Diseño y Programación O.O:
Justificación de modificaciones (Taller 01)

Camilo Morillo y Juan Sebastián Ortega Romero

Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes

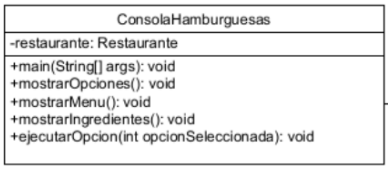
1 de septiembre de 2021

I. Aplicación Inicial (Original)

En general, consideramos que diseño propuesto originalmente para la aplicación tiene grandes ventajas estructurales por lo este fue completamente respetado durante el desarrollo del taller. Las modificaciones que se verán a continuación se centran en resolver problemas específicos obviados por el diseño original. Sin embargo, NO se le realizó ninguna modificación a la jerarquía de clases, al comportamiento general de las mismas ni a las relaciones entre objetos.

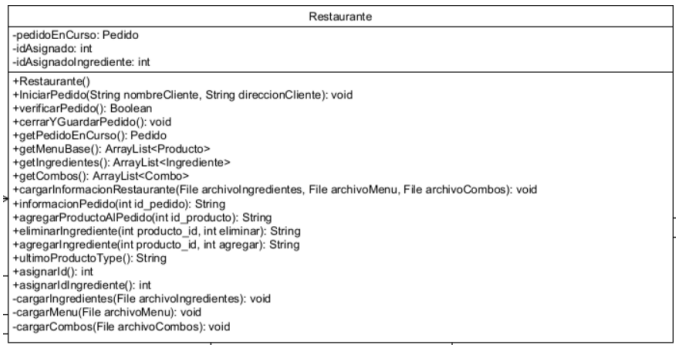
i. Segmentación y cambio de nombre: clase Aplicación

En primera instancia se decidió realizar un cambio superficial del nombre de la clase aplicación, pasando a ser la clase *ConsolaHamburguesa*. El motivo de este cambio no es más que establecer de forma concisa la distinción conceptual que tiene está clase, puesto que es la única que interactúa directamente con el usuario. En adición, se decidió segmentar esta clase en múltiples métodos que separan la información que se le muestra al usuario por productos (mostrarMenu()) e ingredientes (mostrarIngredientes()), así como se instanció el método mostrarOpciones() para distinguir con mayor facilidad las opciones que se le muestran al usuario. Cabe aclarar que todas las funciones anteriormente mencionadas siguen manteniendo un retorno de tipo *void*, esto con el objetivo de que su única función sea recibir o mostrar datos directamente en la terminal, respetando así el patrón modelo, vista, controlador. En adición, se instanció el atributo *restaurante* de tipo Restaurante, este atributo es fundamental puesto que es el puente de comunicación con el paquete que contiene el *modelo*.



ii. Atributos y métodos adicionales: clase Restaurante

La clase restaurante, al ser el intermediario entre la consola y el procesamiento de datos, debe contener los métodos necesarios para que la primera le asigne instrucciones a la segunda. Sin embargo, con los métodos planteados inicialmente, ciertas instrucciones específicas como agregar o eliminar ingredientes del producto actual no estaban contempladas. Con esto en mente, se construyeron algunos métodos más específicos para la necesidades del usuario que debían ser comunicadas al procesamiento. Algunos de estos nuevos métodos son *agregarIngrediente()*, *agregarProducto()* y *eliminarIngrediente()*. En adición, se pedía que los productos e ingredientes pudieran ser seleccionados mediante un ID (no proporcionado) y no por su nombre completo. Para lograr satisfacer esta necesidad se crearon los atributos *idAsignado* e *idAsignadoIngrediente* (junto con el método *asignarId()*) para que en el momento de cargar los datos, los objetos pudieran ser construidos con un ID asociado.



iii. Obtención de Id: clase ProductoMenu, Combo e Ingrediente

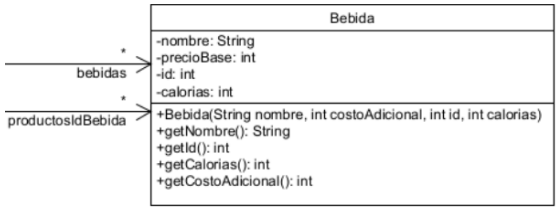
Para lograr que los productos, combos e ingredientes pudieran tener un ID único (como se vio en el punto anterior), fue necesario modificar los parámetros del método constructor de dichas clases añadiendo un parámetro extra: el *ID*. En conjunto a este parámetro se añadió el método *getId()* pero el funcionamiento y estructura general de estas clases por fuera de esta modificación permaneció exactamente igual.

II. Aplicación Modificada

Sobre las modificaciones vistas en la sección anterior se construyeron nuevas clases, métodos y atributos para lograr satisfacer los requerimientos de la versión modificada.

i. Implementación de bebidas por separado: clase Bebida

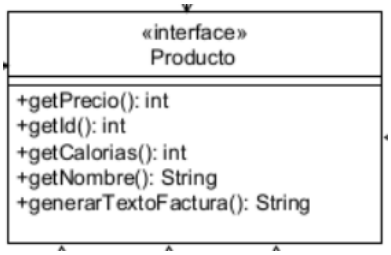
En la versión original de la aplicación, las bebidas se encuentran dentro del archivo general de productos. Así mismo, estas son cargadas a la par que los demás productos por lo que terminan siendo añadidas como objetos de tipo ProductoMenu. En la modificación, se separaron las bebidas del archivo de carga inicial y se añadieron a un nuevo archivo exclusivo para estas. Con el objetivo de modelar las bebidas completamente por separado, se decidió que estas debían pertenecer a una clase única pero que también compartiera métodos con los demás productos. De esta forma, se plantea una nueva clase *Bebida* que implemente la ya conocida interfaz *Producto*. Esta nueva clase del procesamiento es homóloga a la clase ProductoMenu, manteniendo sus métodos y atributos. En adición, para asegurar la compatibilidad con esta nueva clase, se crean métodos de acceso en Restaurante (Ej. getBebidas()) que permiten manipular las bebidas y hacerlas parte del pedido final. Así mismo, la consola ahora cuenta con una opción reservada para añadir bebidas.



ii. Creación de atributos y métodos para las calorías

La segunda modificación debía añadir funcionalidades con tal de que el cliente pudiera ver la cantidad total de calorías que iba a consumir al recibir la factura de su pedido. Puesto que los datos iniciales no incluían ningún campo referente a las calorías, estas fueron asignadas manualmente a bebidas, productos e ingredientes individuales. Debido a esta nueva característica se formularon los atributos de *calorias* así como el método de *getCalorias()* en todas las clases que implementan la clase *Producto*. El modelo de

conteo de calorías para los combos y el total del pedido se realizó de forma homóloga a como se sumaron los precios.



iii. Verificación de la existencia de pedidos idénticos

La última modificación requería que se le notificase al cliente en dado caso que una orden idéntica a la suya ya hubiera sido pedida con anterioridad, en otras palabras, se debía poder detectar pedidos que contuvieran exactamente los mismos objetos que implementan Producto. Para realizar esta comparación se creó el método *verificarPedidoRepetido()* en la clase Restaurante. Puesto que los objetos de tipo Pedido siempre iban a tener asignado un nombre, dirección e ID diferente, era imposible realizar la comparación a ese nivel, por lo que se decidió realizarla al nivel de los objetos de tipo producto contenidos dentro del pedido por medio del método *getItemsPedido()*. Este método tiene la particularidad de retornar un objeto de tipo *ArrayList* cuyos elementos internos son de tipo *Producto*, es decir, una lista de Productos. Por lo tanto, al comparar la lista obtenida del pedido actual con la lista obtenida por todos los demás pedidos almacenados anteriormente, sería posible evaluar una coincidencia. Esta coincidencia se daría únicamente cuando todos los productos dentro de la lista del pedido actual también estén contenidos por una lista de algún pedido anterior. Para esto se utilizó el método *<ArrayList<T>.containsAll(<ArrayList<T>)* de java.util, ya que en el interior de este método se utiliza el método *.equals()* encargado de verificar si dos objetos son exactamente iguales, cumpliéndose así el propósito de la modificación.

```
public boolean contains(Object o) {
    Iterator<E> it = iterator();
    if (o==null) {
        while (it.hasNext())
            if (it.next()==null)
                return true;
    } else {
        while (it.hasNext())
            if (o.equals(it.next()))
                return true;
    }
    return false;
}
```