Práctica 10 — UT7

Gestión de Stock con JavaFX

Guía de Referencia del Proyecto

DAM2 · Desarrollo de Interfaces

Índice de Contenidos

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 utilizando un **ArrayList**, sin necesidad de base de datos. La aplicación incluye validaciones de seguridad y pruebas unitarias escritas con **JUnit 5** y **Mockito**.

Objetivos principales:

- Diseñar una interfaz gráfica usable con JavaFX y FXML.
- Implementar la capa de lógica de negocio con validaciones robustas.
- Cubrir los casos principales con pruebas unitarias (mínimo 5 por clase).
- Aplicar validaciones de seguridad que impidan datos erróneos o maliciosos.

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 del proyecto. Las más importantes son:

Dependencia	Versión	Uso
javafx-controls	21.0.1	Componentes UI (Button, TextField, TableView…)
javafx-fxml	21.0.1	Carga de 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
mockito-junit-jupiter	5.8.0	Integración Mockito con JUnit 5

2.3 Plugin de JavaFX para Maven

El plugin **javafx-maven-plugin** (versión 0.0.8) permite ejecutar la aplicación JavaFX directamente desde Maven sin necesidad de configurar el módulo path manualmente:

```
mvn javafx:run
```

2.4 Ejecutar las pruebas

```
mvn test
```

3. Estructura del Proyecto

El proyecto sigue el patrón **MVC** con la siguiente organización de directorios:

```
StockManagementFX/
  pom.xml
  src/
    main/
      java/com/example/stock/
        MainApp.java        # Punto de entrada JavaFX
        Product.java        # Modelo de datos (M)
        ProductService.java  # Logica de negocio
        ProductController.java # Controlador (C)
      resources/com/example/stock/
        ProductView.fxml     # Vista (V)
    test/
      java/com/example/stock/
        ProductServiceTest.java # JUnit + Mockito
        SecurityTests.java     # Validaciones seguridad
```

4. Descripción de Clases y Métodos

4.1 Product.java — Modelo de datos

Representa la entidad de negocio: un producto con nombre, cantidad y precio. No contiene lógica, solo datos y sus accesores.

Elemento	Tipo	Descripción
name	String	Nombre del producto (clave identificativa)
quantity	int	Unidades disponibles en stock
price	double	Precio unitario en euros
Product(name, quantity, price)	Constructor	Crea un producto con los tres atributos obligatorios
getName() / setName()	Getter/Setter	Acceso y modificación del nombre
getQuantity() / setQuantity()	Getter/Setter	Acceso y modificación de la cantidad
getPrice() / setPrice()	Getter/Setter	Acceso y modificación del precio
toString()	Override	Representacion legible del producto para logs/debug

4.2 ProductService.java — Lógica de negocio

Contiene toda la lógica de la aplicación y el almacenamiento en memoria. Es la clase central del proyecto. Las constantes y el patrón de validación se definen a nivel de clase para reutilizarlos en todos los métodos.

Elemento	Tipo	Descripción
products	List<Product>	ArrayList interno donde se almacenan todos los productos
INVALID_CHARS	Pattern (regex)	Patrón que detecta caracteres peligrosos (<>";%&+)
MAX_NAME_LENGTH	int (100)	Limite maximo de caracteres para el nombre
addProduct(name, qty, price)	Product	Valida y añade un nuevo producto; devuelve el creado
removeProduct(index)	void	Elimina el producto en la posicion dada
updateProduct(index, name, qty, price)	void	Actualiza los datos del producto en ese indice
getProducts()	List<Product>	Devuelve lista no modificable (Collections.unmodifiableList)
getProductCount()	int	Numero total de productos almacenados
findByName(name)	Product	Busca por nombre (case-insensitive); devuelve null si no existe
validateProduct(...)	void (privado)	Llama a validateName, validateQuantity y validatePrice
validateName(name)	void (privado)	Verifica que no sea nulo, vacio, largo ni peligroso
validateQuantity(qty)	void (privado)	Verifica que la cantidad sea >= 0
validatePrice(price)	void (privado)	Verifica que el precio sea > 0, no NaN ni infinito

4.3 ProductController.java — Controlador de la UI

Clase anotada con **@FXML** que recibe los eventos de los botones y elementos de la interfaz gráfica. Actúa de puente entre la vista (FXML) y el servicio.

Campo / Método	Descripción
@FXML txtName, txtQuantity, txtPrice	Campos de texto enlazados desde el FXML
@FXML tableProducts, colName…	Tabla y columnas enlazadas desde el FXML
@FXML lblStatus	Etiqueta de estado/mensajes al usuario
observableList	ObservableList<Product> que enlaza la TableView con los datos
productService	Instancia del servicio; la controladora no la mockea
initialize()	Configura columnas, fuente de datos y listener de seleccion de fila
handleAddProduct()	Lee el formulario, llama a addProduct() y refresca la tabla
handleUpdateProduct()	Actualiza el producto seleccionado con los datos del formulario
handleDeleteProduct()	Muestra confirmacion y elimina el producto seleccionado
handleClear()	Limpia todos los campos del formulario
refreshTable()	Sincroniza el observableList con el estado actual del servicio
setStatus(msg, isError)	Actualiza lblStatus con color verde (OK) o rojo (error)
parseQuantity(text)	Convierte texto a int; lanza IllegalArgumentException si falla
parsePrice(text)	Convierte texto a double; acepta coma como separador decimal

4.4 ProductView.fxml — Vista JavaFX

Define la interfaz gráfica en formato XML. Las ventajas de usar FXML frente a construir la UI por código son la separación de responsabilidades, la legibilidad y la posibilidad de usar Scene Builder para diseñar visualmente.

Componente FXML	Descripción
BorderPane (raíz)	Layout principal: top=cabecera, center=contenido, bottom=estado
HBox (top)	Cabecera oscura con el titulo de la aplicacion
TitledPane (formulario)	Sección colapsable con el formulario de entrada de datos
GridPane	Organiza los labels y TextField en filas y columnas alineadas
TextField txtName/txtQuantity/txtPrice	Campos de entrada enlazados con fx:id
HBox con 4 Buttons	Botones Anadir, Actualizar, Eliminar, Limpiar con colores
TitledPane (tabla)	Sección que contiene la TableView con el listado de productos
TableView fx:id=tableProducts	Tabla principal; columnas enlazadas via PropertyValueFactory
Label fx:id=lblStatus (bottom)	Barra de estado inferior con mensajes de error o exito

4.5 MainApp.java — Punto de entrada

Extiende **Application** (clase base de JavaFX). El método **start(Stage)** carga el FXML con **FXMLLoader**, crea la escena y muestra la ventana principal. El método estático **main()** llama a **launch()** para iniciar el ciclo de vida de JavaFX.

5. Validaciones de Seguridad

Todas las validaciones se ejecutan en **ProductService** antes de insertar o modificar cualquier dato. Esto garantiza que la capa de negocio nunca recibe datos incorrectos independientemente de por dónde vengan (UI, tests, API futura…).

Validación	Regla	Excepción lanzada
Nombre nulo o vacío	name != null && !name.trim().isEmpty()	IllegalArgumentException
Nombre demasiado largo	length <= 100 caracteres	IllegalArgumentException
Caracteres peligrosos	Sin < > " ' % ; ( ) & +	IllegalArgumentException
Cantidad negativa	quantity >= 0	IllegalArgumentException
Precio <= 0	price > 0	IllegalArgumentException
Precio NaN / Infinito	!isNaN && !isInfinite	IllegalArgumentException
Indice fuera de rango	0 <= index < size()	IndexOutOfBoundsException

Protección XSS: El patrón regex **INVALID_CHARS** bloquea los caracteres utilizados en inyecciones HTML y JavaScript (<, >, ", ', %, ;, (,), &, +). Esto es esencial si los datos se mostraran en un contexto web en el futuro.

Inmutabilidad de la lista: El metodo **getProducts()** devuelve **Collections.unmodifiableList()**, lo que impide que codigo externo modifique el ArrayList interno sin pasar por las validaciones del servicio.

6. Pruebas Unitarias — ProductServiceTest

Esta clase usa la anotación **@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 llamados con los argumentos correctos.

Test	Propósito	Por qué es importante
Test 1: Añadir producto válido	addProduct() con datos correctos crea el producto y lo almacena	Verifica el flujo principal (happy path) del sistema.
Test 2: Nombre vacío	addProduct('', ...) lanza IllegalArgumentException	No permite productos sin identificador en el stock.
Test 3: Precio negativo	addProduct con price=-1 lanza excepcion	Un precio negativo no tiene sentido de negocio.
Test 4: Cantidad negativa	addProduct con quantity=-5 lanza excepcion	El stock no puede ser negativo.
Test 5: Eliminar producto	removeProduct(0) reduce el conteo y reordena la lista	La eliminacion debe ser funcional y no dejar huecos.
Test 6: Indice invalido (Mockito)	removeProduct(99) lanza IndexOutOfBoundsException; verify() confirma que no se llamo a save()	Usa @Spy para auditar invocaciones sin efectos secundarios.
Test 7: Actualizar producto	updateProduct() modifica nombre, cantidad y precio correctamente	La edicion es una operacion critica que debe conservar la integridad.
Test 8: Buscar por nombre (Mockito)	findByName() devuelve el producto correcto; verify() confirma la busqueda	Prueba la busqueda case-insensitive y el uso de Mockito spy.
Test 9: Nombre con XSS	Nombre '<script>alert()</script>' lanza excepcion	Prueba la proteccion contra inyeccion de codigo.
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 las validaciones de seguridad. Cada test intenta meter un dato problemático y comprueba que el sistema lo rechaza correctamente.

Test	Escenario de ataque / caso limite	Por qué se prueba
Seg-1: Precio = 0	price=0.0 como valor limite entre valid e invalido.	El precio cero se podria colar en validaciones mal escritas.
Seg-2: Solo espacios	name=' ' (trim devuelve cadena vacia).	Evita el bypass de la validacion de nombre vacio con espacios.
Seg-3: Inyeccion SQL	name="'; DROP TABLE products; --"	Simula un ataque SQL injection clasico en el campo nombre.
Seg-4: Nombre largo (101 chars)	Cadena de 101 'A' que supera el limite de 100 caracteres.	Previene buffer overflow conceptual y datos inutilizables.
Seg-5: Precio infinito	price=Double.POSITIVE_INFINITY.	Java permite infinito en doubles; la app debe rechazarlo.
Seg-6: Precio NaN	price=Double.NaN (resultado de 0.0/0.0, etc.).	NaN produce comportamiento impredecible en calculos.
Seg-7: HTML en nombre	name='Producto' con etiquetas HTML.	Evita XSS si los datos se mostraran en una vista web.
Seg-8: Indice negativo en update	updateProduct(-1, ...) debe lanzar excepcion.	Indices negativos son un vector de acceso indebido.
Seg-9: Nombre null	name=null lanza excepcion con mensaje claro.	NullPointerExceptions silenciosas son un fallo de robustez.
Seg-10: Cantidad = 0 (valido)	quantity=0 es aceptable (producto agotado).	Confirma que el limite inferior correcto es 0, no 1.

8. Cómo Ejecutar el Proyecto

8.1 Desde la terminal (Maven)

1. Clonar o descomprimir el proyecto en una carpeta local.

2. Abrir un terminal en la raíz del proyecto (donde está el pom.xml).

3. Compilar y ejecutar la interfaz JavaFX:

```
mvn javafx:run
```

4. Ejecutar todas las pruebas unitarias:

```
mvn test
```

5. Ver el informe de pruebas en: **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.
- Para lanzar la app: click derecho en **MainApp.java** → **Run 'MainApp.main()'**.
- Para los tests: click derecho en la carpeta **src/test** → Run All Tests.

8.3 Posibles problemas y soluciones

Problema	Solución
Error: 'JavaFX runtime not found'	Usar mvn javafx:run en lugar de ejecutar el .jar directamente.
Tests no detectados	Verificar que maven-surefire-plugin 3.2.2 está en el pom.xml.
Ventana no abre / FXML no encontrado	Comprobar que ProductView.fxml está en src/main/resources/com/example/stoc
Dependencias no descargan	Ejecutar mvn dependency:resolve o revisar la conexion a internet.