Fase II - Desarrollo Móvil
KOTLIN FUNDAMENTALS

# PROYECTO: MOMO PLANTS

Kotlin Fundamentals

Integrantes Equipo 3:
Canchola Jiménez Orlando
García Barajas Luis Manuel
Mejía Toledo Guadalupe Estefanía
Morales Tavera Jonatan Arturo
Suárez Pinto Sergio Rafael

# Documentación Momo Plants



# FASE II DESARROLLO MÓVIL

**Kotlin Fundamentals**

**Proyecto:**
Momo Plants


**Equipo 3:**

Jonatan Arturo Morales Tavera

Guadalupe Estefanía Mejía Toledo

Luis Manuel García Barajas

Orlando Canchola Jiménez

Sergio Rafael Suarez Pinto

# Definición del proyecto

Momo Plants es un proyecto de comercio electrónico que tiene como objetivo ofrecer una aplicación móvil para la venta de plantas sin un espacio físico de ventas. La solución se enfoca en facilitar y simplificar la compra desde cualquier lugar mediante un dispositivo móvil.

En la presente fase, el proyecto se ha desarrollado utilizando el entorno de desarrollo integrado (IDE) Intellij Idea, con una implementación en consola.

# Requerimientos

- Conocimientos adquiridos durante las sesiones 1-8.
- (IDE) IntelliJ instalado para desarrollar el proyecto.
- Cada integrante cuente con GitHub para contribuir en el repositorio “Momo_Plants”.
- Cuenta en Notion para el desarrollo del proyecto: Diagramas de flujo, UML, Funciones en general, Documentación.

# Sesión 1

## Reto final: Definición del proyecto

Después de llevar a cabo una encuesta, se ha determinado que el proyecto a desarrollar consistirá en un sitio web de comercio electrónico especializado en la venta de plantas, el cual llevará el nombre de "Momo Plants". La elección de este proyecto se basó en las lecciones aprendidas durante la fase anterior, en la cual se utilizaron distintos proyectos realizados en la plataforma Figma.

https://www.figma.com/file/VCh9VcZeRNLbyTf4KMsAR7/Actividades-Estefania?node-id=208%3A1208&t=nldVYVFUWyL1i02U-1

Establecimos las diferentes funcionalidades de la app a desarrollar, las cuales fueron:

- Menú principal
  - Crear cuenta
  - Iniciar sesión
  - Cerrar sesión
  - Agregar planta al carrito
  - Ver carrito
  - Buscar planta por nombre
  - Ver historial de pedidos
  - Cambiar cantidad
  - Eliminar planta

**Postwork**

- Abstraer el concepto de ciclos y condicionales.
- Manejar de forma básica el IDE.
- Aplicar variables y realizar operaciones sobre ellas.

Algunas de las variables que se definieron para ser utilizadas son las siguientes:

`Username`: Almacena el nombre del usuario.

```
print("Nombre de usuario: ")
val username = readln()
```

`Password`: Almacena la contraseña del usuario.

```
print("Contraseña: ")
val password = readln()
```

`OrderList`: Sera la encargada de guardar el listado de ordenes.

```
private val orderList = mutableListOf<OrderEntity>()
```

`NameP`: Será el encargado de almacenar el nombre una planta de la cual se realizara una búsqueda.

```
print("Ingresa el nombre de la planta a buscar: ")
val nameP = readlnOrNull()?.trim()
```

`Quantity`: Sera el encargado de almacenar el numero de plantas que vamos a agregar a nuestro pedido.

```
print("Cantidad nueva de plantas que deseas agregar: ")
            val quantity = readlnOrNull()?.toIntOrNull()
```

`IndexP`: Será el ID de la planta de la cual vamos a modificar la cantidad dentro de nuestro carrito

```
print("Ingresa el id de la planta a cambiar cantidad: ")
val indexP = readlnOrNull()?.trim()?.toIntOrNull()
```

# Sesión 2

## Postwork - Maneja y controla el flujo de datos

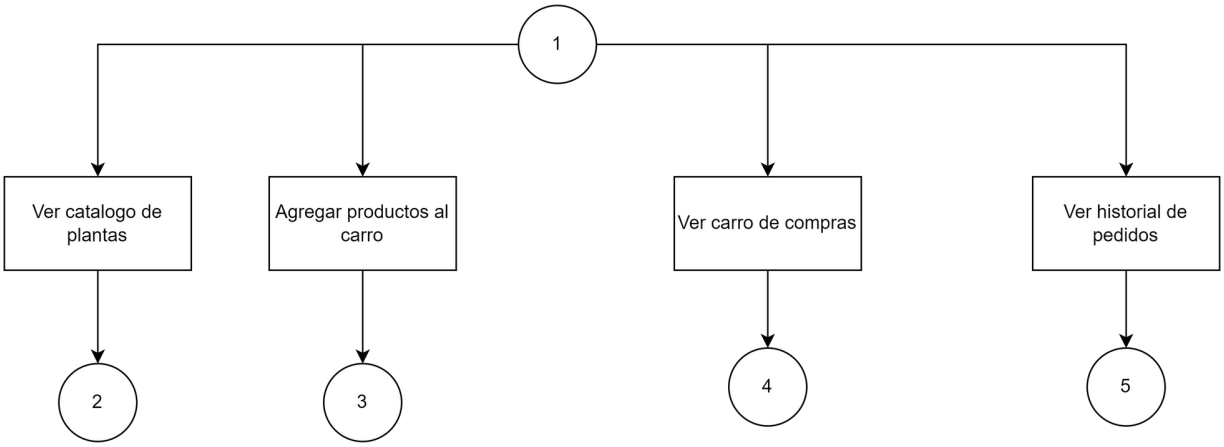
Con el propósito de llevar a cabo las funciones requeridas, se elaboraron diagramas de flujo con el propósito de brindar una representación visual y clara acerca de la secuencia de operaciones que se requerían para cada tarea específica. Los diagramas de flujo permitieron una mejor comprensión del proceso de trabajo y facilitaron la identificación de posibles problemas o ineficiencias en la ejecución de las tareas. De esta manera, se logró una planificación más efectiva y un mayor control sobre el desempeño de cada tarea en particular.

1. **Menu**

Es una función llamada "showMenu()" que tiene la finalidad de mostrar un menú de opciones al usuario. Este menú está representado por una lista de opciones, cada una de las cuales se representa como un par de valores. El primer valor del par es un número entero que representa la opción del menú, mientras que el segundo valor del par es una cadena de texto que representa la descripción de la opción del menú.

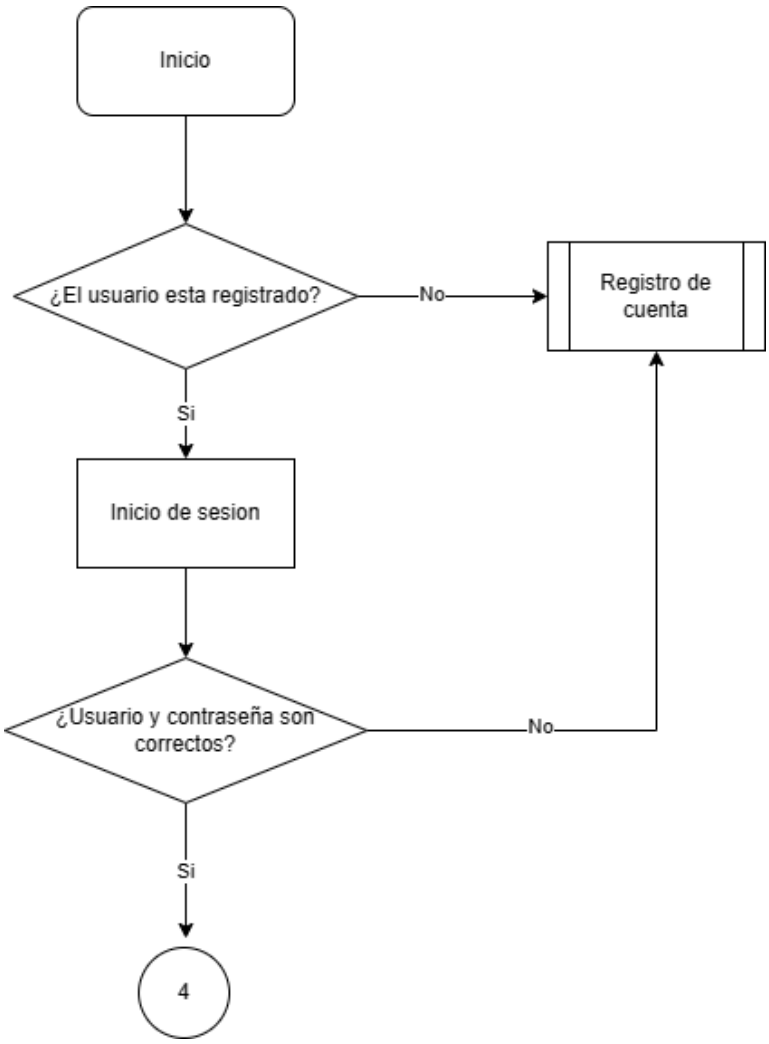
Dentro de la función, se define la lista "menu" que contiene todas las opciones del menú. Cada opción se representa como un par de valores utilizando la clase Pair. Luego, se llama a la función "forEach" de la lista "menu" para imprimir cada opción en la pantalla.

Dentro del bloque de la función "forEach", se define un parámetro llamado "option" que representa cada elemento de la lista "menu". Para cada elemento, se imprime en la pantalla un mensaje que incluye el número de la opción y su descripción. El número de la opción se obtiene utilizando la propiedad "first" del objeto "option", mientras que la descripción de la opción se obtiene utilizando la propiedad "second" del objeto "option".




```
fun showMenu() {
        val menu = listOf(
            Pair(1,"Agregar planta al carrito"),
            Pair(2,"Buscar planta"),
            Pair(3,"Eliminar planta"),
            Pair(4,"Mostrar plantas del carrito"),
            Pair(5,"Mostrar plantas del catálogo"),
            Pair(6,"Finalizar Pedido"),
            Pair(7,"Ver pedidos anteriores"),
            Pair(8,"Salir")
        )
        return menu.forEach { option-> println("${option.first}.- ${option.second}") }
    }
```

2. **Login**




- SignIn: Es función llamada "screenSignIn()" que tiene la finalidad de solicitar al usuario sus credenciales de inicio de sesión para acceder al sistema. La función comienza imprimiendo un mensaje en pantalla para que el usuario sepa que se espera que ingrese sus credenciales.

  A continuación, la función solicita al usuario su nombre de usuario y contraseña mediante las funciones "print" y "readln". Los valores ingresados por el usuario se almacenan en las variables "username" y "password", respectivamente.

  Una vez que se tienen las credenciales del usuario, se llama a la función "login" del objeto "servicio", que se espera que sea un objeto que contenga la lógica de autenticación del

sistema. La función "login" toma como parámetro una instancia de "UserEntity" que contiene el nombre de usuario y la contraseña ingresados por el usuario.

Si la función "login" del objeto "servicio" autentica al usuario, se imprime un mensaje de inicio de sesión exitoso en pantalla y se llama a la función "screenMenu()", que se espera que muestre un menú de opciones al usuario.

Después de esto, la función "screenSignIn()" llama a las funciones "sleep()" y "cleanScreen()", que se espera que realicen ciertas tareas para mejorar la presentación de la pantalla, como pausar la ejecución del programa durante un breve momento y borrar el contenido de la pantalla para ocultar las credenciales ingresadas por el usuario.

```
private fun screenSignIn() {
        println("Ingresa tus credenciales para iniciar sesión")
        print("Nombre de usuario: ")
        val username = readln()
        print("Contraseña: ")
        val password = readln()
        servicio.login(UserEntity(username, password))
        if (servicio.authenticatedUser) {
            println("Inicio de sesión exitoso!")
            screenMenu()
        }
        sleep()
        cleanScreen()
    }
```

- SignUp: De esta acción se encarga una función llamada "screenSignUp()" que tiene la finalidad de permitir al usuario crear una cuenta en el sistema para poder iniciar sesión en él. La función comienza imprimiendo un mensaje en pantalla para que el usuario sepa que se espera que cree una cuenta.

  A continuación, la función solicita al usuario su nombre de usuario mediante la función "print" y "readln". El valor ingresado por el usuario se almacena en la variable "username".

  Después, se verifica si el nombre de usuario ingresado ya existe en el sistema utilizando la función "userExist" del objeto "servicio". Si el nombre de usuario ya existe, se imprime un mensaje en pantalla informando al usuario que ese nombre de usuario ya está en uso. Si el nombre de usuario no existe, se solicita al usuario su contraseña mediante las funciones "print" y "readln", y se registra la nueva cuenta de usuario en el sistema utilizando la función "userRegister" del objeto "servicio" que toma como parámetro una instancia de "UserEntity" que contiene el nombre de usuario y la contraseña ingresados por el usuario.

  Finalmente, la función "screenSignUp()" llama a las funciones "sleep()" y "cleanScreen()", que se espera que realicen ciertas tareas para mejorar la presentación de la pantalla, como

pausar la ejecución del programa durante un breve momento y borrar el contenido de la pantalla para ocultar las credenciales ingresadas por el usuario.

```
private fun screenSignUp() {
        println("Crea una cuenta para iniciar sesión")
        print("Nombre de usuario: ")
        val username = readln()
        if (servicio.userExist(username)) {
            println("Ese nombre de usuario ya está en uso. Inténtalo de nuevo.")
        } else {
            print("Contraseña: ")
            val password = readln()
            servicio.userRegister(UserEntity(username, password))
            println("Cuenta creada exitosamente! Ahora puedes iniciar sesión.")
        }
        sleep()
        cleanScreen()
    }
```

3. **Buscar planta**

De esta acción se encarga una función llamada `askPlantToFind()` que permite al usuario buscar una planta en el carrito de compras mediante su nombre. El proceso comienza imprimiendo un mensaje en pantalla solicitando al usuario que ingrese el nombre de la planta que desea buscar.

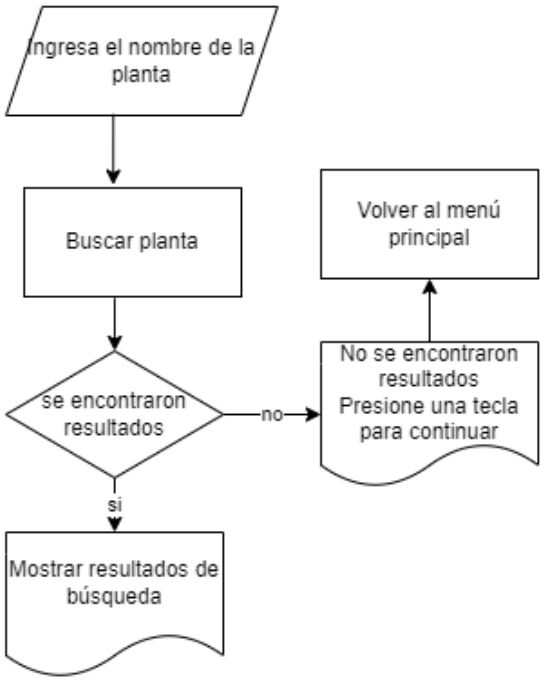
El nombre de la planta ingresado por el usuario se almacena en la variable `nameP`. Se utiliza `readlnOrNull()?.trim()` para leer el nombre de la planta ingresado por el usuario y, en caso de que no se ingrese nada, se asigna el valor nulo.

Después, se verifica si el valor ingresado por el usuario es diferente de nulo. Si es así, se busca la planta en el carrito utilizando la función `findItem()`. Si se encuentra el elemento en el carrito, se imprime un mensaje en pantalla indicando que la planta ha sido encontrada en el carrito, de lo contrario, se imprime un mensaje indicando que la planta no se encuentra en el carrito.

Si el valor ingresado por el usuario es nulo, se imprime un mensaje en pantalla indicando que los datos ingresados son inválidos.

En resumen, esta función brinda una funcionalidad básica para buscar plantas en el carrito de compras, utilizando el nombre de la planta como criterio de búsqueda.

Diagrama de flujo para Buscar Planta




```
fun askPlantToFind() {
        print("Ingresa el nombre de la planta a buscar: ")
        val nameP = readlnOrNull()?.trim()
        if (nameP != null) {
            val item = findItem(nameP)

            if (item != null) {
                println("La planta ${item.plant.name} se encuentra en el carrito.")
            } else {
                println("La planta $nameP no se encuentra en el carrito.")
            }
        } else {
            println("Error: datos ingresados inválidos.")
        }
    }
```

4. **Agregar compras al carrito**

De esta acción se encarga una función llamada `askPlantToAdd()` que permite al usuario agregar una planta al carrito de compras.

Primero, se solicita al usuario que ingrese el identificador único (id) de la planta que desea agregar. El id ingresado se lee desde la entrada estándar utilizando la función `readlnOrNull()`. Luego, se verifica si el id es válido (es decir, es un número entero y se encuentra dentro de los límites del índice de la lista de plantas).

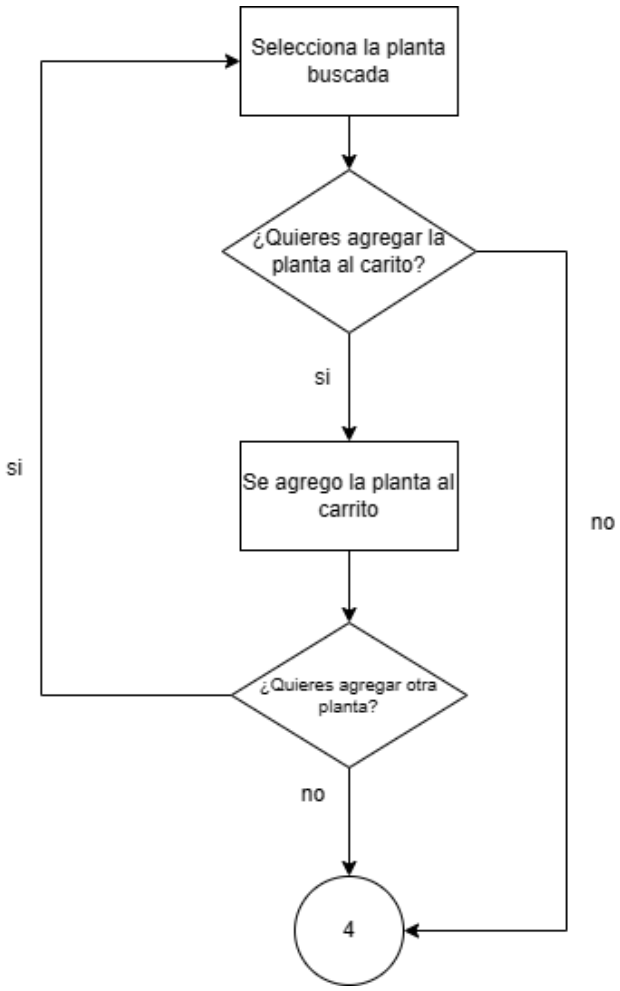
Si el id es válido, se solicita al usuario que ingrese la cantidad de plantas que desea agregar al carrito. La cantidad ingresada se almacena en la variable `quantity`.

Si la cantidad ingresada es válida (es decir, es un número entero mayor que cero), se crea una instancia de la clase `ItemEntity` que contiene la planta seleccionada y la cantidad ingresada, y se agrega al carrito mediante la función `addItem()`. Finalmente, se imprime un mensaje en pantalla confirmando que se ha agregado la cantidad especificada de plantas al carrito.

Si la cantidad ingresada no es válida, se imprime un mensaje en pantalla indicando que la cantidad es inválida.

Si el id ingresado es inválido, se imprime un mensaje en pantalla indicando que los datos son inválidos.

En resumen, esta función brinda una funcionalidad básica para agregar plantas al carrito de compras mediante la entrada del usuario, y proporciona mensajes de error y confirmación según corresponda.




```
fun askPlantToAdd() {
        print("Ingresa el id de la planta: ")
        val id: Int? = readlnOrNull()?.trim()?.toIntOrNull()
        if (id != null && id in plants.indices) {
            print("Cantidad de plantas que deseas agregar: ")
            val quantity = readlnOrNull()?.toIntOrNull()
            if (quantity != null && quantity > 0) {
                val plant = plants[id]
                // TODO
                addItem(ItemEntity(plant, quantity))
                println("Se agregaron $quantity planta(s) de ${plant.name} al carrito.")
            } else {
                println("Error: cantidad invalida.")
            }
        } else {
            println("Error: Datos invalidos")
        }
    }
```

5. **Eliminar plantas del carrito**

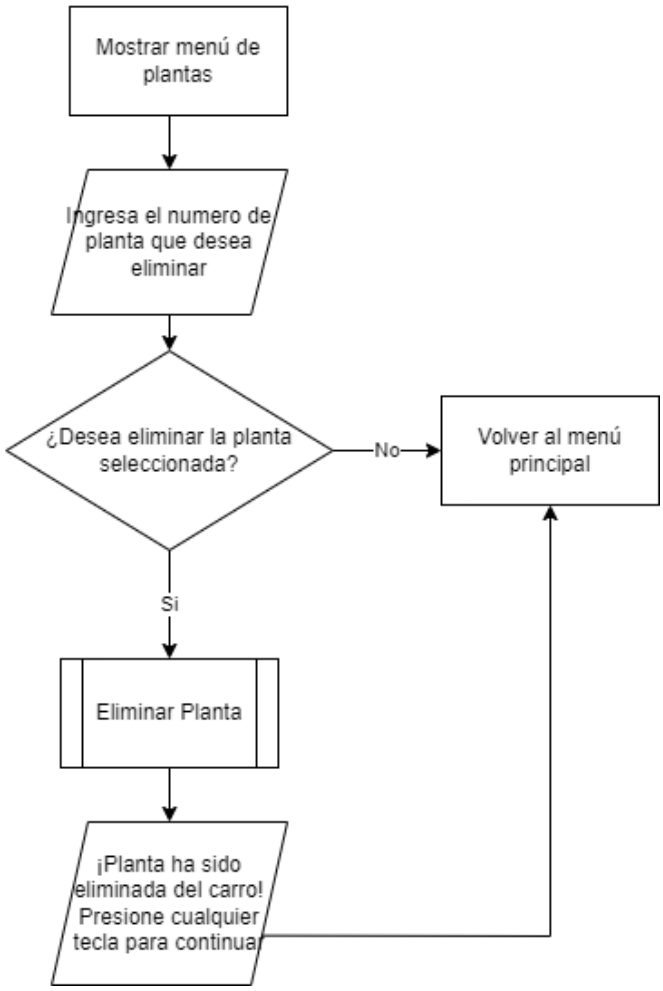
De esta acción se encarga una función `askPlantToRemove()`, que se utiliza en una aplicación de carrito de compras de plantas para permitir al usuario eliminar una planta del carrito.

El código comienza pidiéndole al usuario que ingrese el ID de la planta que desea eliminar. Luego, verifica si el ID ingresado es válido y está dentro del rango de índices de elementos del carrito de compras. Si es así, el código muestra un mensaje preguntando si el usuario realmente desea eliminar la planta del carrito y presenta dos opciones: "1.- Sí, eliminar" o "2.- Cancelar".

Si el usuario selecciona la opción "1", la función llama a la función `removeItem()` para eliminar la planta del carrito y muestra un mensaje confirmando que la planta ha sido eliminada. Si el usuario selecciona cualquier otra opción o no ingresa una opción válida, la función simplemente muestra un mensaje que le pide al usuario que presione Enter para continuar.

Si el ID de la planta ingresado no es válido o no está dentro del rango de índices de elementos del carrito de compras, la función muestra un mensaje indicando que no existe el ID.

Diagrama de Flujo para eliminar plantas de carrito



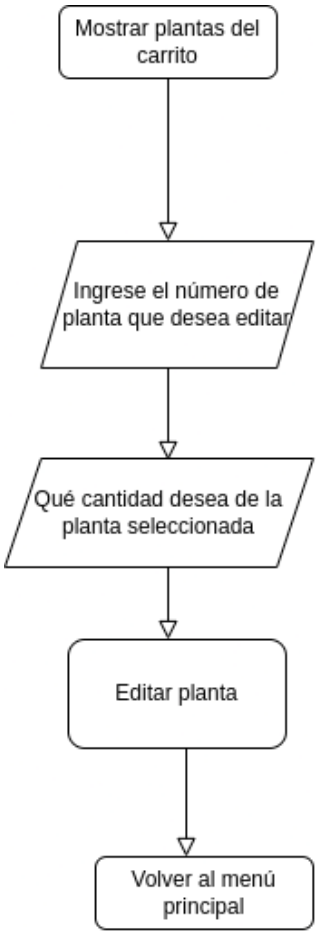

```
fun askPlantToRemove() {
        print("Ingresa el id de planta que deseas eliminar: ")
        val indexP = readlnOrNull()?.trim()?.toIntOrNull()
        if (indexP != null && indexP in 0 until shoppingCart.items.size){
            val itemToRemove = shoppingCart.items[indexP]
            println("¿Seguro que desea eliminar la planta ${itemToRemove.plant.name} del carrito ?")
            println("1.- Sí, eliminar")
            println("2.- Cancelar")
            when(readlnOrNull()?.trim()?.toIntOrNull()){
                1 -> {
                    removeItem(indexP)
                    println("${itemToRemove.plant.name} se ha eliminado del carrito")
                }
                else -> println("Presiona enter para continuar")
            }
        }else{
            println("No existe el id")
```

```
        }
    }
```

6. **Editar carrito**

De esta acción se encarga una función `updatePlant()` comienza por solicitar al usuario que ingrese el ID de la planta que desea actualizar en términos de cantidad. Si el ID ingresado es válido, es decir, está en el rango de índices de las plantas en el carrito, se solicita al usuario que ingrese la nueva cantidad de plantas que desea agregar. Si la cantidad ingresada es mayor que cero, se actualiza la cantidad de la planta en el carrito y se muestra un mensaje que indica que se ha agregado la cantidad especificada de plantas al carrito.

Si el ID ingresado no es válido, se muestra un mensaje de error que indica que no existe el ID.




```
private fun updatePlant(){
        print("Ingresa el id de la planta a cambiar cantidad: ")
        val indexP = readlnOrNull()?.trim()?.toIntOrNull()
        if (indexP != null && indexP in 0 until shoppingCart.items.size){
```

```
            print("Cantidad nueva de plantas que deseas agregar: ")
            val quantity = readlnOrNull()?.toIntOrNull()
            if (quantity != null && quantity > 0) {
                val plant = plants[indexP]
                updateItem(ItemEntity(plant, quantity), indexP)
                println("Se agregaron $quantity planta(s) de ${plant.name} al carrito.")
            } else {
                println("Error: cantidad invalida.")
            }
        }else{
            println("No existe el id")
        }
    }
```

7. **Ver ordenes anteriores**

De esta acción se encarga una función llamada `showOldOrders` que se encarga de mostrar las órdenes antiguas que el usuario ha realizado.

La función comienza verificando si la lista de órdenes (`orderList`) está vacía. Si está vacía, la función imprime "No hay ordenes antiguas." Si la lista no está vacía, la función itera a través de cada elemento de la lista utilizando el método `forEach`.

Para cada orden, la función itera a través de los elementos de la lista de artículos de esa orden utilizando el método `forEachIndexed`. Dentro de este bucle, la función imprime el índice del artículo, el nombre de la planta y la cantidad del artículo en la orden.

Finalmente, después de mostrar todas las órdenes antiguas, la función llama a otra función `menuShow` para volver a mostrar el menú principal.

```
fun showOldOrders(){
        if (orderList.isEmpty()) {
            println("No hay ordenes antiguas.")
        } else {
            println("Plantas en el carrito:")
            orderList.forEach {
                it.items.forEachIndexed { index, item ->
                    println("Id: ${index}, Nombre: ${item.plant.name}, Cantidad: ${item.quantity}")
                }
            }
            menuShow()
        }
    }
```

8. **CheckOut**

Se define la función `checkOut()` que se encarga de procesar la orden actual del usuario.

En primer lugar, la función calcula el número total de plantas en el carrito sumando la cantidad de cada item. Luego, muestra un mensaje al usuario preguntando si desea finalizar su pedido o cancelarlo, o volver al menú principal.

El usuario ingresa su elección a través de la entrada de la consola, y luego se utiliza un bloque `when` para determinar la acción a tomar en función de su elección. Si el usuario selecciona la opción 1, la función llama a la función `updatedStatus()` para actualizar el estado de la orden a "pagado". Si el usuario selecciona la opción 2, la función llama a `updatedStatus()` para establecer el estado de la orden como "cancelado". Si el usuario selecciona la opción 3, la función llama a `showMenu()` para mostrar el menú principal. Si el usuario ingresa una opción no válida, se muestra un mensaje de error.

```
fun checkOut(){
        val total = shoppingCart.items.sumOf { it.quantity }
        println("Su pedido actualmente tiene $total items")
        println("¿Desea finalizar su pedido?")
        println("1. Finalizar Pedido")
        println("2. Cancelar Pedido")
        println("2. Volver al menú Principal")
        println("Ingresa una opción")
        when (readlnOrNull()?.toIntOrNull()) {
            1 -> updatedStatus(OrderStatus.PAID)
            2 -> updatedStatus(OrderStatus.CANCELED)
            3 -> showMenu()
            else -> println("Opción inválida. Inténtalo de nuevo.")
        }
    }
```

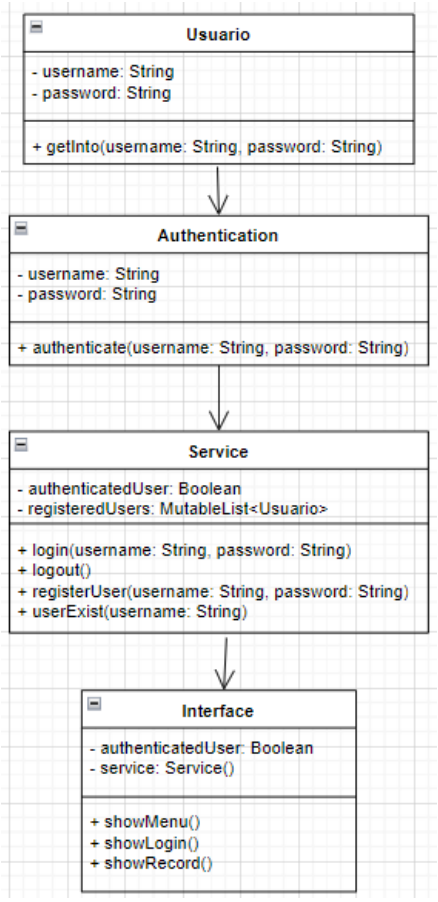
# Sesión 3

## Postwork - Programación orientada a objetos

La programación orientada a objetos (POO) se utiliza en toda la aplicación para llevar a cabo diversas actividades y gestionar adecuadamente los datos correspondientes. En este sentido, se ha aplicado POO para lograr una mayor modularidad, facilidad de mantenimiento y escalabilidad del sistema.

En el marco del desarrollo de la aplicación, se ha diseñado un diagrama UML que contempla la conexión de cuatro clases que trabajan conjuntamente para llevar a cabo el proceso de registro e inicio de sesión de los usuarios. Dicho diagrama ha sido desarrollado siguiendo las mejores

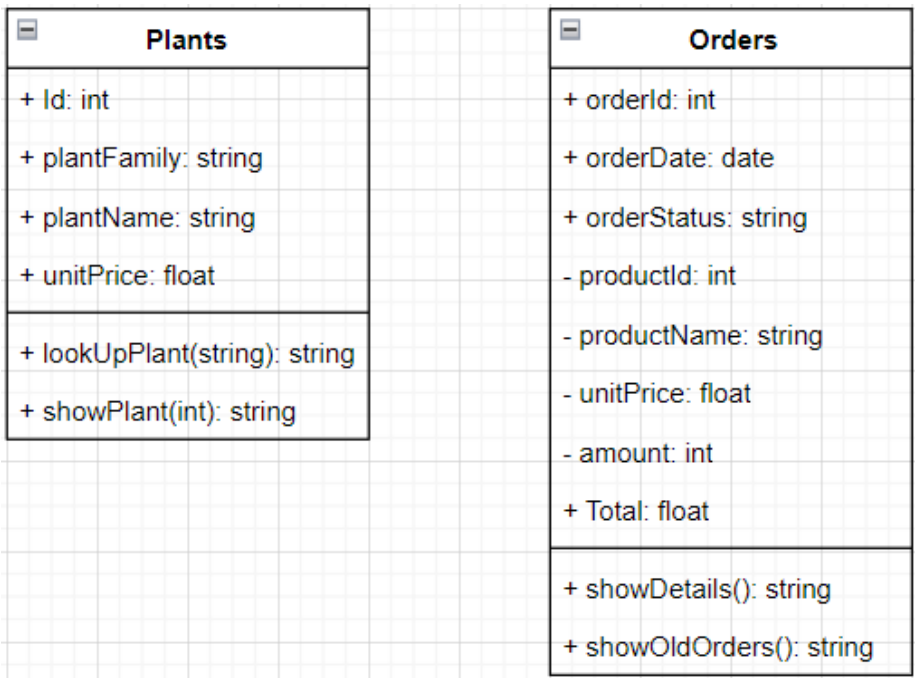
prácticas y los principios de la ingeniería del software, y se ha utilizado como herramienta para guiar el proceso de desarrollo de la aplicación y asegurar la calidad del software producido.




En la clase Orders se han definido variables para almacenar información importante sobre los pedidos, como su identificador, estado actual y los artículos que lo conforman. Además, se han implementado métodos para ver detalles y ver ordenes anteriores.

Por otro lado, en la clase Plants se han definido variables para almacenar información sobre las plantas disponibles en el sistema, como su identificador, nombre, familia de plantas y precio unitario. También se han implementado métodos para ver las plantas y sus detalles.

Este diagrama UML ha sido diseñado siguiendo las mejores prácticas de la ingeniería del software, y se ha utilizado como una herramienta para guiar el proceso de desarrollo de la aplicación y asegurar la calidad del software producido. Su implementación es fundamental para garantizar la modularidad, escalabilidad y facilidad de mantenimiento de la aplicación en su conjunto.




# Sesión 4

## Postwork - Fundamentos de programación

### Data Class

Utilizamos los data class para almacenar diferentes datos los cuales fueron los siguientes:

- ItemEntity: clase de datos en Kotlin llamada `ItemEntity`. Esta clase tiene dos propiedades: `plant` y `quantity`.

  La propiedad `plant` es un objeto de la clase `PlantEntity`, que representa una planta en la tienda. La propiedad `quantity` es un número entero que representa la cantidad de esa planta que el usuario ha agregado al carrito de compras.

  La clase de datos en Kotlin es una clase que se utiliza para almacenar datos, y se genera automáticamente el código de los métodos `equals()`, `hashCode()`, `toString()` y `copy()`. También puede incluir funciones personalizadas si es necesario.

  En este caso, `ItemEntity` se utiliza para representar los elementos en el carrito de compras del usuario y para rastrear la cantidad de cada planta que el usuario ha agregado.

```
data class ItemEntity(
    val plant: PlantEntity,
    var quantity: Int
)
```

- `OrderEntity:` Este código define una clase de datos llamada "OrderEntity" que representa una entidad de pedido en un sistema.

  La clase tiene tres propiedades:

  - "id" es un entero que representa el identificador del pedido. El valor predeterminado es 0.
  - "status" es una variable mutable de tipo "OrderStatus" que representa el estado del pedido. El valor predeterminado es "OrderStatus.PENDING", que es uno de los estados posibles del pedido.
  - "items" es una lista mutable de elementos de tipo "ItemEntity" que representa los artículos que se han pedido. El valor predeterminado es una lista vacía.

```
data class OrderEntity(
    val id: Int = 0,
    var status: OrderStatus = OrderStatus.PENDING,
    var items: MutableList<ItemEntity> = mutableListOf()
)
```

- `PlantEntity`: Este código define una clase de datos llamada "PlantEntity" que representa una entidad de planta en un sistema.

  La clase tiene siete propiedades:

  - "id" es un entero que representa el identificador de la planta. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.
  - "name" es una cadena de texto que representa el nombre de la planta. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.
  - "origin" es una cadena de texto que representa el lugar de origen de la planta. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.

- "weather" es una cadena de texto que representa el tipo de clima que necesita la planta para crecer. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.
- "format" es una cadena de texto que representa el formato o presentación de la planta (por ejemplo, maceta o enraizada). Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.
- "price" es un entero que representa el precio de la planta. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.
- "stock" es un entero que representa la cantidad de unidades en stock de la planta. Este valor es opcional (tiene un valor predeterminado de 0) y se puede proporcionar al construir una instancia de la clase, pero si no se proporciona, se establecerá automáticamente en 0.

```
data class PlantEntity(
    val id: Int,
    val name: String,
    val origin: String,
    val weather: String,
    val format: String,
    val price: Int,
    val stock: Int = 0
)
```

- `UserEntity:` Este código define una clase de datos llamada "UserEntity" que representa una entidad de usuario en un sistema.

  La clase tiene dos propiedades:

  - "username" es una cadena de texto que representa el nombre de usuario del usuario. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.
  - "password" es una cadena de texto que representa la contraseña del usuario. Este valor es obligatorio (no tiene un valor predeterminado) y debe ser proporcionado al construir una instancia de la clase.

```
data class UserEntity(
    val username: String,
    val password: String
)
```

# Sesión 5

## Postwork - Programación funcional

Se implementó la programación funcional ya que nos ayudó a dividir la mayor cantidad tareas posible en funciones, las cuales pueden ser usadas por otras tareas, las funciones que se realizaron son:

- finItemByName

Es una función interna, la cuál nos permite encontrar un Item mediante su nombre. La variable plantaName que es el nombre de la planta es tomado como un dato String “texto”.

```
internal fun findItemByName(plantName: String): PlantEntity? =
         shoppingCart.find { item -> item.name.lowercase().contains(plantName.lowercase()) }
```

- findItemByID

Es una función declarada interna, la cuál nos permite encontrar el Item por su Id. El id de la planta es tomado como entero.

```
internal fun findItemByID(idPlant: Int): PlantEntity? = plants.find { item -> item.id == idPlant }
```

---

# Sesión 6

## Postwork - Interoperabilidad Kotlin - Java

Se realizada una clase en Java la cual da la autenticación al usuario al ingresar contraseña y usuario en Login y muestra un mensaje si los datos son incorrectos: `println("Credenciales incorrectas")` en caso contrario de que sean correctos los datos se ingresa al menú.

La clase Service que se había implementado en Java se pasó al lenguaje Kotlin, añadiéndole más funcionalidades.

```
class Service {
    var authenticatedUser: Boolean = false
    private val registeredUsers = mutableListOf<UserEntity>()

    fun login(user: UserEntity) {
        if (registeredUsers.contains(user)) {
            authenticatedUser = true
        } else {
            println("Credenciales incorrectas")
        }
    }

    fun logout() {
        authenticatedUser = false
    }

    fun userRegister(user: UserEntity) {
        registeredUsers.add(user)
    }

    fun userExist(username: String): Boolean = registeredUsers.any { it.username == username }
}
```

**Paquetes Java**

Como parte de la interoperabilidad entre los lenguajes Kotlin y Java, utilizamos la importación de los siguientes paquetes:

```
import org.junit.jupiter.api.Assertions.*
import org.junit.jupiter.api.Test
```

Estos paquetes son utilizados para realizar tests al proyecto.

Una manera adicional en que hemos utilizado la interoperabilidad entre Java y Kotlin es mediante la clase Date contenida en los paquetes de Java. Para importarla, hemos empleado las siguientes directivas de importación:

```
import java.util.Date;
import java.text.SimpleDateFormat;
```

En el código escrito en Kotlin, utilizaremos estas clases de la siguiente manera:

```
private fun payToCart() {
        val now = Date(); // <-- Se crea una instancia de la clase Date
```

```
        val date = SimpleDateFormat("E yyyy.MM.dd 'at' hh:mm:ss a zzz") // <-- Se le aplica el formato
        println("Fecha actual:  ${date.format(now)}") // <-- Se imprime la fecha actual
        println("Su listado de plantas a comprar es el siguiente: ")
        println("------------------------------------------------------------")
        shoppingCart.forEach {
            println("Id: ${it.id}, Nombre: ${it.name}, Cantidad: ${it.quantity}, Precio: ${it.price}")
        }
        println("------------------------------------------------------------")
        println("Total a pagar: $${calcTotal()}")
        print("Presiona enter para continuar -> ")
        oldOrders.add(shoppingCart)
        shoppingCart.clear()
        readlnOrNull()?.toIntOrNull()
    }
```

---

# Sesión 7

## Postwork - Manejo de errores

Para manejar los diversos errores que se pudieran presentar en el proyecto, utilizamos el manejo de excepciones con el **try - catch.**

Se agrego el manejo de excepciones **try - catch** en las funciones `deleteFromCart()` y `addToCart()`.

- `deleteFromCart()`

  Esta función es utilizada para eliminar una compra del carrito, por lo cual al entrar a try ejecuta el código pidiendo la cantidad de plantas a eliminar y en dado caso que se presente un error el catch lo atrapa y muestra un mensaje en consola: `println("Error: $e").`

```
private fun deleteFromCart() {
        print("Ingresa el id de planta que deseas eliminar: ")
        val indexP = readlnOrNull()?.trim()?.toIntOrNull()
        val plant = shoppingCart.find { item -> item.id == indexP }
        if (plant == null) {
            println("Datos Invalidos")
            return
        }
        try {
            print("Cantidad de plantas que deseas disminuir: ")
            val quantity = readlnOrNull()?.toIntOrNull()
            if (quantity == null || quantity < 1) {
                println("Cantidad invalida")
                return
            }
            if (plant.quantity > quantity) {
                plant.quantity -= quantity
            } else {
                shoppingCart.remove(plant)
            }
```

```
        } catch (e: Exception) {
            println("Error: $e")
        }
    }
```

- `addToCart()`

  Esta funcion es para agregar una planta al carrito, en el momento de agregar la cantidad de dicha planta se pueden presentar errores, por lo utilizamos el try para ir testeando lo que hay dentro de este y si se presenta un error atraparlo dentro del catch, mostrando el mensaje de Exeption `“catch (e: Exception)”`.

```
fun addToCart() {
        print("Ingresa el id de planta que deseas agregar: ")
        val indexP = readlnOrNull()?.trim()?.toIntOrNull()
        val plant = findItemByID(indexP ?: -1)
        if (plant == null) {
            println("Datos Invalidos")
            return
        }
        try {
            print("Cantidad de plantas que deseas agregar: ")
            val quantity = readlnOrNull()?.toIntOrNull()

            if (quantity == null || quantity < 1) {
                println("Cantidad invalida")
                return
            }
            if (plant in shoppingCart) {
                shoppingCart.find { it == plant }?.let { it.quantity += quantity }
            } else {
                plant.quantity = quantity
                shoppingCart.add(plant)
            }
        } catch (e: Exception) {
            println("Error: $e")
        }
    }
```

**Operador Elvis**

Se utilizó el operar Elvis en la función llamada addToCart, la cuál agrega plantas al carrito:

```
val plant = findItemByID(indexP ?: -1)
```

---

# Sesión 8

## Postwork - Programación asíncrona

### Coroutines y Flow

Para agregar las Coroutines dentro del proyecto se agrega la siguiente dependencia:

- `implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.4"` se utiliza para agregar la biblioteca de Kotlin Coroutines al proyecto.

  Kotlin Coroutines es una biblioteca que proporciona una manera fácil y segura de escribir código asíncrono y concurrente en Kotlin. Permite escribir código en un estilo secuencial y fácil de leer, mientras que el motor de Coroutines se encarga de la ejecución asincrónica y concurrente de este código.

  La implementación "org.jetbrains.kotlinx:kotlinx-coroutines-core" agrega las clases y funciones principales de la biblioteca de Kotlin Coroutines al proyecto. Esto incluye funciones para crear y lanzar Coroutines, mecanismos de suspensión, métodos para controlar la ejecución y cancelación de Coroutines, y más.

  En resumen, la implementación "implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.4"" sirve para agregar la biblioteca de Kotlin Coroutines al proyecto, lo que permite escribir código asincrónico y concurrente de manera fácil y segura en Kotlin.

  En la sección de programación asíncrona, se ha implementado una simulación de solicitud a una API para obtener información sobre plantas. Para lograr esto, se han utilizado coroutines, lo que ha permitido realizar la solicitud de manera eficiente. El siguiente fragmento de código ejemplifica el proceso:

```
fun getPlantsByCoroutines() {
        runBlocking {
            PlantsDatabase.getAllPlants().forEach {
                plants.add(it)
                delay(100)
            }
        }
    }
```

Aquí esta la explicación linea por linea:

- `fun getPlantsByCoroutines() {`: Se define una función llamada `getPlantsByCoroutines` que no tiene parámetros de entrada.
- `runBlocking { ... }`: Se llama a la función `runBlocking` que bloquea el hilo actual hasta que se completen todas las tareas dentro del bloque de código. Esto es útil en este caso ya que estamos usando coroutines dentro de una función principal que no es suspendida.
- `PlantsDatabase.getAllPlants().forEach { ... }`: Se llama a la función `getAllPlants` en la base de datos de plantas (asumiendo que esto es una función que devuelve todas las plantas en la base de datos) y se itera sobre cada una de ellas.

- `plants.add(it)`: Se agrega cada planta a una lista llamada `plants`.
- `delay(100)`: Se hace una pausa de 100 milisegundos antes de procesar la siguiente planta. Esta pausa se realiza utilizando la función `delay`, que es una función suspendida que detiene la ejecución de la coroutine actual sin bloquear el hilo principal.

Y dicha función se manda a llamar al iniciar la primer vista de la aplicación:

```
fun screenSplash() {
        Cart.cart.getPlantsByCoroutines() // <----------- Obtencion de los datos
        println("Bienvenido a la aplicación!")
        var opcion: Int
        do {
            println()
            println("1. Iniciar sesión")
            println("2. Crear cuenta")
            println("3. Salir")
            print("Selecciona una opción: ")
            opcion = readln().toInt()
            sleep()
            cleanScreen()
            when (opcion) {
                1 -> screenSignIn()
                2 -> screenSignUp()
                3 -> println("Hasta luego!")
                else -> println("Opción inválida. Inténtalo de nuevo.")
            }
        } while (opcion != 3)
    }
```

---

# Extras

## Testing

El test se refiere a una clase o conjunto de clases que contienen código que verifica que el código de producción funciona según lo esperado.

Los tests se utilizan para comprobar que el código de producción (el código que se escribe para cumplir con los requisitos de una aplicación) funciona correctamente en diferentes situaciones, como cuando se ingresan diferentes valores de entrada o cuando se llaman a diferentes métodos.

En Kotlin, los tests se escriben utilizando herramientas como JUnit o KotlinTest. Estos frameworks proporcionan funciones y extensiones para escribir y ejecutar pruebas unitarias e integración de manera fácil y eficiente.

En IntelliJ IDEA, los tests se pueden crear como clases Kotlin normales con métodos de prueba definidos. Estos métodos de prueba se decoran con anotaciones específicas del framework de

pruebas utilizado, como @Test para JUnit. Cuando se ejecutan las pruebas, se verifica que el código de producción funcione según lo esperado y se informa cualquier problema o error encontrado.

En resumen, en un proyecto de Kotlin en IntelliJ IDEA, el test se refiere a una clase o conjunto de clases que contienen código que verifica que el código de producción funciona según lo esperado. Se utilizan herramientas como JUnit o KotlinTest para escribir y ejecutar pruebas de manera fácil y eficiente.

Se han añadido las siguientes implementaciones a un proyecto con el fin de llevar a cabo la prueba de las clases y garantizar la correcta funcionalidad de sus funciones.

Las implementaciones que se llevarán a cabo son las siguientes:

1. `implementation 'org.junit.jupiter:junit-jupiter:5.8.1'`: Agrega la dependencia de JUnit Jupiter en su versión 5.8.1 al proyecto. JUnit Jupiter es un popular framework de pruebas unitarias para Java y Kotlin que permite escribir pruebas de forma fácil y eficiente. Con esta implementación, se pueden agregar las funciones y anotaciones de JUnit Jupiter para definir y ejecutar pruebas unitarias e integración en el proyecto.
2. `testImplementation 'org.jetbrains.kotlin:kotlin-test'`: Agrega la dependencia de la biblioteca de pruebas de Kotlin al proyecto. La biblioteca de pruebas de Kotlin proporciona herramientas para escribir y ejecutar pruebas unitarias e integración en Kotlin. Con esta implementación, se pueden agregar las funciones y extensiones de KotlinTest para aserciones, manejo de excepciones y otras herramientas útiles para escribir pruebas en el proyecto.

Se agregaron dos clases de test las cuales son CartTest y ServiceTest.

▼ **CartTest**

```
@Test
    fun addItem_oneItemAdded_oneItemSize() {
        //Given
        val cart = Cart()
        val item = PlantsDatabase.getAllPlants()[0]
        cart.addItem(item)
        //When
        val result = cart.shoppingCart.size
        //Then
        assertEquals(1, result)
    }
```

A continuación, se detalla el funcionamiento del código:

- Dentro del método se definen tres secciones clásicas de una prueba unitaria: Given (Dado), When (Cuando) y Then (Entonces). Estas secciones indican lo que se espera

que suceda en la prueba.

- En la sección Given se crea un objeto de la clase `Cart` y se obtiene un ítem de la base de datos de plantas mediante la llamada a `PlantsDatabase.getAllPlants()[0]`.
- En la sección When se agrega el ítem obtenido a la instancia de `Cart` creada anteriormente mediante la llamada a `cart.addItem(item)`.
- En la sección Then se verifica que el tamaño de la instancia de `Cart` es igual a uno mediante la llamada a `assertEquals(1, result)`.
- Si el resultado de la prueba es satisfactorio, la prueba unitaria finalizará correctamente. De lo contrario, el framework de pruebas indicará el fallo y el desarrollador deberá corregir el código para que la prueba pase exitosamente.

```
@Test
    fun findItem_existsItem_returnsItem() {
        //Given
        val cart = Cart()
        val plantName = "Galatea"
        cart.apply {
            addItem(PlantsDatabase.getAllPlants()[8])
            addItem(PlantsDatabase.getAllPlants()[5])
            addItem(PlantsDatabase.getAllPlants()[10])
            addItem(PlantsDatabase.getAllPlants()[15])
            addItem(PlantsDatabase.getAllPlants()[20])
            addItem(PlantsDatabase.getAllPlants()[25])
            addItem(PlantsDatabase.getAllPlants()[30])
        }
        //When
        val result = cart.findItemByName(plantName)
        //Then
        assertEquals(ItemEntity(PlantsDatabase.getAllPlants()[8], 1), ItemEntity(result!!, 1))
    }
```

A continuación, se detalla el funcionamiento del código:

- Dentro del método se definen tres secciones clásicas de una prueba unitaria: Given (Dado), When (Cuando) y Then (Entonces). Estas secciones indican lo que se espera que suceda en la prueba.
- En la sección Given se crea un objeto de la clase `Cart` y se agregan varios ítems de la base de datos de plantas mediante la llamada a `cart.addItem()`. Además, se define el nombre de uno de los ítems, `plantName`, que será utilizado para buscarlo posteriormente.
- En la sección When se llama al método `cart.findItemByName(plantName)` para buscar el ítem con el nombre `plantName`.
- En la sección Then se verifica que el ítem devuelto por `findItemByName` sea igual al ítem esperado mediante la llamada a `assertEquals(ItemEntity(PlantsDatabase.getAllPlants()[8],`

`1), ItemEntity(result!!, 1))`.

- Si el resultado de la prueba es satisfactorio, la prueba unitaria finalizará correctamente. De lo contrario, el framework de pruebas indicará el fallo y el desarrollador deberá corregir el código para que la prueba pase exitosamente.

```
@Test
    fun findItem_noExistsItem_returnsNull() {
        //Given
        val cart = Cart()
        val plantName = "Parangaricutirimicuaro"
        cart.apply {
            addItem(PlantsDatabase.getAllPlants()[8])
            addItem(PlantsDatabase.getAllPlants()[5])
            addItem(PlantsDatabase.getAllPlants()[10])
            addItem(PlantsDatabase.getAllPlants()[15])
            addItem(PlantsDatabase.getAllPlants()[20])
            addItem(PlantsDatabase.getAllPlants()[25])
            addItem(PlantsDatabase.getAllPlants()[30])
        }
        //When
        val result = cart.findItemByName(plantName)
        //Then
        assertEquals(null, result)
    }
```

A continuación, se detalla el funcionamiento del código:

- Dentro del método se definen tres secciones clásicas de una prueba unitaria: Given (Dado), When (Cuando) y Then (Entonces). Estas secciones indican lo que se espera que suceda en la prueba.
- En la sección Given se crea un objeto de la clase `Cart` y se agregan varios ítems de la base de datos de plantas mediante la llamada a `cart.addItem()`. Además, se define el nombre de un ítem que no existe en el carrito de compras, `plantName`, que será utilizado para buscarlo posteriormente.
- En la sección When se llama al método `cart.findItemByName(plantName)` para buscar el ítem con el nombre `plantName`.
- En la sección Then se verifica que el resultado de la búsqueda sea igual a `null` mediante la llamada a `assertEquals(null, result)`.
- Si el resultado de la prueba es satisfactorio, la prueba unitaria finalizará correctamente. De lo contrario, el framework de pruebas indicará el fallo y el desarrollador deberá corregir el código para que la prueba pase exitosamente.

```
@Test
    fun updateItem_itemExists_itemUpdated() {
```

```
        //Given
        val cart = Cart()
        cart.apply {
            addItem(PlantsDatabase.getAllPlants()[5])
            addItem(PlantsDatabase.getAllPlants()[0])
            addItem(PlantsDatabase.getAllPlants()[10])
        }
        //When
        cart.updateItem(PlantsDatabase.getAllPlants()[0], 1)
        //Then
        assertEquals(3, cart.shoppingCart.size)
    }
```

A continuación, se detalla el funcionamiento del código:

- Dentro del método se definen tres secciones clásicas de una prueba unitaria: Given (Dado), When (Cuando) y Then (Entonces). Estas secciones indican lo que se espera que suceda en la prueba.
- En la sección Given se crea un objeto de la clase `Cart` y se agregan varios ítems de la base de datos de plantas mediante la llamada a `cart.addItem()`. Además, se define el nombre de un ítem que no existe en el carrito de compras, `plantName`, que será utilizado para buscarlo posteriormente.
- En la sección When se llama al método `cart.findItemByName(plantName)` para buscar el ítem con el nombre `plantName`.
- En la sección Then se verifica que el resultado de la búsqueda sea igual a `null` mediante la llamada a `assertEquals(null, result)`.
- Si el resultado de la prueba es satisfactorio, la prueba unitaria finalizará correctamente. De lo contrario, el framework de pruebas indicará el fallo y el desarrollador deberá corregir el código para que la prueba pase exitosamente.

```
@Test
    fun increaseQuantity_incrementByFive_quantityIncreasedFive() {
        //Given
        val cart = Cart()
        val plant1 = PlantsDatabase.getAllPlants()[5]
        val plant2 = PlantsDatabase.getAllPlants()[0]
        val plant3 = PlantsDatabase.getAllPlants()[10]
        plant1.quantity = 1
        plant2.quantity = 2
        plant3.quantity = 3
        cart.apply {
            addItem(plant1)
            addItem(plant2)
            addItem(plant3)
        }
        //When
        cart.increaseQuantity(PlantsDatabase.getAllPlants()[0], 5)
```

```
        //Then
        val plantExpected = PlantsDatabase.getAllPlants()[0]
        plantExpected.quantity = 7
        assertEquals(plantExpected, cart.shoppingCart[1])
    }
```

A continuación, se detalla el funcionamiento del código:

En el método `increaseQuantity()`, se aumenta la cantidad de un producto en el carrito de compras en la cantidad especificada. El método recibe dos parámetros: el primer parámetro es el objeto `Item` (planta en este caso) que se quiere actualizar, y el segundo parámetro es la cantidad que se desea agregar a la cantidad actual del objeto.

En este caso de prueba en particular, se crean tres plantas, se les asigna una cantidad específica y se agregan al carrito. Luego, se llama al método `increaseQuantity()` para incrementar en 5 la cantidad de una de las plantas agregadas. Finalmente, se verifica que la cantidad de la planta aumentó en 5 en el carrito de compras.

El método `assertEquals()` se utiliza para comparar el objeto esperado con el objeto obtenido. En este caso, se crea un objeto `plantExpected` con los mismos atributos que el objeto `Item` actualizado y se compara con el objeto en la posición correspondiente del carrito de compras. Si ambos objetos son iguales, entonces el caso de prueba se considera exitoso.

```
@Test
    fun decreaseQuantity_decreaseByOne_quantityDecreasedOne() {
        //Given
        val cart = Cart()
        val plant1 = PlantsDatabase.getAllPlants()[5]
        val plant2 = PlantsDatabase.getAllPlants()[0]
        val plant3 = PlantsDatabase.getAllPlants()[10]
        plant1.quantity = 1
        plant2.quantity = 2
        plant3.quantity = 3
        cart.apply {
            addItem(plant1)
            addItem(plant2)
            addItem(plant3)
        }
        //When
        cart.decreaseQuantity(plant2, 1)
        //Then
        val plantExpected = PlantsDatabase.getAllPlants()[0]
        plantExpected.quantity = 1
        assertEquals(plantExpected, cart.shoppingCart[1])
    }
```

A continuación, se detalla el funcionamiento del código:

En el bloque "Given" se crea una instancia de la clase "Cart" y se agregan tres objetos de la clase "Plant" a la lista de compras del carrito. Cada planta tiene una cantidad preestablecida

para este test.

En el bloque "When" se llama al método "decreaseQuantity" y se le pasa como argumento la planta "plant2" y el número 1, que indica la cantidad a reducir.

En el bloque "Then" se espera que la cantidad de la planta "plant2" se reduzca en 1 unidad, por lo que se crea una planta "plantExpected" con la misma información que "plant2" pero con la cantidad ajustada en 1. Finalmente, se verifica que la planta en la posición 1 de la lista de compras sea igual a "plantExpected" mediante la función "assertEquals". Si la cantidad ha sido reducida en 1 unidad, la prueba pasará exitosamente.

```
@Test
    fun calcTotal_addFiveItems_getCostFiveItems() {
        //Given
        val cart = Cart()
        val plant1 = PlantsDatabase.getAllPlants()[5]
        val plant2 = PlantsDatabase.getAllPlants()[0]
        val plant3 = PlantsDatabase.getAllPlants()[10]
        plant1.quantity = 1
        plant2.quantity = 2
        plant3.quantity = 3
        cart.apply {
            addItem(plant1)
            addItem(plant2)
            addItem(plant3)
        }
        //When
        val result = cart.calcTotal()
        //Then
        val cost1 = plant1.quantity * plant1.price
        val cost2 = plant2.quantity * plant2.price
        val cost3 = plant3.quantity * plant3.price
        val totalCost = cost1 + cost2 + cost3
        assertEquals(totalCost.toDouble(), result)
    }
}
```

A continuación, se detalla el funcionamiento del código:

El código comienza creando un objeto `Cart`, y luego crea tres objetos `Plant` y establece sus cantidades. Luego, agrega los tres objetos `Plant` al carrito de compras utilizando el método `addItem()`.

Después, la prueba llama a la función `calcTotal()` en el objeto `cart`, que devuelve el costo total de todos los elementos en el carrito. Luego, la prueba calcula manualmente el costo total de los tres objetos `Plant` utilizando sus cantidades y precios, y lo compara con el valor devuelto por `calcTotal()`. La prueba pasa si estos dos valores son iguales.

En resumen, esta prueba comprueba que la función `calcTotal()` calcula correctamente el costo total de los elementos en el carrito de compras, lo que es importante para garantizar la

precisión del costo total que se muestra al usuario.

▼ **ServiceTest**

```
@Test
    fun login_noUserRegistered_authenticatedUserFalse() {
        //Given
        val user = UserEntity("Pablo", "123456")
        val service = Service()
        //When
        service.login(user)
        val result = service.authenticatedUser
        //Then
        assertEquals(false, result)
    }
```

A continuación, se detalla el funcionamiento del código:

- `@Test` es una anotación de JUnit que indica que el método siguiente es un test unitario.
- `fun login_noUserRegistered_authenticatedUserFalse()` es el nombre del test, que describe el escenario que se está probando.
- Dentro del test se crea un objeto `UserEntity` con el nombre de usuario "Pablo" y la contraseña "123456", que se utilizará para llamar a la función `login` más adelante.
- También se crea un objeto `Service`, que es el objeto que se está probando en este test.
- Luego se llama a la función `login` del objeto `Service` pasando como parámetro el objeto `UserEntity` creado previamente.
- A continuación, se guarda el valor de la propiedad `authenticatedUser` del objeto `Service` en la variable `result`.
- Por último, se utiliza la función `assertEquals` de JUnit para verificar que `result` es igual a `false`.

```
@Test
    fun login_userRegistered_authenticatedUserTrue() {
        //Given
        val username = "Pablo"
        val password = "123456"
        val user = UserEntity(username, password)
        val service = Service()
        service.userRegister(user)
        //When
        service.login(UserEntity(username, password))
        val result = service.authenticatedUser
        //Then
        assertEquals(true, result)
    }
```

A continuación, se detalla el funcionamiento del código:

- `@Test` es una anotación de JUnit que indica que el método siguiente es un test unitario.
- `fun login_userRegistered_authenticatedUserTrue()` es el nombre del test, que describe el escenario que se está probando.
- Dentro del test se define un nombre de usuario (`username`) y una contraseña (`password`) que se utilizarán para crear un objeto `UserEntity`.
- A continuación, se crea un objeto `UserEntity` con el `username` y `password` y se pasa como parámetro a la función `userRegister` del objeto `Service` creado previamente, registrando así el usuario.
- Luego se llama a la función `login` del objeto `Service` pasando como parámetro el mismo objeto `UserEntity` creado previamente.
- Se guarda el valor de la propiedad `authenticatedUser` del objeto `Service` en la variable `result`.
- Finalmente, se utiliza la función `assertEquals` de JUnit para verificar que `result` es igual a `true`.

```
@Test
    fun existeUsuario_userExists_true() {
        //Given
        val user = UserEntity("Pablo", "123456")
        val service = Service()
        service.userRegister(user)
        //When
        val result = service.userExist(user.username)
        //Then
        assertEquals(true, result)
    }
```

A continuación, se detalla el funcionamiento del código:

- `@Test` es una anotación de JUnit que indica que el método siguiente es un test unitario.
- `fun existeUsuario_userExists_true()` es el nombre del test, que describe el escenario que se está probando.
- Dentro del test se crea un objeto `UserEntity` con el nombre de usuario "Pablo" y la contraseña "123456", que se utilizará para registrar al usuario en el objeto `Service`.
- También se crea un objeto `Service`, que es el objeto que se está probando en este test.
- Se registra el usuario creado previamente en el objeto `Service` llamando a la función `userRegister`.

- Luego se llama a la función `userExist` del objeto `Service` pasando como parámetro el `username` del objeto `UserEntity` creado previamente.
- Se guarda el resultado de la función `userExist` en la variable `result`.
- Finalmente, se utiliza la función `assertEquals` de JUnit para verificar que `result` es igual a `true`.

```
@Test
    fun existeUsuario_noUserExists_false() {
        //Given
        val user = UserEntity("Pablo", "123456")
        val service = Service()
        //When
        val result = service.userExist(user.username)
        //Then
        assertEquals(false, result)
    }
```

A continuación, se detalla el funcionamiento del código:

- `@Test` es una anotación de JUnit que indica que el método siguiente es un test unitario.
- `fun existeUsuario_noUserExists_false()` es el nombre del test, que describe el escenario que se está probando.
- Dentro del test se crea un objeto `UserEntity` con el nombre de usuario "Pablo" y la contraseña "123456", pero no se registra en el objeto `Service`.
- También se crea un objeto `Service`, que es el objeto que se está probando en este test.
- Luego se llama a la función `userExist` del objeto `Service` pasando como parámetro el `username` del objeto `UserEntity` creado previamente.
- Se guarda el resultado de la función `userExist` en la variable `result`.
- Finalmente, se utiliza la función `assertEquals` de JUnit para verificar que `result` es igual a `false`.

---