PRACTICA 10 UT7

Gestión de Stock con JavaFX

Guia de Referencia del Proyecto

DAM2 — Desarrollo de Interfaces

1. Introducción y Objetivos

Esta práctica consiste en desarrollar una aplicación de escritorio en JavaFX siguiendo el patrón de diseño MVC (Model-View-Controller) para gestionar el stock de productos de una tienda.

Los datos se almacenan en memoria usando un ArrayList, sin base de datos. La aplicación incluye validaciones de seguridad completas y pruebas unitarias escritas con JUnit 5 y Mockito.

Objetivos principales

- Diseñar una interfaz gráfica usable con JavaFX y FXML.
- Implementar la lógica de negocio con validaciones robustas contra datos erróneos.
- Cubrir los casos principales con un mínimo de 5 pruebas unitarias por clase de test.
- Aplicar validaciones de seguridad que impidan datos maliciosos (XSS, valores negativos, NaN...).
- Estructurar el proyecto con el patrón MVC para separar responsabilidades.

2. Configuración del Entorno

2.1 Requisitos previos

- Java 17 o superior instalado (se recomienda JDK 17 LTS).
- Maven 3.8+ instalado y configurado en el PATH del sistema.
- IDE compatible: IntelliJ IDEA, Eclipse o VS Code con extensiones Java.

2.2 Dependencias del pom.xml

El fichero pom.xml declara todas las dependencias necesarias. A continuación se detalla cada una:

Dependencia	Versión	Propósito
javafx-controls	21.0.1	Componentes UI: Button, TextField, TableView, Label...
javafx-fxml	21.0.1	Permite cargar archivos .fxml en tiempo de ejecución
junit-jupiter	5.10.1	Framework de pruebas unitarias JUnit 5
mockito-core	5.8.0	Mocking y verificación de comportamiento en tests
mockito-junit-jupiter	5.8.0	Integra Mockito con la extensión de JUnit 5
maven-surefire-plugin	3.2.2	Plugin Maven para ejecutar pruebas JUnit 5
javafx-maven-plugin	0.0.8	Permite lanzar la app JavaFX con mvn javafx:run

2.3 Comandos principales

Ejecutar la interfaz gráfica: `mvn javafx:run`

Ejecutar todas las pruebas: `mvn test`

Informe de pruebas generado en: `target/surefire-reports/`

3. Estructura del Proyecto

El proyecto sigue el patrón MVC con la siguiente estructura de directorios y archivos:

```
practica_10/
  pom.xml                                <- Configuración Maven
  src/
    main/
      java/dev/mendo/
        MainApp.java                     <- Punto de entrada JavaFX
        Product.java                     <- Modelo de datos (M)
        ProductService.java              <- Logica de negocio
        ProductController.java           <- Controlador de la UI (C)
      resources/
        ProductView.fxml                 <- Vista JavaFX (V)
    test/
      java/
        ProductServiceTest.java          <- JUnit 5 + Mockito
        SecurityTests.java               <- Pruebas de seguridad
```

Cada capa del patron MVC tiene una responsabilidad bien definida:

Capa	Clase	Responsabilidad
Modelo (M)	Product.java	Estructura de datos del producto
Logica	ProductService.java	Reglas de negocio, validaciones y almacenamiento en ArrayList
Controlador (C)	ProductController.java	Recibe eventos de la UI, llama al servicio y actualiza la vista
Vista (V)	ProductView.fxml	Define la interfaz gráfica en formato XML declarativo
Entrada	MainApp.java	Carga el FXML y lanza la ventana principal de JavaFX

4. Descripción de Clases y Métodos

4.1 Product.java — Modelo de datos

Clase POJO (Plain Old Java Object) que representa la entidad de negocio. No contiene lógica, solo datos y sus accesorios. JavaFX utiliza los nombres de los getters para enlazar las columnas de la TableView mediante PropertyValueFactory.

Elemento	Tipo	Descripción
name	String	Nombre del producto. Actúa como identificador visible en la tabla.
quantity	int	Unidades disponibles en stock. Puede ser 0 (agotado) pero no negativo.
price	double	Precio unitario en euros. Debe ser mayor que cero.
Product(name, qty, price)	Constructor	Crea un producto con los tres atributos obligatorios.
getName() / setName()	Getter/Setter	Acceso y modificación del nombre del producto.
getQuantity() / setQuantity()	Getter/Setter	Acceso y modificación de la cantidad en stock.
getPrice() / setPrice()	Getter/Setter	Acceso y modificación del precio unitario.
toString()	Override	Representación legible del producto para logs y depuración.

4.2 ProductService.java — Logica de negocio

Clase central del proyecto. Contiene el almacenamiento en memoria y todas las validaciones. Es la única clase que puede modificar la lista interna de productos.

Variables y constantes

Elemento	Tipo	Descripción
products	List<Product>	ArrayList interno que almacena todos los productos del stock.
INVALID_CHARS	Pattern (regex)	Patrón que detecta caracteres peligrosos: < > " ' % ; ( ) & +
MAX_NAME_LENGTH	int (100)	Límite máximo de caracteres permitido para el nombre.

Métodos públicos

Método	Retorno	Descripción
addProduct(name, qty, price)	Product	Valida los campos y agrega el producto al ArrayList. Devuelve el producto creado.

removeProduct(index)	void	Elimina el producto en la posición indicada. Lanza una excepción si el índice no es válido.
updateProduct(index, name, qty, price)	void	Actualiza los datos del producto en la posición dada tras validar los nuevos valores.
getProducts()	List<Product>	Devuelve la lista como Collections.unmodifiableList para proteger el estado interno.
getProductCount()	int	Devuelve el número total de productos almacenados.
findByName(name)	Product	Busca un producto por nombre ignorando las mayúsculas. Devuelve null si no existe.

Métodos privados de validación

Método	Descripción
validateProduct(name, qty, price)	Método coordinador: llama a los tres validadores individuales en secuencia.
validateName(name)	Verifica que no sea null, no esté vacío, no supere 100 chars y no tenga caracteres peligrosos.
validateQuantity(qty)	Verifica que la cantidad sea mayor o igual a cero.
validatePrice(price)	Verifica que el precio sea mayor que cero, que no sea NaN ni infinito.

4.3 ProductController.java — Controlador de la UI

Clase anotada con @FXML. JavaFX inyecta automaticamente los campos anotados al cargar el FXML. Actua de puente entre la vista y el servicio de negocio.

Campos FXML inyectados

Campo	Tipo	Descripción
txtName	TextField	Campo de texto para introducir o editar el nombre del producto.
txtQuantity	TextField	Campo de texto para la cantidad. Se parsea a int en handleAddProduct.
txtPrice	TextField	Campo de texto para el precio. Acepta tanto punto como coma decimal.
tableProducts	TableView<Product>	Tabla principal. Muestra los productos del observableList.
colName, colQuantity, colPrice	TableColumn	Columnas enlazadas con los getters de Product via PropertyValueFactory.
lblStatus	Label	Barra de estado inferior: verde para éxito, rojo para errores.

Metodos de eventos y auxiliares

Método	Descripción
initialize()	Configura las columnas, enlaza el observableList y agrega un listener para rellenar el formulario al seleccionar una fila.
handleAddProduct()	Lee el formulario, llama a productService.addProduct() y refresca la tabla. Captura IllegalArgumentException con setStatus().
handleUpdateProduct()	Verifica que haya fila seleccionada, llama a updateProduct() y refresca la tabla.
handleDeleteProduct()	Muestra diálogo de confirmación y llama a removeProduct() si el usuario acepta.
handleClear()	Limpia los tres campos del formulario y selecciona la fila de la tabla.
refreshTable()	Llama a observableList.setAll() con la lista del servicio para actualizar la TableView.
setStatus(msg, isError)	Cambia el texto y color de lblStatus: verde para ok, rojo para error.
parseQuantity(text)	Convertir String a int. Lanza IllegalArgumentException con mensaje claro si el texto no es numérico.
parsePrice(text)	Convertir String a double. Reemplaza coma por punto para aceptar ambos formatos decimales.

4.4 ProductView.fxml — Vista JavaFX

Define la interfaz gráfica en formato XML declarativo. Usar FXML en lugar de construir la UI por código permite separar la vista del controlador y facilita el diseño visual con Scene Builder.

Componente FXML	Descripción
BorderPane (raíz)	Layout principal. Divide la ventana en top (cabecera), center (contenido) y bottom (estado).
HBox con Label (top)	Cabecera azul oscuro con el título de la aplicación.
TitledPane (formulario)	Sección con título que agrupa el formulario de entrada de datos.
GridPane	Organiza Labels y TextFields en filas y columnas perfectamente alineadas.
TextField txtName/txtQuantity/txtPrice	Campos de entrada enlazados al controlador mediante fx:id.
HBox con 4 Buttons	Botones Añadir (verde), Actualizar (azul), Eliminar (rojo), Limpiar (gris).
TitledPane con TableView	Tabla que muestra los productos. Nombre, Cantidad y Precio.
Label lblStatus (bottom)	Barra de estado inferior con retroalimentación visual al usuario.

4.5 MainApp.java — Punto de entrada

Extiende Application (clase base de JavaFX). El método start(Stage) carga el FXML con FXMLLoader, crea la Scene y la asigna al Stage principal. El método estático main() invoca launch() para inicializar el ciclo de vida de JavaFX.

Método	Descripción
start(Stage primaryStage)	Carga ProductView.fxml, crea la escena de 800x550 px y muestra la ventana.
main(String[] args)	Punto de entrada de la JVM. Llama a launch(args) que dispara el ciclo JavaFX.

5. Validaciones de Seguridad

Todas las validaciones se centralizan en ProductService, lo que garantiza que los datos sean correctos independientemente de la capa que los introduzca. El controlador propaga las excepciones a la etiqueta de estado para informar al usuario.

Validacion	Regla aplicada	Excepcion lanzada
Nombre nulo	name != null	IllegalArgumentException
Nombre vacio o solo espacios	name.trim() no vacío	IllegalArgumentException
Nombre demasiado largo	longitud <= 100 caracteres	IllegalArgumentException
Caracteres peligrosos (XSS/SQL)	Sin < > " ' % ; ( ) & +	IllegalArgumentException
Cantidad negativa	quantity >= 0	IllegalArgumentException
Precio <= 0	price > 0	IllegalArgumentException
Precio NaN	!Double.isNaN(price)	IllegalArgumentException
Precio infinito	!Double.isInfinite(price)	IllegalArgumentException
Indice fuera de rango (remove/update)	0 <= index < size()	IndexOutOfBoundsException

La expresión regular usada para detectar caracteres peligrosos es:

```
[<>"'%;()&+]
```

Esta expresión bloquea los caracteres más comunes en ataques de tipo XSS (Cross-Site Scripting) e inyección SQL, proporcionando una capa defensiva incluso si en el futuro los datos se exponen a una vista web o base de datos.

Inmutabilidad de la lista: getProducts() devuelve Collections.unmodifiableList(), lo que impide que código externo altere el ArrayList interno sin pasar por las validaciones del servicio.

6. Pruebas Unitarias — ProductServiceTest

Esta clase usa @ExtendWith(MockitoExtension.class) para integrar Mockito con JUnit 5. Se declara un @Spy de ProductService, que ejecuta el código real pero permite verificar con verify() que ciertos métodos fueron invocados con los argumentos esperados.

Diferencia Spy vs Mock: Un Mock reemplaza completamente el objeto (devuelve null por defecto). Un Spy envuelve el objeto real y solo intercepta lo que se le pida, permitiendo comprobar interacciones sin alterar el comportamiento.

Test	Escenario	Por que es importante
Test 1: Producto válido	addProduct() con datos correctos crea y almacena el producto.	Verifica el flujo principal (happy path) del sistema.
Test 2: Nombre vacío	addProduct('', ...) lanza IllegalArgumentException.	Impide productos sin identificador en el stock.
Test 3: Precio negativo	addProduct con price=-1 lanza excepción.	Un precio negativo no tiene sentido de negocio.
Test 4: Cantidad negativa	addProduct con quantity=-5 lanza excepción.	El stock no puede representar unidades negativas.
Test 5: Eliminar producto	removeProduct(0) reduce el conteo y reordena la lista.	La eliminación debe ser funcional y no dejar huecos.
Test 6: Índice válido (Spy)	removeProduct(99) lanza excepción; verify() confirma la llamada exacta.	Demuestra el uso de Mockito @Spy para auditar invocaciones.
Test 7: Actualizar producto	updateProduct() modifica nombre, cantidad y precio.	La edición debe conservar la integridad de los datos.
Test 8: Buscar por nombre (Spy)	findByName() devuelve el producto; verify() confirma la llamada.	Prueba búsqueda case-insensitive y el uso de Mockito spy.
Test 9: Nombre con XSS	Nombre '<script>alert()</script>' lanza excepción.	Prueba la protección contra inyección de código HTML.
Test 10: Lista inmutable	getProducts().add() lanza UnsupportedOperationException.	Garantiza el encapsulamiento del estado interno del servicio.

7. Pruebas de Seguridad — SecurityTests

Esta clase complementa ProductServiceTest con pruebas específicamente orientadas a atacar los límites de las validaciones. Cada test intenta introducir un dato problemático y verifica que el sistema lo rechace correctamente.

Test	Escenario de ataque o caso límite	Por qué se prueba
Seg-1: Precio = 0	price=0.0 como valor límite entre válido e invalido.	El precio cero podría colarse en validaciones mal escritas (>= en vez de >).
Seg-2: Solo espacios	name=' ' — trim devuelve cadena vacía.	Evita el bypass de la validación de nombre vacío usando espacios.
Seg-3: Inyección SQL	name="'; DROP TABLE products; --"	Simula un ataque SQL injection clásico en el campo nombre.
Seg-4: Nombre largo (101)	Cadena de 101 caracteres que supera el límite de 100.	Previene datos inutilizables y posibles desbordamientos de buffer.
Seg-5: Precio infinito	price=Double.POSITIVE_INFINITY.	Java permite infinito en doubles; la app debe rechazarlo explícitamente.
Seg-6: Precio NaN	price=Double.NaN (resultado de 0.0/0.0 entre otros).	NaN produce comportamiento impredecible en cálculos y comparaciones.
Seg-7: HTML en nombre	name='Producto' con etiquetas HTML.	Previene XSS si los datos se mostrarán en una vista web en el futuro.
Seg-8: Índice negativo	updateProduct(-1, ...) debe lanzar excepción.	Los índices negativos son un vector de acceso indebido a la memoria.
Seg-9: Nombre null	name=null lanza IllegalArgumentException con mensaje claro.	Las NullPointerExceptions silenciosas son un fallo grave de robustez.
Seg-10: Cantidad = 0 (válido)	quantity=0 es aceptable (producto agotado).	Confirma que el límite inferior correcto es 0, no 1.

8. Cómo Ejecutar el Proyecto

8.1 Desde la terminal (Maven)

1. Descomprimir el archivo ZIP del proyecto en una carpeta local.
2. Abrir un terminal en la raíz del proyecto (donde se encuentra el pom.xml).
3. Para lanzar la interfaz gráfica JavaFX:

```
mvn javafx:run
```

4. Para ejecutar todas las pruebas unitarias:

```
mvn test
```

5. El informe de resultados se genera en la carpeta:

```
target/surefire-reports/
```

8.2 Desde IntelliJ IDEA

- Abrir el proyecto: File > Open y seleccionar la carpeta raíz.
- IntelliJ detecta el pom.xml automáticamente e importa las dependencias.
- Ejecutar la app: click derecho en MainApp.java > Run 'MainApp.main()'.
- Ejecutar tests: click derecho en src/test > Run All Tests.

8.3 Posibles problemas y soluciones

Problema	Solución
'JavaFX runtime components are missing'	Usar siempre mvn javafx:run en lugar de ejecutar el .jar directamente.
Los tests no se detectan	Verificar que maven-surefire-plugin 3.2.2 está declarado en el pom.xml.
Ventana no abre / FXML no encontrado	Comprobar que ProductView.fxml está en src/main/resources/com/example/stock/.
Dependencias no se descargan	Ejecutar mvn dependency:resolver o verificar la conexión a internet y el repositorio Maven Central.
Error de compilación Java version	Asegurarse de tener JDK 17 instalado y que JAVA_HOME apunta a él.