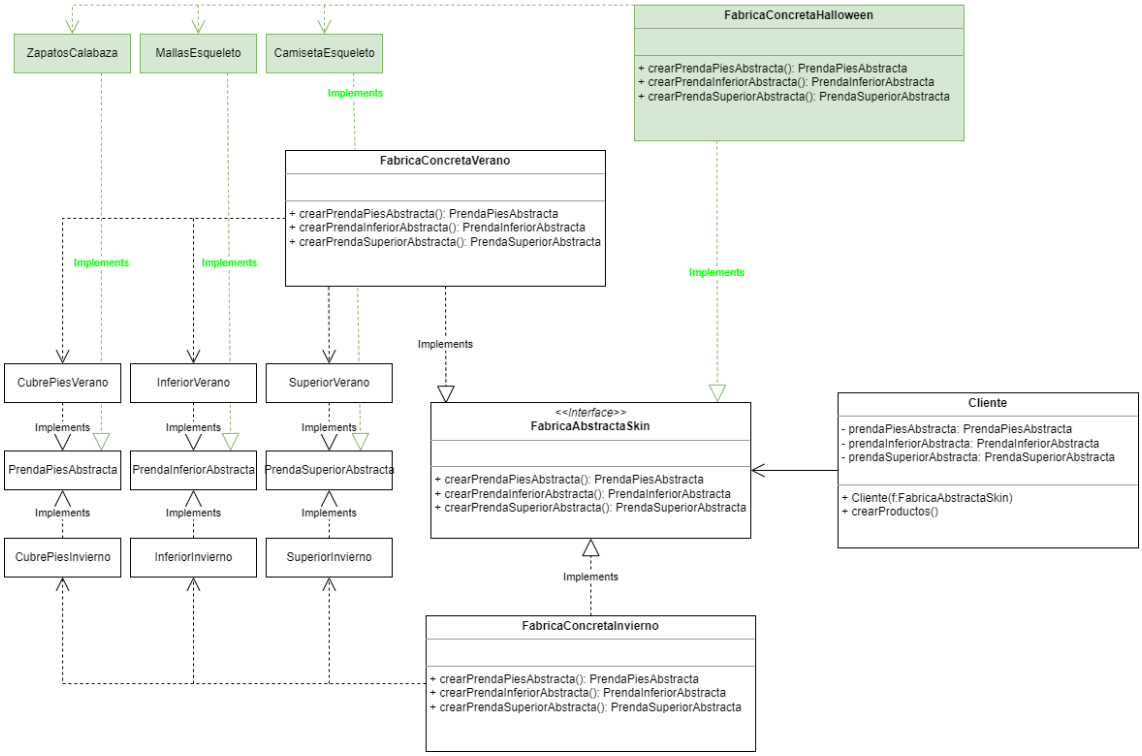


Ejercicio 3: Ampliaciones del sistema propuesto

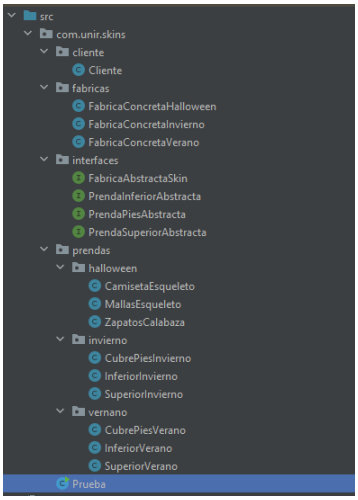
A continuación, vamos a ver los diagramas UML y las implementaciones de cada una de las extensiones pedidas por el enunciado de la actividad.

Ampliación 1

En esta ampliación, se nos pide comprobar el uso del patrón añadiendo una nueva skin con el motivo de un evento de Halloween. Esta ampliación consistirá en añadir una nueva familia de elementos (“FabricaConcretaHalloween”) que implementara de la “FabricaAbstractaSkin” y los productos concretos “ZapatosCalabaza”, “MallasEsqueleto” y “CamisetaEsqueleto”. Nuestro diagrama UML quedaría de la siguiente forma:



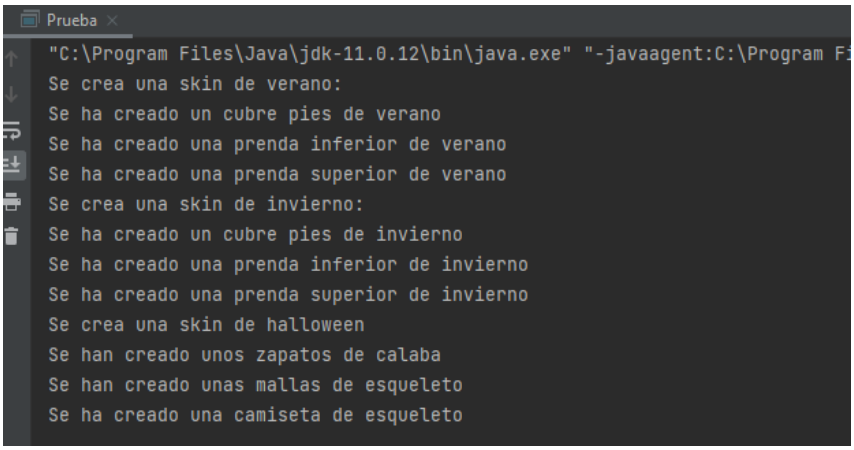
Nuestra estructura del proyecto implementado en Java quedaría así:



Para poder mostrar por pantalla los resultados nuevos de la nueva skin, tenemos que modificar el main proporcionado por la actividad y quedaría de la siguiente forma:

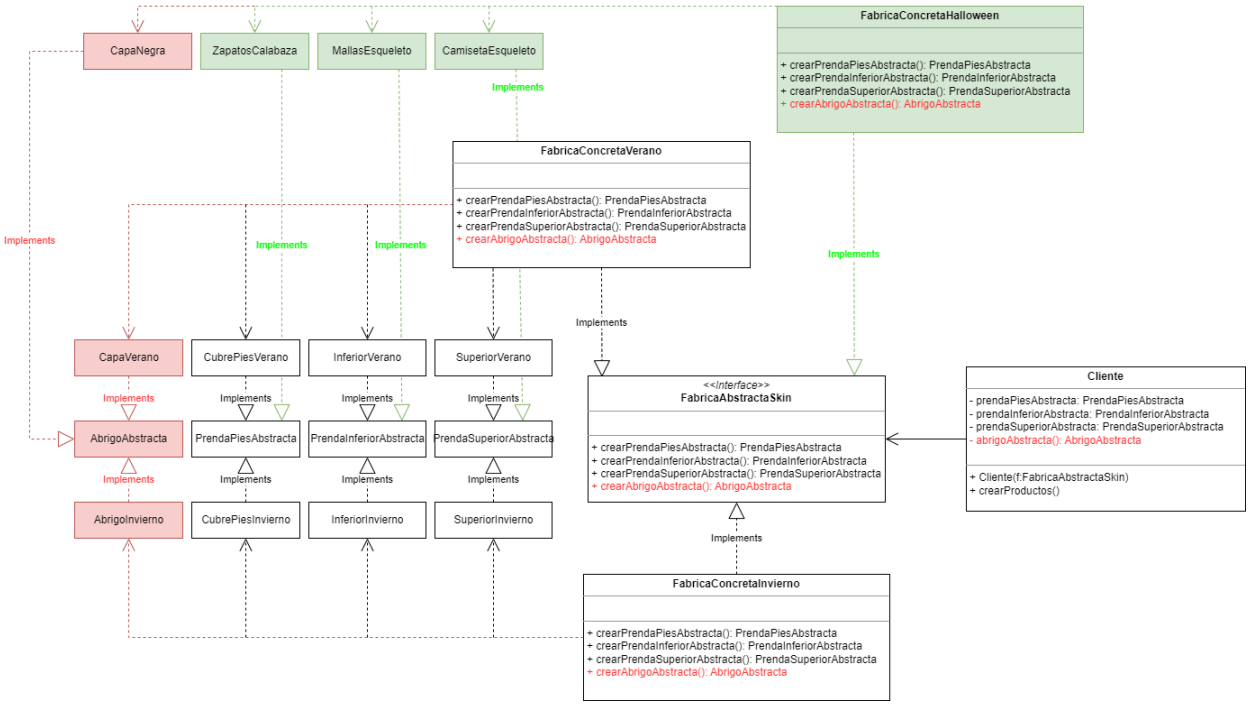
```java
package com.unir.skins;

import ...

public class Prueba {

    public static void main(String[] args) {

        System.out.println("Se crea una skin de verano:");
        Cliente skin_verano = new Cliente(new FabricaConcretaVerano());
        skin_verano.crearProductos();

        System.out.println("Se crea una skin de invierno:");
        Cliente skin_invierno = new Cliente( new FabricaConcretaInvierno());
        skin_invierno.crearProductos();

        System.out.println("Se crea una skin de halloween");
        Cliente skin_halloween = new Cliente(new FabricaConcretaHalloween());
        skin_halloween.crearProductos();
    }

}
```

Y el resultado que nos sale si ejecutamos el método main de nuestra clase “Prueba” sería:

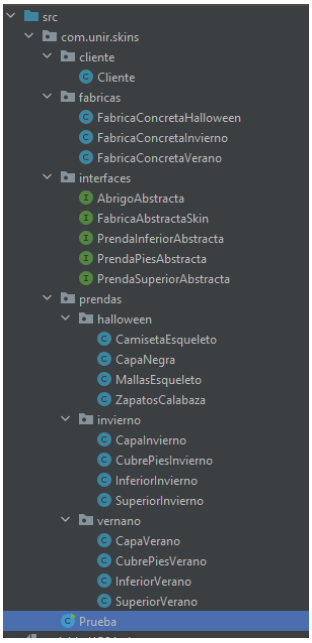


Ampliación 2

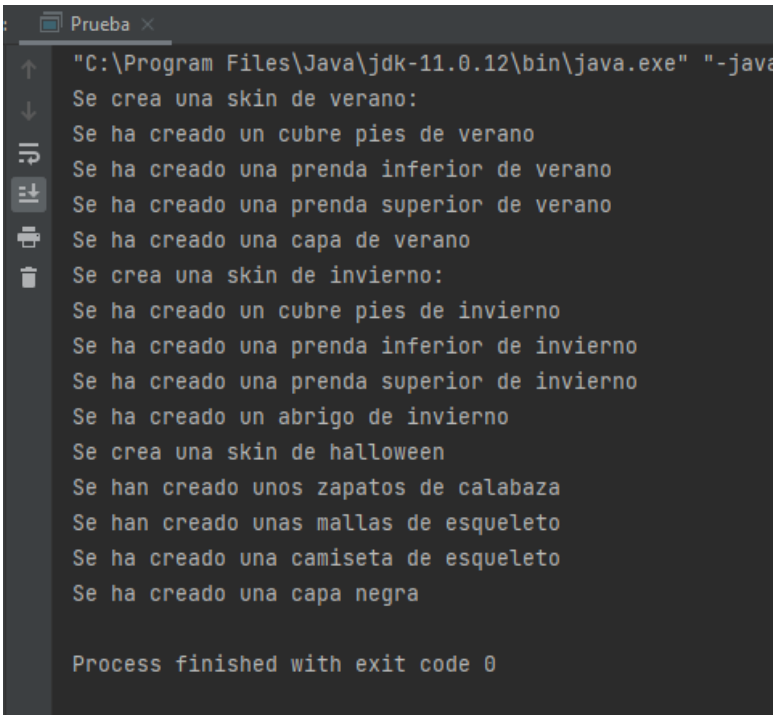
En esta extensión, se pide añadir a la ampliación anterior, un elemento nuevo en cada skin, que concretamente será una prenda de abrigo (“AbrigoAbstracta”). Esto implica que para cada tipo de skin los elementos concretos son: “CapaVerano” para la skin de verano, “AbrigoInvierno” para la skin de invierno y “CapaNegra” para la skin de Halloween. El diagrama UML resultante quedaría de la siguiente manera, indicando los nuevos cambios de la ampliación en color rojo:



La estructura de la implementación con la segunda ampliación sería:



Y el resultado por pantalla, si ejecutamos la clase main sería:


Prueba
"C:\Program Files\Java\jdk-11.0.12\bin\java.exe" "-java
Se crea una skin de verano:
Se ha creado un cubre pies de verano
Se ha creado una prenda inferior de verano
Se ha creado una prenda superior de verano
Se ha creado una capa de verano
Se crea una skin de invierno:
Se ha creado un cubre pies de invierno
Se ha creado una prenda inferior de invierno
Se ha creado una prenda superior de invierno
Se ha creado un abrigo de invierno
Se crea una skin de halloween
Se han creado unos zapatos de calabaza
Se han creado unas mallas de esqueleto
Se ha creado una camiseta de esqueleto
Se ha creado una capa negra
Process finished with exit code 0

Conclusión

Como hemos podido ver a lo largo de la actividad, hemos visto las ventajas y los inconvenientes de utilizar el patrón de creación “Factoría Abstracta” y lo hemos comparado con otros patrones de creación como el “Builder”.

Lo importante es que nos quedemos con las ideas claves para entender ambos patrones y cuando debemos usar uno u otro:

- Factoría Abstracta: cuando queremos representar una familia de objetos relacionados sin especificar sus clases concretas.
- Builder: cuando queremos construir objetos complejos paso a paso y producir así objetos de distintos tipos y representaciones distintas con un mismo código de construcción.

Bibliografía

Puente, A. O. (s.f.). Tema 3. Patrones de Creación.

Puente, A. O. (s.f.). Tema 3. Patrones de Creación (II).