Práctica 03

Noe E. Amador González [redacted]

[redacted]

1. Menciona los principios de diseño esenciales de los patrones Factory, AbstractFactory y Builder.

   - **Factory**
     - Identificar los aspectos que cambian y separarlos de los que se quedan iguales.
     - Crear código abierto a extensión pero cerrado a modificación.
   - **AbstractFactory**
     - Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
   - **Builder**
     - Encapsular la construcción de un objeto/producto y permitir que se construya en pasos.

2. Menciona una desventaja de cada patrón.

   - **Factory**
     - Permitir que una clase difiera la instanciación a subclases puede no llegar a ser conveniente en varios casos.
   - **AbstractFactory**
     - Para añadir un nuevo objeto se necesita volver a escribir todos los métodos de la interfaz los cuales podrían llegar a ser demasiados si los objetos tienen muchas funciones/atributos.
     - Se puede fácilmente comenzar a tener un desorden de archivos en cuanto nuestro programa crece.
   - **Builder**
     - A menudo se usa para construir estructuras compuestas. Lo cual podría dirigirnos a crear objetos muy complejos con muchas composiciones que hay por debajo de su construcción.
     - Builder nos dirige a crear muchas clases cuando nuestro objeto es demasiado grande.

README

- **JUSTIFICACIÓN DEL PATRÓN SELECCIONADO: BUILDER**.
  Yo he elegido Builder porque me ha parecido que la estructura de este patrón de diseño coincide con la tarea a realizar, es decir, queremos construir un auto. En este sentido, Builder posiblemente tiene complejidades similares a los otros 2 patrones. Lo que tiene de especial es que Builder nos permite abstraer a nuestro objeto principal (en este caso BuildAuto) de una manera bastante descriptiva y fácil de identificar en nuestro programa. La forma en la que Builder nos permite descomponer a nuestro objeto en muchas clases (en este caso en piezas) es otra buena manera de representar a un Auto en un diagrama de clases. En general considero que los 3 patrones de diseño son muy parecidos pero lo que hizo a Builder especial es su simplicidad textual y gráfica de identificar los componentes del programa y ser congruente con la tarea que estamos realizando: construir. Por lo que a la hora de implementarlo me pareció ser intuitivo y flexible para tareas como construir objetos con valores de atributos aleatorios.

- La implementación de la práctica se ha llevado a cabo con la versión 8 de Java.

- Para correr la simulacion hay que compilar todos los archivos *.java y despues utilizar la clase *Main* como la **clase principal** que ejecutará la simulación.

- Tomar en cuenta en la simulacion que, para cada iteracion en la que se construye un nuevo auto, el sistema automaticamente le asigna $100 al usuario para gastar y construir un auto. Este valor ha sido seleccionado porque, aun construyendo el auto más caro posible, con este saldo es suficiente para construirlo.

- En este diagrama **no especifiqué metodos constructores**.
  Tambien **omití la representacion de la clase *Menu***, la cual utilicé en la clase principal (Main) como apoyo para imprimir los menus necesarios en consola. Lo anterior con el fin de que la clase Main no se viera saturada de código repetido.