# INFORME ACADÉMICO / TÉCNICO

| | |
|---|---|
| **Título del Informe:** | Creación y ejecución de un proyecto. |
| **Autor(es):** | Jeffrey Manobanda, Jhordy Marcillo |
| **Carrera:** | Ingeniería en Software |
| **Asignatura o Proyecto:** | Desarrollo de aplicaciones móviles |
| **Tutor o Supervisor:** | Ing. Doris Chicaiza |
| **Institución:** | Universidad de las Fuerzas Armadas ESPE |
| **Fecha de entrega:** | 22 de octubre de 2025 |

# Índice

# Índice de figuras

# 1 Introducción

El siguiente informe, presenta la actividad realizada durante el desarrollo de creación y ejecución de 5 ejercicios sobre calcular diferentes escenarios para desarrollar mas la lógica de programación con respecto al entorno de Flutter y de Android Studio. El trabajo se enmarca dentro del campo de la ingeniera de software al realizar un desarrollo de diferentes aplicaciones.
Las herramientas que se utilizaron para desarrollar esta actividad principalmente fue el IDE de Android Studio, ayudado de herramientas para la simular un dispositivo móvil y del lenguaje de desarrollo de dart, los cuales nos van ayudar a tener un mejor desarrollo de la aplicación ayudándonos de ver los progresos en un entorno simulado a dispositivos móviles.
En este informe, se concluyó de manera exitosa los diferentes ejercicios propuestos, indicando de esta manera un buen reforzamiento de términos y lógica de programación y reforzar la familiaridad de estas tecnologías para dispositivos móviles

# 2 Objetivos

## 2.1 Objetivo General

Desarrollar un proyecto técnico aplicando los conocimientos adquiridos en el área de desarrollo de aplicaciones móviles, para implementar un aplicativo de diferentes funcionalidades mediante herramientas de desarrollo libre como Android Studio para reforzar los conocimientos y familiaridad con el lenguaje de dart

## 2.2 Objetivos Específicos

- Diseñar la arquitectura del sistema aplicando el modelo MVC.
- Implementar los módulos de los diferentes sistemas propuestos en el documento
- Validar la funcionalidad y usabilidad del sistema
- Validar la interfaz de la aplicación siendo intuitiva y familiar.

# 3 Marco Teórico

## 3.1 Patrón Modelo-Vista-Controlador (MVC)

El patrón MVC es una arquitectura de software ampliamente utilizada en aplicaciones de interfaz gráfica, tanto web como móviles. Su principal objetivo es separar las responsabilidades del código para mejorar su organización:

- **Modelo (Model):** Representa la estructura de los datos y la lógica de negocio.
  - Almacena información, realiza cálculos y validaciones.
  - No conoce detalles de la interfaz gráfica ni de la forma en que los datos se mostrarán.

  - **Ejemplo:** en un ejercicio de conversión de unidades, el modelo calcula yardas, pies, pulgadas o centímetros a partir de una medida en metros.

- **Vista (View):** Es responsable de presentar la información al usuario y de capturar sus acciones.

  - Puede incluir botones, campos de entrada, listas y otros widgets visuales.
  - Se comunica con el controlador para obtener los datos procesados
  - **Ejemplo:** en un ejercicio de descuento en un supermercado, la vista muestra los campos de ingreso del total de la compra y el número elegido, y luego despliega el descuento aplicado.

- **Controlador (Controller):** Actúa como intermediario entre la vista y el modelo.

  - Recibe las acciones del usuario desde la vista, invoca métodos del modelo para procesar datos y devuelve los resultados para ser presentados.
  - Permite que la vista permanezca independiente de la lógica del negocio y que el modelo permanezca independiente de la presentación

  La implementación de MVC permite modularizar el código, haciendo que cada componente pueda ser modificado sin afectar a los demás. Por ejemplo, cambiar el diseño visual de un ejercicio no implica modificar los cálculos que se realizan en el modelo, y viceversa.

## Aplicación en Flutter

Flutter es un framework de desarrollo de aplicaciones móviles de Google que permite crear interfaces nativas para Android e iOS con un único código base. Su lenguaje principal es **Dart**, un lenguaje orientado a objetos que combina características de programación estructurada y funcional.

En Flutter, cada pantalla o componente visual se representa mediante un *widget*, que puede ser *stateful* (con estado interno) o *stateless* (sin estado). En el contexto de la arquitectura **MVC**:

- Cada ejercicio se implementa como un *widget stateful*, que actúa como la vista.
- Los controladores son clases de Dart que gestionan la interacción y llaman al modelo correspondiente.
- Los modelos son clases puras que contienen variables y métodos para realizar cálculos, transformaciones o validaciones.

Este enfoque permite que los desarrolladores puedan agregar nuevos ejercicios o modificar los existentes de manera aislada, lo que resulta muy útil en aplicaciones educativas o de simulaciones, donde se requiere probar múltiples escenarios de cálculo.

### Aplicaciones prácticas de los ejercicios

Los ejercicios implementados, tales como:

1. Conversión de unidades métricas
2. Conversión de capacidades de almacenamiento
3. Cálculo de cuotas financieras
4. Promedios de producción y verificación de incentivos
5. Cálculo de descuentos en supermercados

permiten a los estudiantes y usuarios interactuar con problemas cotidianos de matemáticas y finanzas, aplicando los principios de aritmética, proporciones y porcentajes.

Cada ejercicio sigue el patrón **MVC**:

- **Modelo:** realiza los cálculos de forma confiable.
- **Controlador:** recibe la entrada del usuario y devuelve los resultados del modelo a la vista.
- **Vista:** muestra los resultados de manera visual y atractiva, con campos de entrada y botones interactivos.

Esto no solo facilita la enseñanza y el aprendizaje de los conceptos matemáticos, sino que también fortalece las habilidades en programación orientada a objetos y en el diseño de interfaces.

# 4 Desarrollo

Para comenzar, se empezó realizando las diferentes carpetas de cada uno de los ejercicios, al utilizar el modelo de MVC, es pertinente que cada uno de los ejercicios se separen siguiendo este modelo de programación como se visualiza en la Ilustración 1.

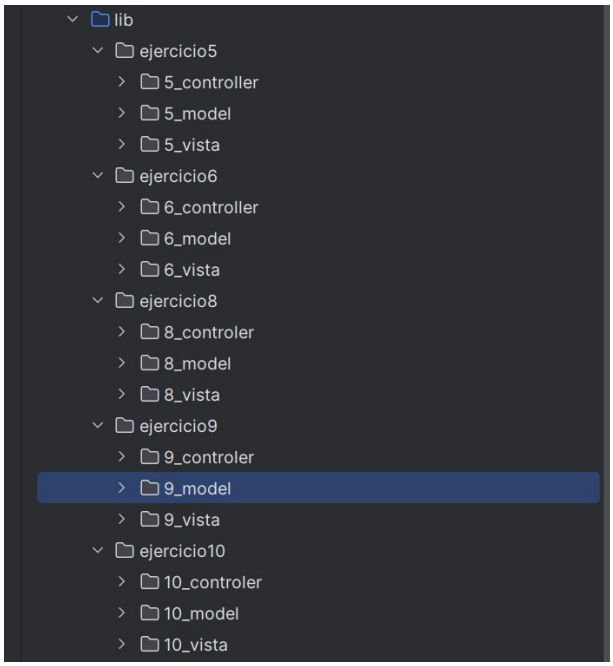



Figura 1: Creación de carpetas.

Siguiendo se desarrolló los diferentes archivos dentro de cada una de las carpetas para que de esta manera sea mas modular y sea mas sencillo hacer cambios posteriormente, esto se hace en cada una de las carpetas creadas como se puede visualizar en la ilustración 2.

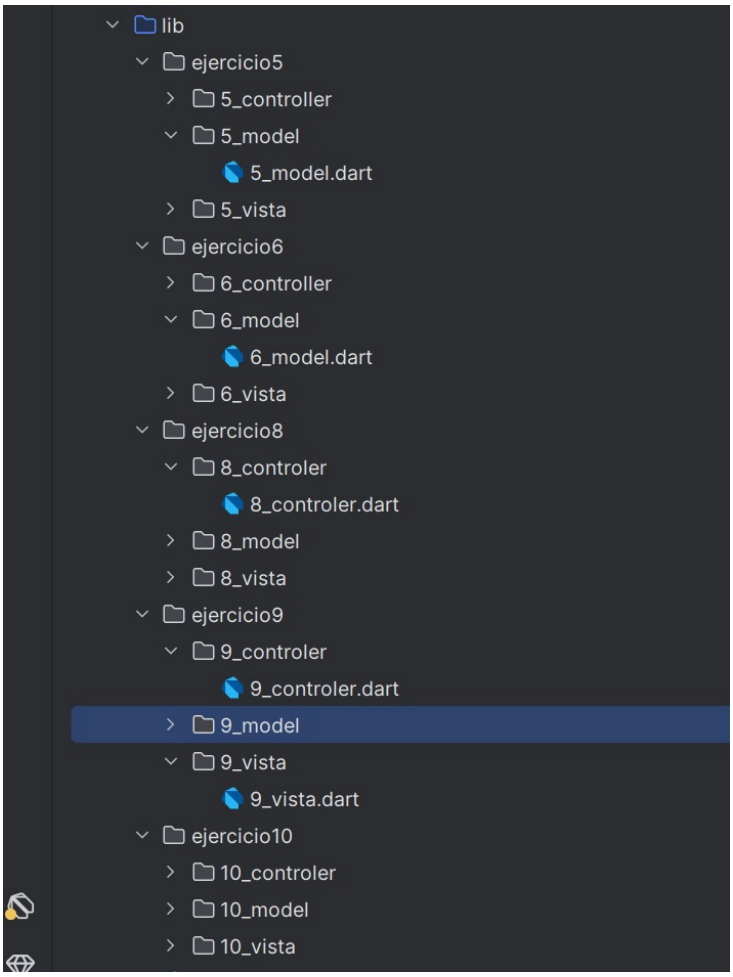
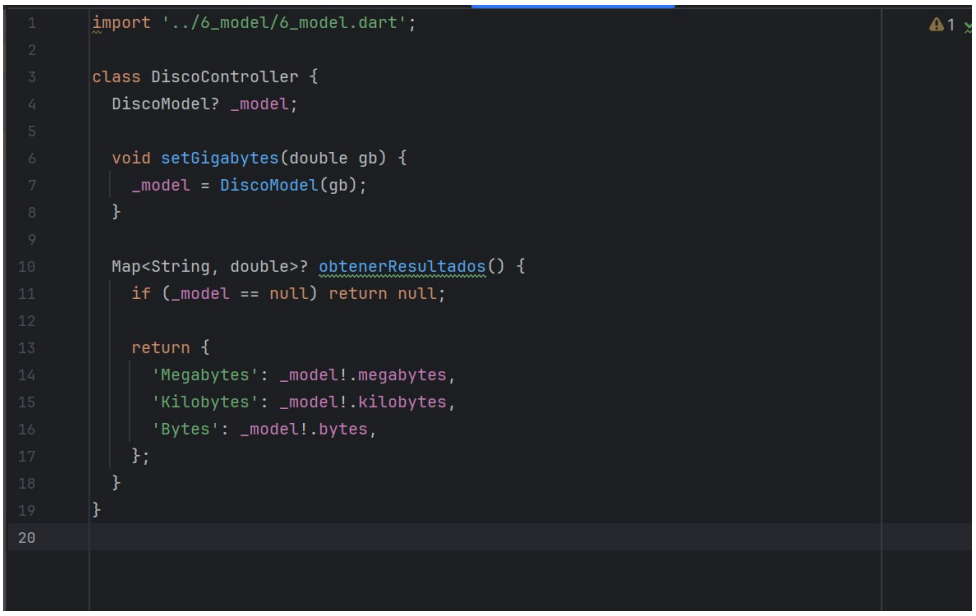



Figura 2: Archivos dentro de cada carpeta.

Para continuar con el desarrollo del proyecto de manera organizada y eficiente, se creó un controlador para cada uno de los ejercicios. En el controlador se maneja principalmente la lógica de procesamiento de cada ejercicio, actuando como intermediario entre la vista y el modelo.
En el modelo se declaran todas las variables y métodos necesarios para los cálculos o transformaciones de datos, asegurando que la lógica del negocio esté separada de la interfaz.
Finalmente, en la vista se define cómo se organizará la interfaz de usuario, incluyendo campos de entrada, botones, listas y presentación de resultados. Esta estructura es consistente y se aplica de manera uniforme a todos los módulos existentes en el proyecto.



```dart
import '../6_model/6_model.dart';

class DiscoController {
  DiscoModel? _model;

  void setGigabytes(double gb) {
    _model = DiscoModel(gb);
  }

  Map<String, double>? obtenerResultados() {
    if (_model == null) return null;

    return {
      'Megabytes': _model!.megabytes,
      'Kilobytes': _model!.kilobytes,
      'Bytes': _model!.bytes,
    };
  }
}
```

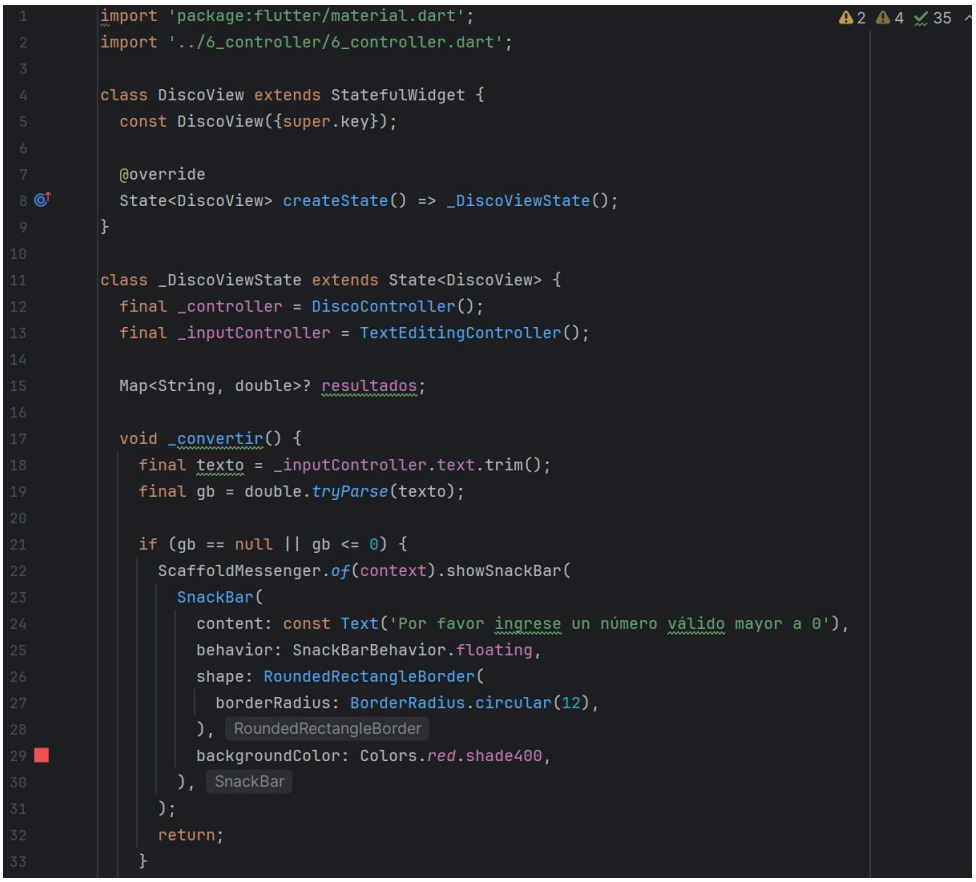
Figura 3: Controlador del ejercicio 6.

```dart
class DiscoModel {
  final double gigabytes;

  DiscoModel(this.gigabytes);

  double get megabytes => gigabytes * 1024;
  double get kilobytes => megabytes * 1024;
  double get bytes => kilobytes * 1024;
}
```

Figura 4: Modelo del ejercicio 6.



```dart
import 'package:flutter/material.dart';
import '../6_controller/6_controller.dart';

class DiscoView extends StatefulWidget {
  const DiscoView({super.key});

  @override
  State<DiscoView> createState() => _DiscoViewState();
}

class _DiscoViewState extends State<DiscoView> {
  final _controller = DiscoController();
  final _inputController = TextEditingController();

  Map<String, double>? resultados;

  void _convertir() {
    final texto = _inputController.text.trim();
    final gb = double.tryParse(texto);

    if (gb == null || gb <= 0) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: const Text('Por favor ingrese un número válido mayor a 0'),
          behavior: SnackBarBehavior.floating,
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(12),
          ),
          backgroundColor: Colors.red.shade400,
        ),
      );
      return;
    }
```

Figura 5: Vista del ejercicio 6.

# 5 Conclusiones y Recomendaciones

## Conclusiones

- La implementación del patrón Modelo-Vista-Controlador (MVC) permitió organizar el proyecto en módulos independientes y claramente diferenciados, donde cada componente tiene un rol específico: los modelos se encargan de los datos y cálculos, los controladores gestionan la lógica y las vistas muestran la información al usuario. Esta separación facilita la comprensión del flujo de datos, reduce la posibilidad de errores al modificar funciones individuales y permite que los desarrolladores trabajen de manera más eficiente en diferentes partes del proyecto sin afectar otras áreas. Además, esta estructura promueve buenas prácticas de programación orientada a objetos y hace que la aplicación sea más profesional y mantenible.
- Al organizar cada ejercicio como un módulo autónomo con su propio modelo y controlador, se logra que los componentes puedan ser reutilizados en futuros ejercicios o aplicaciones similares, lo que optimiza el tiempo de desarrollo y reduce la duplicación de código. Esta arquitectura también facilita la escalabilidad del proyecto, permitiendo agregar nuevas funcionalidades o ejercicios con un impacto mínimo sobre los módulos existentes. La modularidad, combinada

con la reutilización, contribuye a que la aplicación sea más flexible y adaptable a cambios o expansiones futuras.

- Separar la interfaz de la lógica de negocio permite que las vistas se centren en ofrecer una experiencia visual clara, interactiva e intuitiva, mientras que los cálculos y validaciones se realizan de forma consistente en los modelos y controladores. Esto asegura que los resultados mostrados al usuario sean precisos y confiables. Además, la consistencia en la presentación y la interacción contribuye a que los usuarios puedan ingresar datos sin confusión y obtener resultados inmediatos, mejorando la usabilidad general de la aplicación.

### Recomendaciones

- Es recomendable que todas las vistas sigan un diseño uniforme en términos de colores, tipografía, márgenes y disposición de widgets. Esto no solo mejora la apariencia profesional de la aplicación, sino que también facilita la navegación y comprensión por parte del usuario. Una interfaz clara y consistente permite que los usuarios identifiquen rápidamente los campos de entrada, los botones de acción y los resultados, evitando confusiones y errores al interactuar con la aplicación
- Se sugiere validar exhaustivamente los datos ingresados por el usuario en cada ejercicio, controlando entradas vacías, valores no numéricos o fuera de rango, y proporcionando retroalimentación inmediata mediante mensajes o alertas. Esto garantiza que la lógica del modelo reciba datos correctos, evitando errores en tiempo de ejecución y asegurando la confiabilidad de los resultados. Además, la validación mejora la experiencia del usuario al guiarlo para ingresar datos válidos de manera intuitiva.
- Cada módulo, controlador y modelo debe contar con documentación clara y comentarios que expliquen la función de los métodos y variables, lo que facilita la comprensión del código por parte de otros desarrolladores o estudiantes. Asimismo, se recomienda mantener una estructura de carpetas organizada y coherente, de manera que futuros ejercicios o funcionalidades puedan integrarse fácilmente sin alterar la arquitectura existente. Esta práctica asegura la sostenibilidad, la escalabilidad y la eficiencia del mantenimiento a largo plazo del proyecto.

## 6 Referencias Bibliográficas


- Google Developers. *Flutter Documentation.* Disponible en: `https://docs.flutter.dev`
- Fuente oficial de Flutter que explica la arquitectura, widgets, estado de la aplicación y buenas prácticas para el desarrollo de aplicaciones móviles multiplataforma con Dart

# 7 Anexos


Figura 6: Vista principal del ejercicio.


Figura 7: Ejercicio 5 - Conversor de Unidades.


Figura 8: Ejercicio 6 - Conversor de Capacidad de Disco.

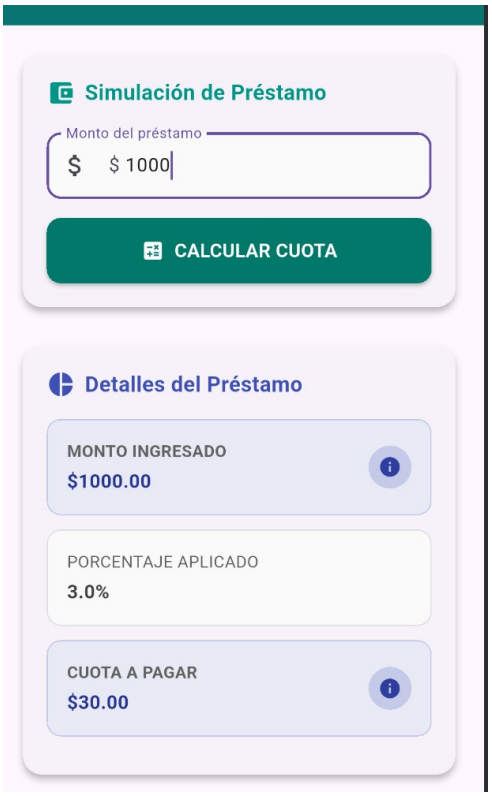



Figura 9: Ejercicio 8 - Calculadora de Préstamos.


Figura 10: Ejercicio 9 - Producción Semanal.

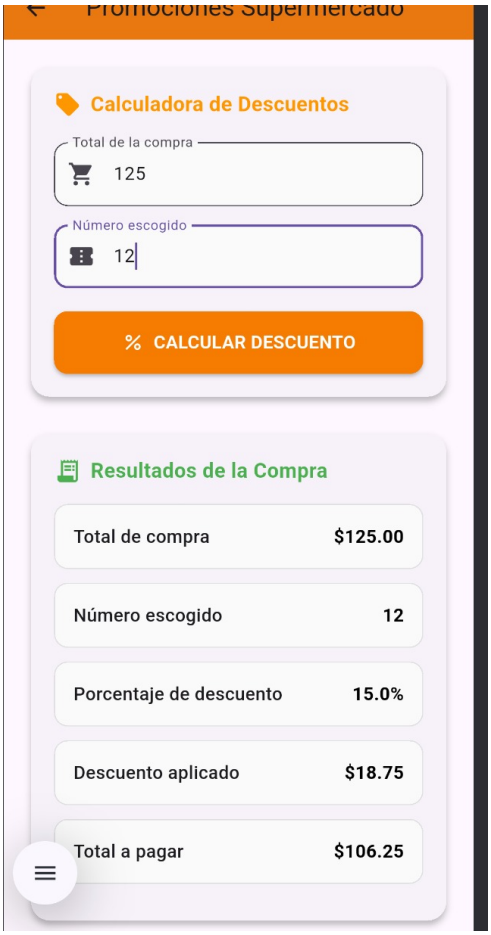



Figura 11: Ejercicio 10 - Calculadora de Descuentos.