FARMEASY

Diagramas y uso de Git

Integrantes

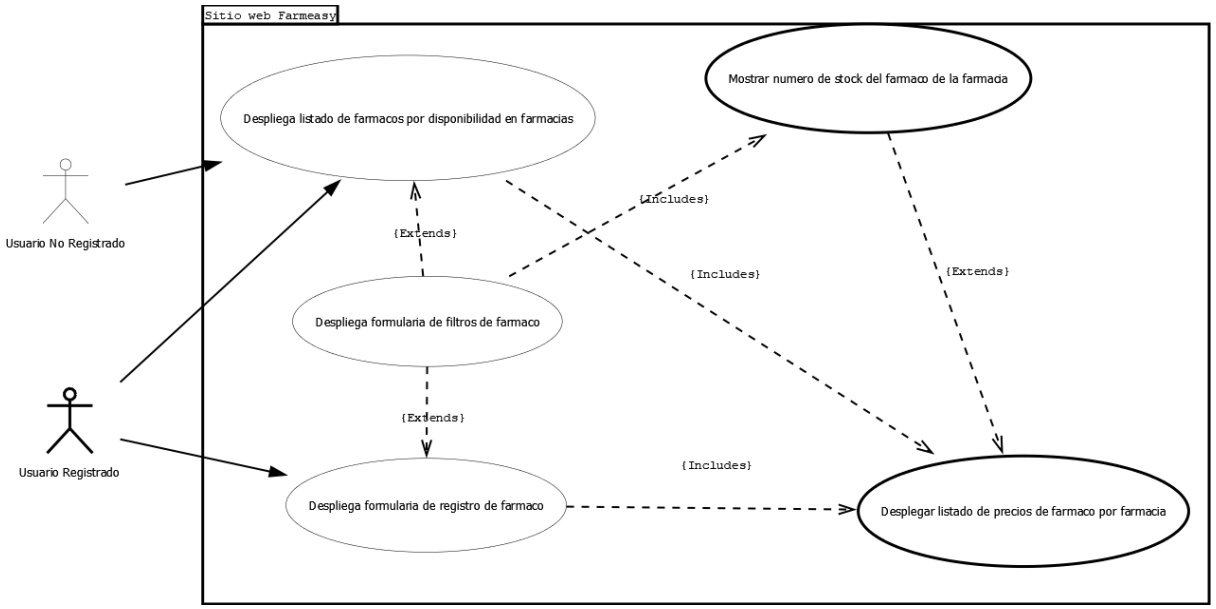
José Burgos
Cristopher Gallegos
Enrique Pincheira

Universidad de La frontera
22-04-2025

Introducción

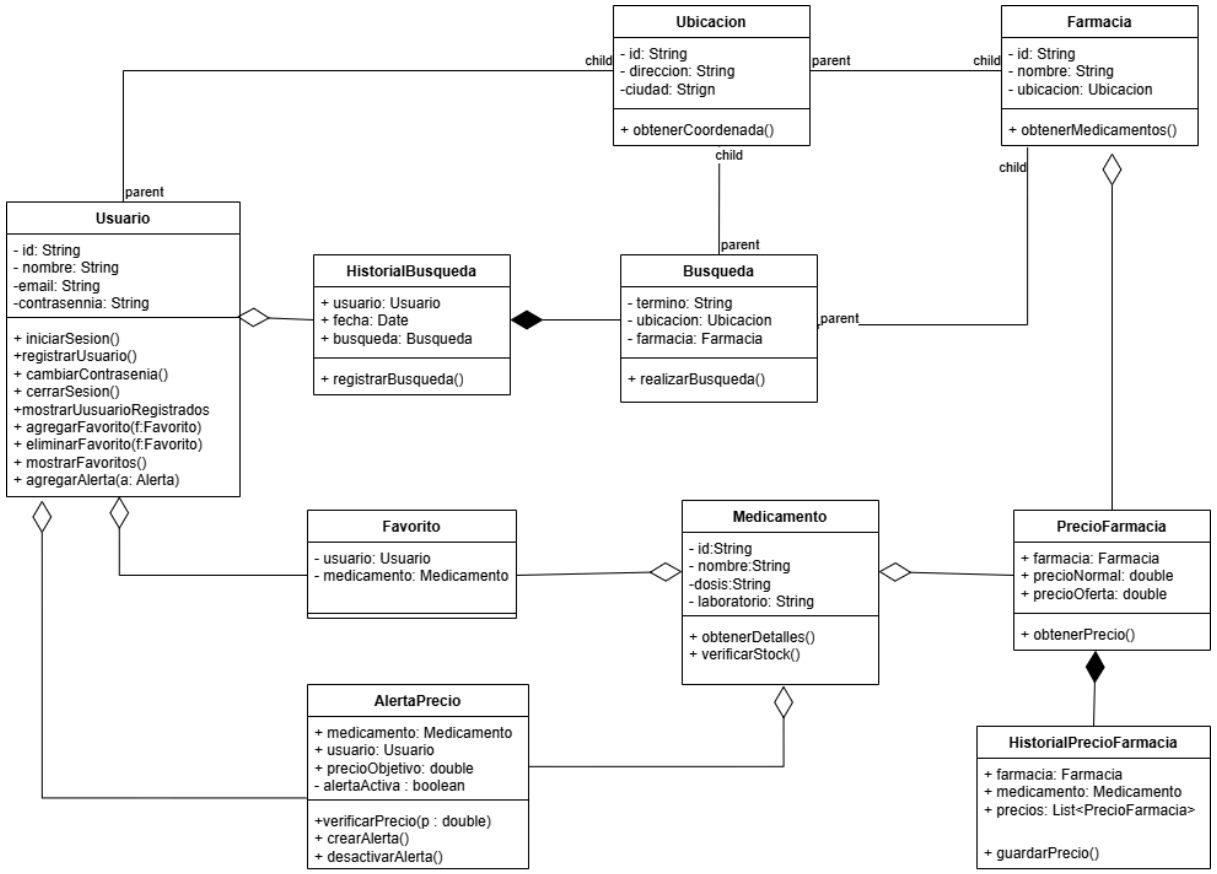
Actualmente los precios de los medicamentos entre farmacias pueden variar demasiado, en ocasiones puede resultar difícil comparar los precios entre ellos además de compararlos por tipo o bioequivalente, por ello aparece la propuesta de FARMAEASY la cual propone poder comparar los precios de los medicamentos en distintas farmacias, además incluyendo una función para poder buscar por ubicación. Esta es una web para realizar búsquedas de medicamentos no de compra de ellos, esta facilita comparar precios de medicamentos.

Diseño del sistema



- “Despliega listado de fármacos por disponibilidad en farmacias”: El usuario ingresa un fármaco específico al buscador, y subsecuentemente el sitio despliega un listado que toma en cuenta la disponibilidad de las farmacias en el sistema (ordenado por “disponible”/”no disponible”)
- “Despliega formulario de registro de fármaco”: El usuario (registrado en el sitio web) desea registrar y guardar en el sitio un fármaco para tener en observación para cambios de precio y disponibilidad en el futuro y para búsqueda inmediata en caso de querer usarlo
- “Despliega formulario de filtros de fármaco”: Después de ingresar un nombre de fármaco, la opción existe de especificar la locación y otros filtros de búsqueda para obtener un resultado más específico.
- “Desplegar listado de precios de fármaco por farmacia”: El sistema recibe un fármaco válido, y muestra el listado del precio del fármaco por farmacia al usuario
- “Mostrar número de stock del fármaco de la farmacia”: El sistema muestra al usuario el stock de la farmacia seleccionada para el fármaco específico

Diagrama de clases



Implementación

Descripción de la solución desarrollada

El sistema ha sido desarrollado utilizando los principios de Programación Orientada a Objetos (POO), lo cual permite una estructura modular, escalable y fácil de mantener. Cada clase representa una entidad del dominio, como Usuario, Medicamento, Farmacia, PrecioFarmacia, entre otras. Estas clases encapsulan tanto los atributos (datos) como los métodos (comportamiento) relacionados con dicha entidad.

Gestión de Datos y Operaciones CRUD

Las funcionalidades del sistema se implementan a través de métodos que permiten realizar las operaciones básicas sobre los datos:

- **Crear**: Se instancian objetos mediante constructores y se guardan en colecciones o bases de datos.

- **Leer**: Se accede a la información de los objetos usando métodos getters o consultas personalizadas.

- **Actualizar**: Se modifican los datos de los objetos mediante setters o métodos específicos de edición.

- **Eliminar**: Se eliminan registros desde las estructuras de almacenamiento o mediante operaciones en la base de datos.

Ejemplo de diseño POO aplicado

- Un objeto Usuario puede tener una lista de Favorito, HistorialBusqueda y AlertaPrecio, modelando una relación de agregación.

- La clase PrecioFarmacia funciona como intermediaria entre Farmacia y Medicamento, registrando precios actualizados.

- Cada HistorialPrecioFarmacia se genera a partir de cambios en los precios de PrecioFarmacia, almacenando un registro histórico por fecha.

Persistencia y estructura del sistema

- Las clases se integran con una capa de servicios y repositorios que abstraen el acceso a la base de datos.

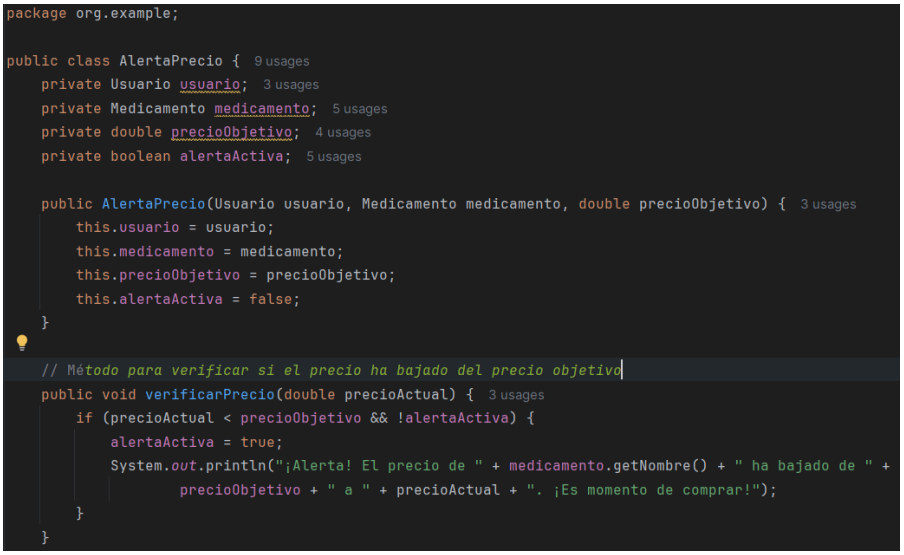
- Se aplica el patrón de diseño DAO (Data Access Object) o el uso de frameworks como Spring Data JPA, permitiendo separar la lógica de negocio del acceso a datos.

- El sistema soporta validaciones, relaciones entre objetos y seguimiento histórico (por ejemplo, cambios de precios o búsquedas de usuarios registrados).

Código fuente

1. Clase AlertaPrecio

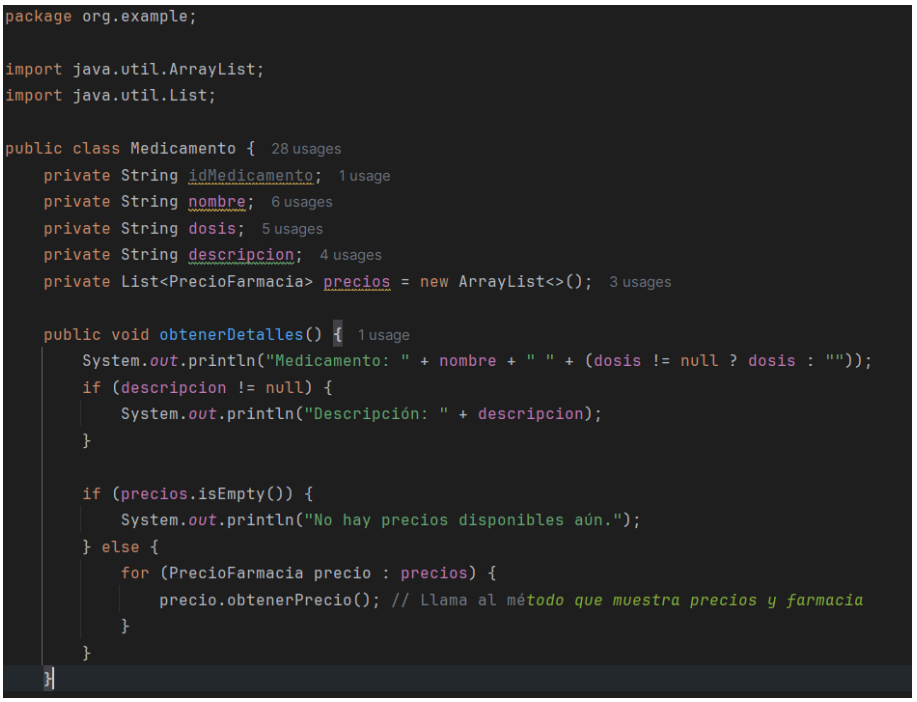
Encapsula el comportamiento y los datos necesarios para crear alertas personalizadas según un precio objetivo definido por el usuario. Aplica asociación con las clases Usuario y Medicamento, y mantiene el estado de la alerta:



```java
package org.example;

public class AlertaPrecio {  9 usages
    private Usuario usuario;  3 usages
    private Medicamento medicamento;  5 usages
    private double precioObjetivo;  4 usages
    private boolean alertaActiva;  5 usages

    public AlertaPrecio(Usuario usuario, Medicamento medicamento, double precioObjetivo) {  3 usages
        this.usuario = usuario;
        this.medicamento = medicamento;
        this.precioObjetivo = precioObjetivo;
        this.alertaActiva = false;
    }

    // Método para verificar si el precio ha bajado del precio objetivo
    public void verificarPrecio(double precioActual) {  3 usages
        if (precioActual < precioObjetivo && !alertaActiva) {
            alertaActiva = true;
            System.out.println("¡Alerta! El precio de " + medicamento.getNombre() + " ha bajado de " +
                    precioObjetivo + " a " + precioActual + ". ¡Es momento de comprar!");
        }
    }
```

2. Clase Medicamento

Representa un modelo de medicamento con atributos esenciales como nombre, dosis y una lista de precios. Aplica composición mediante la inclusión de objetos PrecioFarmacia, y provee métodos para mostrar detalles del medicamento, respetando el principio de responsabilidad única.



```java
package org.example;

import java.util.ArrayList;
import java.util.List;

public class Medicamento {  28 usages
    private String idMedicamento;  1 usage
    private String nombre;  6 usages
    private String dosis;  5 usages
    private String descripcion;  4 usages
    private List<PrecioFarmacia> precios = new ArrayList<>();  3 usages

    public void obtenerDetalles() {  1 usage
        System.out.println("Medicamento: " + nombre + " " + (dosis != null ? dosis : ""));
        if (descripcion != null) {
            System.out.println("Descripción: " + descripcion);
        }

        if (precios.isEmpty()) {
            System.out.println("No hay precios disponibles aún.");
        } else {
            for (PrecioFarmacia precio : precios) {
                precio.obtenerPrecio(); // Llama al método que muestra precios y farmacia
            }
        }
    }
```

3. Clase Farmacia

Simula el comportamiento de una farmacia en el sistema, incluyendo la capacidad de obtener medicamentos mediante scraping (simulado), y gestionar precios relacionados. Se observa la correcta separación de responsabilidades.

```java
package org.example;

import java.util.ArrayList;
import java.util.List;

public class Farmacia {  7 usages
    private String idFarmacia;  no usages
    private String nombre;  3 usages
    private Ubicacion ubicacion;  2 usages
    private List<Medicamento> listaMedicamentos = new ArrayList<>(); //  INICIALIZADA  4 usages
    private List<PrecioFarmacia> precios = new ArrayList<>();  2 usages

    public Farmacia(String nombre, Ubicacion ubicacion){  no usages
        this.nombre = nombre;
        this.ubicacion = ubicacion;
    }

    public void obtenerMedicamentos(){  no usages

        System.out.println("Obteniendo medicamentos de la página web de la farmacia: " + nombre);

        // Simulamos que se extrajeron estos medicamentos
        listaMedicamentos.add(new Medicamento( nombre: "Paracetamol",  dosis: "500mg"));
        listaMedicamentos.add(new Medicamento( nombre: "Ibuprofeno",  dosis: "400mg"));
        listaMedicamentos.add(new Medicamento( nombre: "Amoxicilina",  dosis: "500mg"));

        System.out.println("Medicamentos obtenidos correctamente.");
    }
```

4. Clase PrecioFarmacia

Permite mantener un historial de precios para un medicamento en una farmacia específica. Usa asociación bidireccional con Medicamento y Farmacia.

```java
package org.example;

public class PrecioFarmacia {  9 usages
    private Farmacia farmacia;  4 usages
    private Medicamento medicamento;  4 usages
    private double precioNormal;  4 usages
    private double precioOferta;  4 usages

    public PrecioFarmacia(Farmacia farmacia, Medicamento medicamento, double precioNormal, double precioOferta){
        this.farmacia=farmacia;
        this.medicamento=medicamento;
        this.precioNormal=precioNormal;
        this.precioOferta=precioOferta;
    }

    public void obtenerPrecio(){  2 usages
        System.out.println("Farmacia: " + farmacia.getNombre());
        System.out.println("Medicamento: " + medicamento.getNombre());
        System.out.println("Precio normal: $" + precioNormal);
        System.out.println("Precio oferta: $" + precioOferta);
    }
```

5. Clase HistorialPrecioFarmacia

Muestra el uso de una clase contenedora con una clase interna RegistroPrecio, para almacenar un historial de precios con marca temporal. Este diseño promueve la encapsulación y la trazabilidad.

```java
package org.example;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

public class HistorialPrecioFarmacia {  no usages
    private String idHistorial;  no usages
    private List<RegistroPrecio> historialPrecios = new ArrayList<>();  3 usages

    // Clase interna para guardar el precio en una fecha específica
    public static class RegistroPrecio {  5 usages
        private PrecioFarmacia precioFarmacia;  2 usages
        private LocalDateTime fecha;  2 usages

        public RegistroPrecio(PrecioFarmacia precioFarmacia, LocalDateTime fecha) {  1 usage
            this.precioFarmacia = precioFarmacia;
            this.fecha = fecha;
        }

        public void mostrarRegistro() {  1 usage
            System.out.println("Fecha: " + fecha);
            precioFarmacia.obtenerPrecio();
        }
    }

    public void guardarPrecio(PrecioFarmacia precioActual) {  no usages
        RegistroPrecio nuevoRegistro = new RegistroPrecio(precioActual, LocalDateTime.now());
        historialPrecios.add(nuevoRegistro);
        System.out.println("Precio guardado en historial.");
    }
```

6. Clase Favorito

Ejemplo de asociación entre entidades y generación de un ID único. Representa el almacenamiento personalizado de medicamentos por parte de los usuarios.

```java
package org.example;

public class Favorito {  10 usages
    private String idFavorito;  2 usages
    private Usuario usuario;  2 usages
    private Medicamento medicamento;  2 usages

    public Favorito(Usuario usuario, Medicamento medicamento) {  2 usages
        this.usuario = usuario;
        this.medicamento = medicamento;
        this.idFavorito = generarIdUnico(); // Genera un ID único para el favorito
    }

    private String generarIdUnico() {  1 usage
        return "FAV-" + System.currentTimeMillis(); // ID único basado en la hora
    }
```

7. Clase Busqueda y HistorialBusqueda

Reflejan la lógica de consultas realizadas por el usuario, demostrando cómo una búsqueda queda registrada junto a la información de ubicación, farmacia y término.

```java
package org.example;

public class Busqueda {  3 usages
    private String termino;  4 usages
    private Ubicacion ubicacion;  4 usages
    private Farmacia farmacia;  5 usages

    public void realizarBusqueda() {  no usages
        System.out.println("Realizando búsqueda de: " + termino);

        if (ubicacion != null) {
            System.out.println("Ubicación de búsqueda: " + ubicacion.getCiudad());
        } else {
            System.out.println("No se especificó ubicación.");
        }

        if (farmacia != null) {
            System.out.println("Buscando en la farmacia: " + farmacia.getNombre());
            for (Medicamento medicamento : farmacia.getListaMedicamentos()) {
                if (medicamento.getNombre().toLowerCase().contains(termino.toLowerCase())) {
                    System.out.println("Medicamento encontrado: " + medicamento.getNombre());
                    medicamento.obtenerDetalles();
                }
            }
        } else {
            System.out.println("No se especificó farmacia.");
        }
    }
```

```java
package org.example;

import java.util.Date;

public class HistorialBusqueda {  no usages
    private Usuario usuario;  3 usages
    private Date fecha;  3 usages
    private Busqueda busqueda;  7 usages

    public HistorialBusqueda(Usuario usuario, Busqueda busqueda) {  no usages
        this.usuario = usuario;
        this.busqueda = busqueda;
        this.fecha = new Date(); // Se asigna la fecha actual al momento de crear el historial
    }

    public void registrarBusqueda() {  no usages
        System.out.println("Registrando búsqueda realizada por: " + usuario.getNombre());
        System.out.println("Fecha: " + fecha);
        System.out.println("Término buscado: " + busqueda.getTermino());
        if (busqueda.getUbicacion() != null) {
            System.out.println("Ubicación: " + busqueda.getUbicacion().getCiudad());
        }
        if (busqueda.getFarmacia() != null) {
            System.out.println("Farmacia: " + busqueda.getFarmacia().getNombre());
        }
    }
```

Pruebas Unitarias

Estrategia de Pruebas

Para garantizar el correcto funcionamiento del sistema y la fiabilidad de las funcionalidades implementadas, se adoptó una estrategia de pruebas unitarias automatizadas utilizando el framework JUnit 5. Esta estrategia se enfoca en probar unidades individuales de código (clases y métodos) de forma aislada, validando que cada componente funcione como se espera ante distintos escenarios.

Las pruebas se implementaron utilizando el entorno de desarrollo IntelliJ IDEA, con soporte para JUnit integrado. En casos donde fue necesario simular objetos externos, se utilizó Mockito como herramienta de mocking para facilitar pruebas independientes.

Cada clase clave del sistema fue acompañada de una clase de prueba correspondiente, validando casos normales, casos límite y manejo de errores.

Casos de Prueba y Resultados

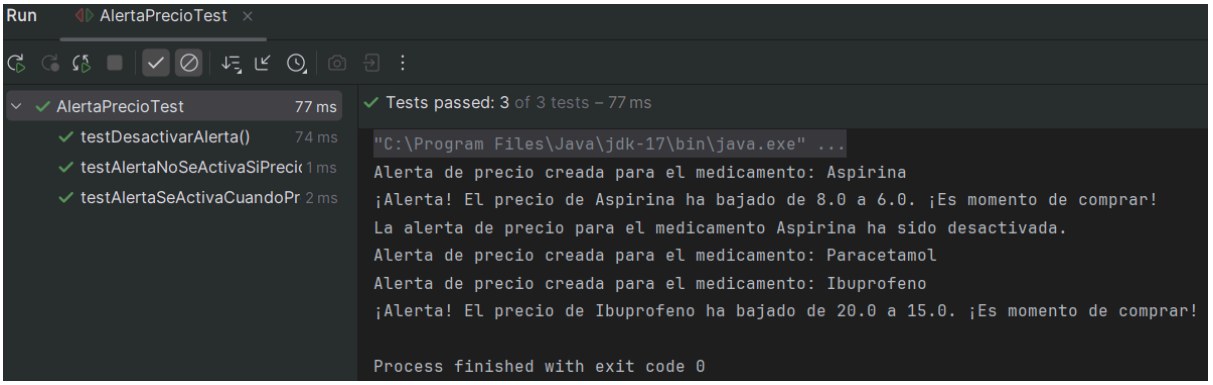
A continuación se muestran ejemplos representativos de casos de prueba implementados:

Clase: AlertaPrecioTest

```java
import org.example.AlertaPrecio;
import org.example.Medicamento;
import org.example.Usuario;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class AlertaPrecioTest {
    @Test
    public void testAlertaSeActivaCuandoPrecioEsMenorAlObjetivo() {
        Usuario usuario = new Usuario( nombre: "Carlos");
        Medicamento medicamento = new Medicamento( nombre: "Ibuprofeno");
        AlertaPrecio alerta = new AlertaPrecio(usuario, medicamento, precioObjetivo: 20.0);

        alerta.crearAlerta();
        alerta.verificarPrecio( precioActual: 15.0);

        assertTrue(alerta.isAlertaActiva());
    }

    @Test
    public void testAlertaNoSeActivaSiPrecioEsMayorOIgualAlObjetivo() {
        Usuario usuario = new Usuario( nombre: "Ana");
        Medicamento medicamento = new Medicamento( nombre: "Paracetamol");
        AlertaPrecio alerta = new AlertaPrecio(usuario, medicamento, precioObjetivo: 10.0);

        alerta.crearAlerta();
        alerta.verificarPrecio( precioActual: 12.0);

        assertFalse(alerta.isAlertaActiva());
    }
```

```java
    @Test
    public void testDesactivarAlerta() {
        Usuario usuario = new Usuario( nombre: "Juan");
        Medicamento medicamento = new Medicamento( nombre: "Aspirina");
        AlertaPrecio alerta = new AlertaPrecio(usuario, medicamento,  precioObjetivo: 8.0);

        alerta.crearAlerta();
        alerta.verificarPrecio( precioActual: 6.0); // activa
        alerta.desactivarAlerta();

        assertFalse(alerta.isAlertaActiva());
    }
}

```

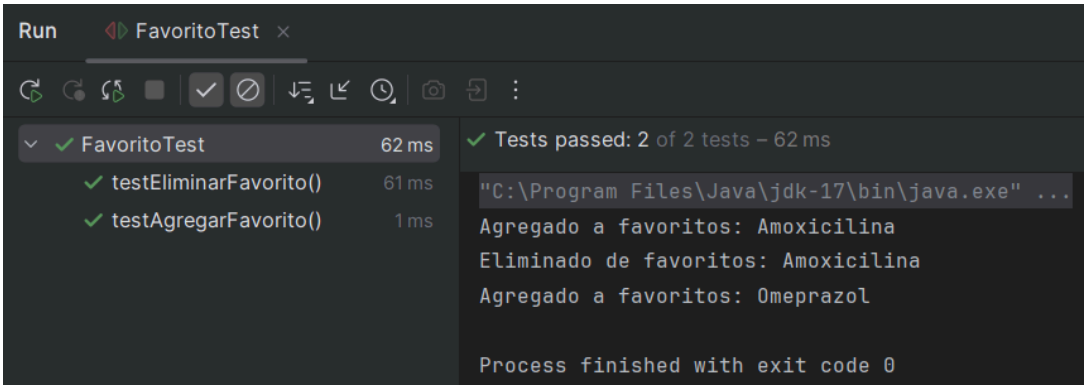
Salida



Clase: FavoritoTest

```java
import org.example.Favorito;
import org.example.Medicamento;
import org.example.Usuario;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class FavoritoTest {

    @Test
    public void testAgregarFavorito() {
        Usuario usuario = new Usuario( nombre: "Luis");
        Medicamento medicamento = new Medicamento( nombre: "Omeprazol");
        Favorito favorito = new Favorito(usuario, medicamento);

        usuario.agregarAFavoritos(favorito);

        assertTrue(usuario.getFavoritos().contains(favorito));
    }

    @Test
    public void testEliminarFavorito() {
        Usuario usuario = new Usuario( nombre: "Marta");
        Medicamento medicamento = new Medicamento( nombre: "Amoxicilina");
        Favorito favorito = new Favorito(usuario, medicamento);

        usuario.agregarAFavoritos(favorito);
        usuario.eliminarDeFavoritos(favorito);

        assertFalse(usuario.getFavoritos().contains(favorito));
    }
}
```

Salida:



Conclusión

En conclusión, la separación clara de responsabilidades entre clases no solo mejora la mantenibilidad del código, sino que también facilita la depuración de errores. Es fundamental inicializar siempre listas como List<Favorito>, List<AlertaPrecio>, entre otras, para prevenir errores de NullPointerException. Además, las pruebas unitarias deben actualizarse regularmente para reflejar cambios en la lógica de negocio o la estructura de las clases. Mantener clases como Favorito como simples entidades de relación, sin lógica de negocio adicional, asegura que la clase principal, como Usuario, mantenga un control coherente sobre sus datos. Este enfoque prepara adecuadamente el terreno para el uso futuro de frameworks como Spring Data JPA, donde las relaciones se pueden gestionar fácilmente mediante anotaciones como @OneToMany y @ManyToOne.