UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Modelado y programación.

Rosa Victoria Villa Padilla

[2025-1] (14/09/2024)

Práctica 2:

Decorator, Iterator, State y Template

Equipo: Christian.

Leon Navarrete Adam Edmundo
Rubio Resendiz Marco Antonio
Valencia Pérez Guillermo Emanuel

Ciudad Universitaria, Cd. Mx. 2024

Anotaciones sobre la estructura de la práctica.

La práctica consistió en implementar un programa simulador de aplicación de viajes para trasladar usuarios de un punto a otro. Se le da al usuario la posibilidad de elegir ubicación, destino y vehículo, así como de agregarle diversos aditamentos a este último.

Elección de patrones de diseño.

Template.

Elegimos el patrón de diseño decorator para el diseño del comportamiento de los Vehículos, de esta manera el flujo general está marcado por el método ejecutaViaje() de la clase abstracta Vehiculo y todas las subclases (Carro, Scooter, Autobus, etc.) seguirán el mismo comportamiento con la excepción de la forma en la que recargan combustible o la forma en la que siguen sus rutas.

Nota: Agregamos la clase VehiculoNulo a las clases concretas de Vehiculo para evitar usar null como argumento en las clases concretas para el patrón decorator (esto nos iba a llevar eventualmente a un NullPointerException). Esta clase es particularmente útil para poder generar estructuras iterables de las clases concretas de los Aditamentos, pues nos sirve para poder mostrarle al usuario los aditamentos mediante instancias y trabajar sobre estas.

State.

Elegimos el patrón de diseño state para definir la forma en la que el vehículo modifica o restringe las acciones que puede realizar a lo largo del flujo marcado por el método ejecutaViaje() de la clase Vehículo. Esto se implementa a través de la interfaz EstadoVehiculo que define los siguientes métodos;

- encender(): Método que en sí solo nos sirve de indicador para poder dar un “inicio” a los estados del Vehiculo, se hizo de esta forma para poder definir, al menos conceptualmente, el punto de inicio de los estados del vehículo.
- movimiento(): Para setear el estado a EstadoEnMovimiento si es posible.
- terminarViaje(): Para setear el estado a EstadoFinDelViaje si es posible.
- esperandoViaje(): Para setear el estado a EstadoEsperando si es posible.
- abrirPuertas(): Abre las puertas del vehículo según sea el caso, esto lo metimos aquí para marcar las posibles restricciones que podría tener el vehículo según su estado. Por ejemplo, si el vehículo está en movimiento, este método tira un mensaje indicando que no es posible abrir las puertas.
- sinCombustible(): Para setear el estado a EstadoRecargandoCombustible si es posible.

Todas las clases concretas que implementen la interfaz tendrán un comportamiento distinto a las demás y una forma de cambiar el estado de una instancia de Vehiculo si la implementacion indica que es una posibilidad para el método.

Decorator.

Elegimos el patrón de diseño decorator para brindar la posibilidad de integrarle aditamentos a una instancia de Vehiculo. Se implementó mediante la clase abstracta Aditamento que extiende de la clase Vehiculo. La clase recibe una instancia de Vehiculo y la asigna como atributo, de esta instancia recibida se proporciona la implementación de los métodos abstractos de la clase Vehiculo, sobrecargando los métodos de la instancia recibida para concretar la implementación.

Todos los aditamentos implementan el método descripcion() que amplía la descripción del Vehiculo agregando el nombre del aditamento correspondiente. Además, cuentan con un método llamado envolver(), que a partir de una instancia de Vehiculo y una instancia de Aditamento es capaz de retornar una instancia del aditamento correspondiente, de esta forma el proceso de decoración de los Vehículos es algo que puede ocurrir internamente, siendo ahora más inmediato el proceso de decoración. La utilidad de este último método se ve en la clase AditamentosManager, pues si contamos con una lista de Aditamentos entonces es mucho más inmediato utilizar envolver() para decorar un vehículo.

Nota: El patrón fue implementado con una clase abstracta y no con una interfaz porque, para nosotros, resultó más práctico sobrecargar los métodos de la instancia de Vehiculo recibida para completar la implementación de la clase Vehiculo y dejar el resto de métodos al despacho dinámico de java. Razonamos que si usaramos una interfaz sería necesario realizar más implementaciones de las que vimos realmente necesarias.

Iterator.

Elegimos el patrón iterator para implementar una Lista de instancias de Vehículo (VehiculoIterator y VehiculoIterable) y un Diccionario de ubicaciones (DestinoIterator y DestinoIterable) para los posibles destinos. En general nos apoyamos de estructuras del jdk como ArrayList y HashMap para desarrollar esta parte.

Respecto al uso del proyecto.

El programa fue desarrollado en la versión de java 11.0.22 y se empleó linux (debian y ubuntu) durante su desarrollo. Para ejecutarlo en linux (para distribuciones basadas en debian) es necesario usar los siguientes comandos en terminal:

```
$ javac -d bin $(find src -name "*.java")
$ java -cp bin mx.unam.ciencias.modelado.practica2.Main
```