Segundo Parcial.

Juan Jose Moncayo Martinez
David Esteban Barreto Alban

Ingeniería de Software ||

Profesor:
Jonathan Lopez Londono

Universidad Autónoma de Occidente

11/04/25

Sistema de Gestión de Listas de Compras – Relato del Desarrollo

Pensamos que era buena idea construir una aplicación web que permitiera gestionar listas de compras de manera práctica y eficiente. La idea es ofrecer funcionalidades esenciales, como la creación y edición de productos, marcar qué ítems ya fueron comprados, mostrar estadísticas en tiempo real y enviar notificaciones automáticas cada vez que se produjeran cambios relevantes , escogimos este tema porque es algo que todas las personas hacemos en la vida cotidiana y que a pesar de ser muy común tenemos problemas al ejercer esta tarea por ejemplo : Se nos olvida comprar algo que es importante, compramos algo que ya tenemos en casa , vamos al supermercado y nos percatamos que hicieron cambios en precios o en el lugar donde ponen habitualmente los productos etc.

Para lograr una solución sólida y mantenible, decidimos estructurar el proyecto utilizando una **arquitectura por capas**, lo cual permite separar claramente las responsabilidades del sistema. Además, integramos dos patrones de diseño clave: **Singleton** y **Observer**. Esto no solo mejora la organización interna del código, sino que también nos ayuda a evitar errores comunes, como el uso excesivo del paso de propiedades (conocido como *Prop Drilling*) y la acumulación de código sin propósito claro (el *anti-patrón Lava Flow*) , sin embargo por temas académicos debemos utilizar estos antipatrones.

El sistema se dividió en **tres capas principales**:

La Capa de Dominio (Lógica Central)

Aquí se concentra la lógica principal de la aplicación. Se diseñaron modelos que representarán nuestros datos de forma clara y reutilizable. Uno de los elementos centrales fue el **modelo de Producto**, que encapsula los atributos básicos de cada ítem: su nombre, la cantidad necesaria y su estado (comprado o no).

Además, se implementó un **observador**, que actúa como sistema de notificación para cualquier cambio en la lista. Gracias a este componente, distintos elementos del sistema pueden mantenerse sincronizados sin necesidad de estar directamente conectados entre sí.

La Capa de Servicios

En esta capa se centralizaron las operaciones importantes y el manejo del estado global. Se construyó un **Servicio de Lista**, que fue implementado siguiendo el patrón **Singleton**. Esto significa que solo existe una única instancia de este servicio durante toda la vida de la aplicación.

El objetivo fue asegurar un punto de acceso unificado a los datos. Así, cada vez que se agregaba un producto, se actualizaban estadísticas o se cambiaba el estado de un ítem, todo ocurre desde un solo lugar, evitando inconsistencias y duplicaciones innecesarias.

La Capa de Presentación

Esta parte del sistema se enfoca en la interfaz de usuario. Fue diseñada para ser clara, intuitiva y reactiva ante los cambios.
Entre los componentes principales se encuentran:

- La **Aplicación Principal**, que se encargó de coordinar la comunicación entre las diferentes piezas.
- Un **Formulario**, para que los usuarios pudieran añadir nuevos productos fácilmente.
- La **Lista de Productos**, donde se muestran los ítems de forma interactiva.
- Y un **Panel de Estadísticas**, que ofrece una vista general del progreso, mostrando en tiempo real cuántos productos han sido comprados y cuántos aún están pendientes.

¿Qué logramos con los patrones de diseño?

Singleton

Este patrón sirve para mantener un estado **consistente** en toda la aplicación. Al tener un solo servicio encargado de gestionar la lista, evitamos errores comunes como tener datos desactualizados en distintos componentes. Además, facilitó la sincronización y redujo el esfuerzo necesario para manejar el flujo de información.

Observer

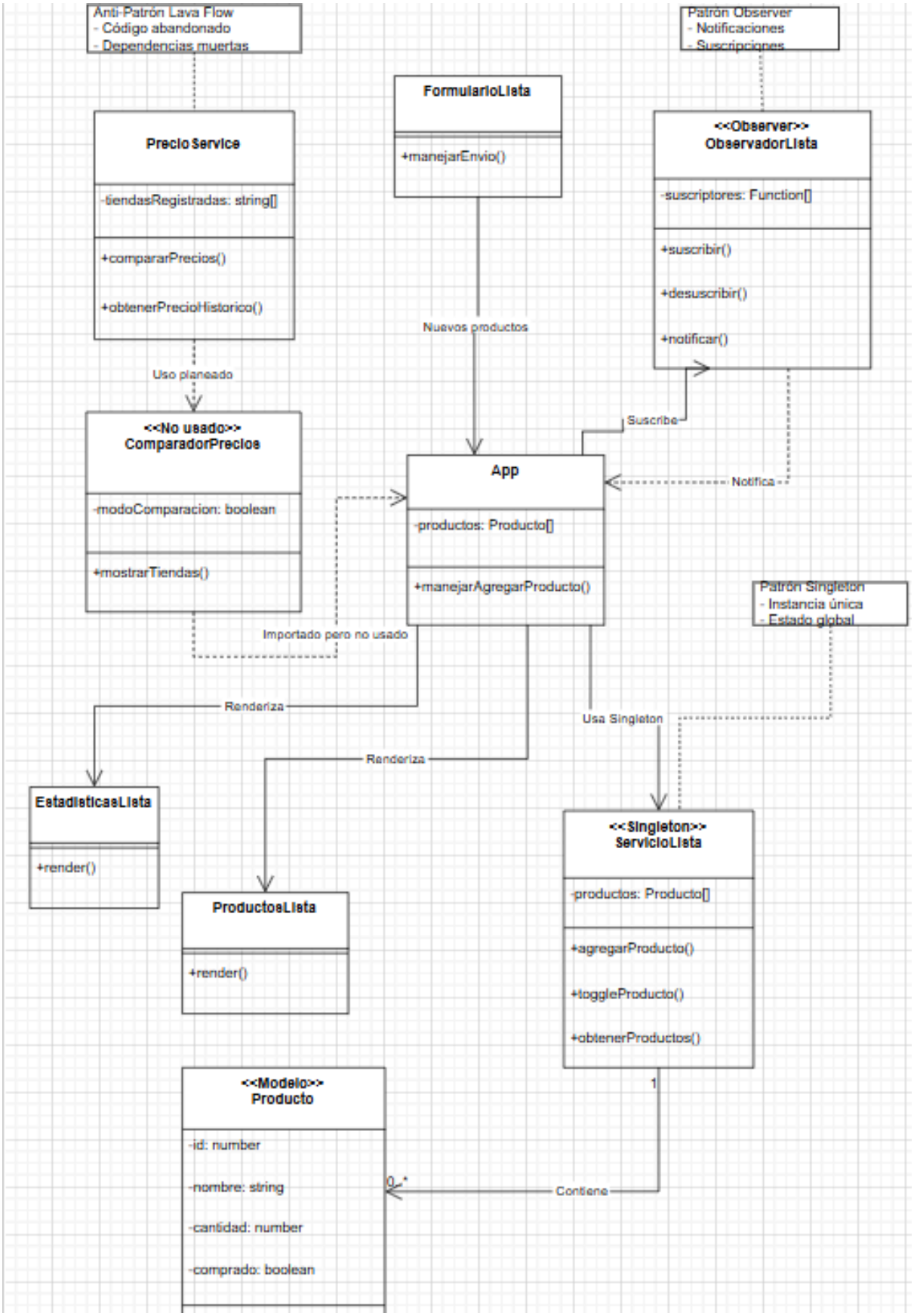
Gracias a este patrón, la aplicación se volvió mucho más **reactiva**. Cualquier cambio en los datos como añadir un nuevo producto o marcar uno como comprado es detectado de inmediato, y los componentes afectados (como la lista de productos o las estadísticas) se actualizan automáticamente. Esto permitió que el sistema fuera más dinámico, eficiente y **desacoplado**, lo cual es fundamental para mantener un código limpio y escalable.

Análisis de Anti-Patrones

Lava Flow
Acumula código abandonado (servicio de precios y componente de comparación) y dependencias no utilizadas, lo que produce :

- Aumento del tamaño del bundle.
- Confusión en el mantenimiento del código.

Diagrama de Clases



Elementos clave:

- Clases principales (Producto, ServicioLista, ObservadorLista).
- Componentes de UI(Interfaz) (App, ProductosLista, FormularioLista).

Relaciones:

- Singleton: `ServicioLista` como eje central de datos.
- Observer: `ObservadorLista` notificando cambios a `App`.
- Lava Flow: Clases no utilizadas, mal realizadas (`PrecioService`, `ComparadorPrecios`) con relaciones fantasmas.

Síntesis :

1. Importancia de la arquitectura:
    - La separación en capas mejora la mantenibilidad y facilita la localización de errores.
2. Ventajas de los patrones:
    - Singleton y Observer demostraron ser esenciales para gestionar estado y comunicaciones complejas.
3. Riesgos de los anti-patrones:
    - Lava Flow degradan temporalmente la calidad del código, evidenciando la necesidad de revisiones constantes.

Conclusión:

Este proyecto fue una gran oportunidad para aplicar buenas y malas prácticas de desarrollo mientras resolvemos un problema real y cotidiano. Logramos construir una aplicación funcional, organizada y fácil de usar, gracias a una arquitectura clara, uso de patrones como Singleton, Observer y uso del antipatrón “Lava-flow”

Más allá de lo técnico, lo importante es que terminamos con una solución útil, bien estructurada. Una app sencilla, pero hecha con cuidado y pensando en cada detalle.

Github: https://github.com/DavidBar24/Parcial2IngeSoft2.git

Diagrama:
https://drive.google.com/file/d/19qPCeEsjiDf4Fr7S_GnrFtfn4E5PNy3k/view?usp=sharing