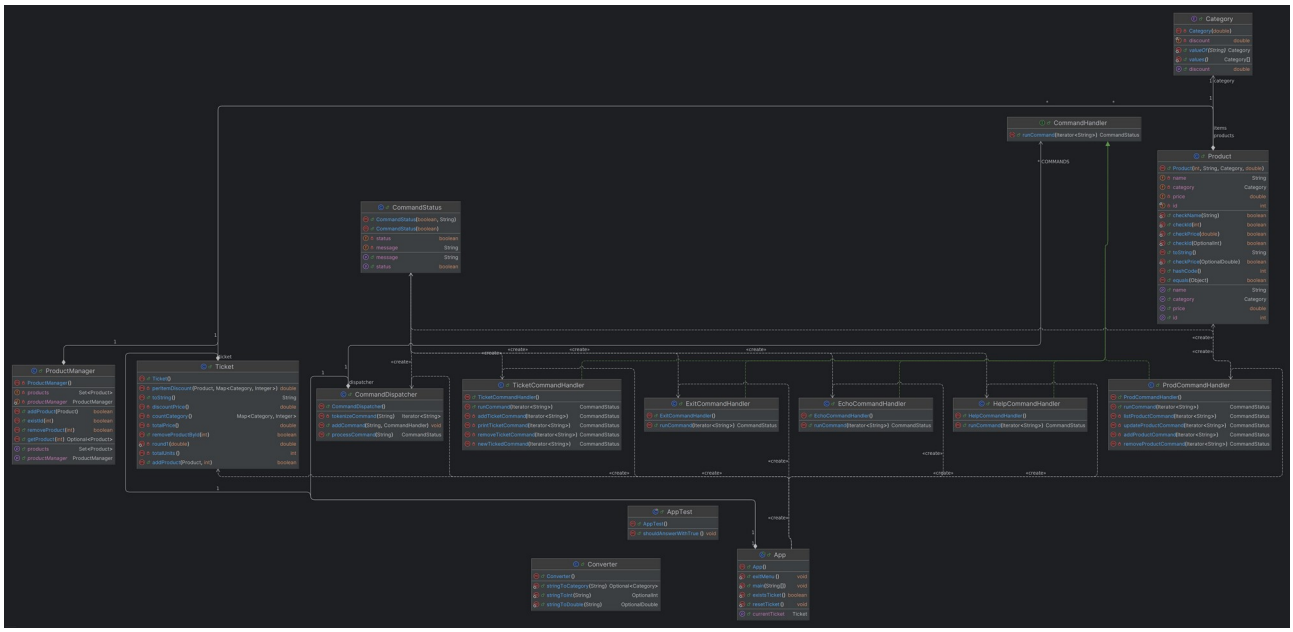
UPM SHOP: Entrega 1 (12-10-25)

By

Iza Pérez Dimas, Timotei Lucian Poenaru, Sarah Soret El Harti, Ekaitz Otero Moreno



Explicación de los distintos importes de librerías utilizados

Clase App:

import java.io.FileNotFoundException; se utiliza para evitar crashes al intentar abrir ficheros que no se encuentren al buscarse, en este caso para crear un bloque try-catch alrededor del FileReader.

Ejemplo:

```
try {
   scanner = new Scanner(new FileReader(args[0]));
} catch (FileNotFoundException e) {
   System.err.println("Fichero " + args[0] + " no encontrado.");
   scanner = new Scanner(System.in);
}
```

import java.io.FileReader; permite leer caracteres de un archivo de texto, en este caso para obtener los datos relativos al input de comandos de un archivo foráneo.

Ejemplo:

```
scanner = new Scanner(new FileReader(args[0]));
```

import java.util.Scanner; permite analizar datos pasados como input, en este caso tanto desde el fichero como la consola.

Ejemplo:

```
scanner = new Scanner(System.in);
```

Clase Ticket:

import java.util.*; permite usar las colecciones genéricas de Java (Map, List, Comparator, etc.), en este caso para gestionar los productos del ticket con un mapa que asocia cada producto a su cantidad, ordenar el ticket al imprimirlo y manejar utilidades como Optional.

Ejemplo:

```
private final Map<Product, Integer> items = new LinkedHashMap<>();
List<Entry<Product, Integer>> ordenados = new ArrayList<>(items.entrySet());
ordenados.sort(Comparator.comparing(e -> e.getKey().getName(), String.CASE_INSENSITIVE_ORDER));
```

import java.util.Map.Entry; permite acceder a cada par clave–valor dentro de un mapa, en este caso para recorrer los productos del ticket y calcular totales y descuentos.

Ejemplo:

```
for (Entry<Product, Integer> e : items.entrySet()) {
    total += e.getKey().getPrice() * e.getValue();
}
```

Clase Product:

import java.util.Objects; se utiliza para generar de manera segura el hashCode del producto y comparar objetos con equals, en este caso para asegurar que dos productos se consideren iguales si tienen el mismo identificador.

Ejemplo:

```
@Override
public int hashCode() {
    return Objects.hashCode(this.getId());
}
```

import java.util.OptionalInt; permite manejar identificadores opcionales sin usar null, en este caso para validar de forma segura un ID que puede o no estar presente.

Ejemplo:

```
public static boolean checkId(OptionalInt id) {
    return id.isPresent() && checkId(id.getAsInt());
}
```

import java.util.OptionalDouble; permite manejar precios opcionales sin usar null, en este caso para validar de forma segura un precio que puede o no estar presente.

Ejemplo:

```
public static boolean checkPrice(OptionalDouble price) {
    return price.isPresent() && checkPrice(price.getAsDouble());
}
```

Clase ProductManager:

import java.util.HashSet; permite almacenar los productos en una colección que no admite duplicados, en este caso para garantizar que no haya productos repetidos en el gestor.

Ejemplo:

```
private Set<Product> products = new HashSet<>();
```

import java.util.Optional; permite devolver de manera segura un producto que puede o no existir, en este caso para la búsqueda por identificador sin retornar null.

Ejemplo:

```
public Optional<Product> getProduct(int id) {
    return products.stream().filter(p -> p.getId() == id).findFirst();
}
```

import java.util.Set; permite declarar la colección de productos como un conjunto genérico, en este caso para definir la interfaz de la colección y poder hacer copias defensivas.

Ejemplo:

```
public Set<Product> getProducts() {
    return new HashSet<>(this.products);
}
```

Clase Converter:

import java.util.OptionalDouble; permite devolver de manera segura un número decimal que puede o no existir, en este caso para la conversión de cadenas a double sin lanzar excepciones fuera del método.

Ejemplo:

import java.util.Optional; permite devolver de manera segura un objeto que puede o no existir, en este caso para la conversión de cadenas a categorías sin retornar null ni lanzar excepciones.

Ejemplo:

```
public static Optional<Category> stringToCategory(String string) {
    try {
        return Optional.of(Category.valueOf(string.toUpperCase()));
    } catch (IllegalArgumentException exception) {
        return Optional.empty();
    }
}
```

Clase CommandDispatcher:

import java.util.ArrayList; permite crear listas dinámicas para almacenar los tokens de un comando, en este caso se usa para recopilar todos los fragmentos del comando antes de procesarlos.

Ejemplo:

```
List<String> tokens = new ArrayList<>();
while (matcher.find()) {
    String token = matcher.group();
    ...
}
```

import java.util.Iterator; permite recorrer colecciones de manera secuencial, en este caso para iterar sobre los tokens de un comando y procesarlos uno a uno.

Ejemplo:

```
Iterator<String> tokens = tokenizeCommand(command);
if (!tokens.hasNext()) {
    return new CommandStatus(false, "No command detected");
}
```

import java.util.List; permite declarar listas de manera genérica, en este caso para definir la colección que almacena los tokens de un comando.

Ejemplo:

```
List<String> tokens = new ArrayList<>();
```

import java.util.Map; permite definir la estructura que asocia nombres de comandos con sus handlers, en este caso para buscar rápidamente el handler de un comando dado.

Ejemplo:

```
private final Map<String, CommandHandler> COMMANDS = new TreeMap<>();
COMMANDS.put(name, commandHandler);
```

import java.util.TreeMap; permite mantener los comandos ordenados por nombre, en este caso para que la colección de comandos tenga un orden alfabético determinista.

Ejemplo:

```
private final Map<String, CommandHandler> COMMANDS = new TreeMap<>();
```

import java.util.regex.Matcher; permite buscar coincidencias de patrones en cadenas de texto, en este caso para extraer los tokens de un comando según la expresión regular definida.

Ejemplo:

```
Matcher matcher = pattern.matcher(command);
while (matcher.find()) {
    ...
}
```

import java.util.regex.Pattern; permite definir patrones de búsqueda de texto mediante expresiones regulares, en este caso para tokenizar correctamente los comandos incluyendo manejo de comillas.

Ejemplo:

```
Pattern pattern = Pattern.compile("\"([^\"]+)\"|\\S+");
```

Clase ProdCommandHandler:

import java.util.Optional; permite manejar valores que pueden o no estar presentes de manera segura sin usar null, en este caso se usa para capturar resultados de conversiones y búsquedas de productos, evitando errores al intentar acceder a un valor inexistente.

Ejemplo:

```
Optional<Category> productCategory = Converter.stringToCategory(tokens.next());
if (productCategory.isEmpty()) {
    return new CommandStatus(false, "Invalid category");
}
product.setCategory(productCategory.get());
```

Varios de estos importes se repiten a lo largo de varias clases, pero dado que su uso es igual a lo largo de todas sus implementaciones solamente se explican la primera vez que aparecen.

Respecto al diseño de clases:

Este sistema de comandos es una plataforma modular que permite ejecutar instrucciones desde consola o archivo. La clase App recibe la entrada y la envía al CommandDispatcher, que tokeniza el texto, identifica el comando principal y ejecuta su CommandHandler correspondiente. Cada handler implementa su propia lógica y devuelve un CommandStatus con el resultado, que luego se muestra en consola. El diseño facilita agregar nuevos comandos y mantener una estructura clara y ordenada. Además, usar Iterator<String> en lugar de arrays evita errores por índices y hace el procesamiento más flexible y seguro.