INFORME ACADÉMICO / TÉCNICO

1. Datos Generales

Título del Informe:	Creación y ejecución de un proyecto
Autor(a):	Mesias Mariscal Bryan Quispe Gabriel Murillo
Carrera:	Ingeniería en Software
Asignatura o Proyecto:	Desarrollo De Aplicaciones Móviles
Tutor o Supervisor:	Doris Karina Chicaiza Angamarca
Institución:	Universidad de las Fuerzas Armadas ESPE – Sede Sangolquí
Fecha de entrega:	22 de octubre de 2025

Índice

1 Introducción

El desarrollo de aplicaciones móviles requiere arquitecturas que faciliten el mantenimiento y escalabilidad. El patrón Modelo-Vista-Controlador (MVC) proporciona separación clara entre lógica de negocio, presentación y control de flujo. Esta separación reduce el acoplamiento entre componentes. La adopción de patrones arquitectónicos es fundamental para software sostenible.

Este informe documenta la implementación del patrón MVC en cinco ejercicios desarrollados con Flutter y Android Studio. Se integró la metodología Atomic Design para organizar componentes de interfaz con consistencia visual. Ambas metodologías proporcionan un enfoque integral para arquitectura y diseño. La refactorización preservó la funcionalidad mejorando la estructura.

Los resultados demuestran que los patrones arquitectónicos mejoran significativamente la calidad del código. La transición hacia arquitecturas estructuradas beneficia la organización técnica y crecimiento del proyecto. Este trabajo es una guía práctica para implementar arquitecturas profesionales en Flutter. Las recomendaciones pueden aplicarse en proyectos de diversa complejidad.

1.1 Objetivos

1.1.1 Objetivo General

Implementar una aplicación móvil en Flutter que integre cinco ejercicios de programación bajo el patrón de arquitectura MVC, manteniendo la lógica original de cada ejercicio y aplicando principios de Atomic Design en la construcción de interfaces de usuario.

1.1.2 Objetivos Específicos

1. Desarrollar ejercicios para adaptarlos al patrón MVC sin alterar su lógica computacional.
2. Implementar componentes reutilizables mediante la metodología Atomic Design.
3. Desarrollar controladores que gestionen la validación de datos y coordinación entre modelo y vista.
4. Implementar interfaces de usuario que eviten problemas de desbordamiento visual.

2 Marco Teórico

2.1 Patrón Modelo-Vista-Controlador (MVC)

El patrón MVC, introducido por Trygve Reenskaug en la década de 1970, divide una aplicación en tres componentes interconectados (Gamma et al., 1994):

- **Modelo:** Representa los datos y la lógica de negocio de la aplicación. Es independiente de la interfaz de usuario y contiene los algoritmos y cálculos necesarios.
- **Vista:** Responsable de la presentación visual de los datos. Muestra información al usuario y captura sus interacciones.
- **Controlador:** Actúa como intermediario entre el Modelo y la Vista. Recibe las entradas del usuario, las valida, y coordina las actualizaciones del Modelo y la Vista.

2.2 Atomic Design

Atomic Design, propuesto por Brad Frost (2016), es una metodología para crear sistemas de diseño mediante la construcción de interfaces desde componentes pequeños hacia componentes más complejos:

1. **Átomos:** Componentes básicos e indivisibles (botones, campos de texto, etiquetas)
2. **Moléculas:** Grupos de átomos que funcionan juntos
3. **Organismos:** Componentes complejos formados por moléculas y átomos
4. **Plantillas:** Estructuras de página que organizan organismos
5. **Páginas:** Instancias específicas de plantillas con contenido real

2.3 Flutter y Dart

Flutter es un framework de código abierto desarrollado por Google para crear aplicaciones multiplataforma. Utiliza el lenguaje de programación Dart y se caracteriza por su arquitectura reactiva basada en widgets (Google LLC, 2023). La combinación de StatefulWidget y StatelessWidget permite crear interfaces dinámicas y eficientes.

3 Desarrollo

3.1 Arquitectura del Proyecto

La estructura del proyecto sigue una organización clara basada en el patrón MVC, dividiendo los componentes en tres capas principales: Model (lógica de negocio), Controller (validación y coordinación), y Vista (interfaz de usuario).

3.2 Implementación de la Capa Modelo

La capa Modelo contiene exclusivamente la lógica de negocio de cada ejercicio, sin dependencias de la interfaz de usuario.

3.2.1 Ejercicio 5: Conversión de Medidas Métricas

El modelo de conversión métrica implementa las fórmulas matemáticas para transformar metros en diferentes unidades:

```
class MetrosModel{
  // Convertir metros a centimetros
  double convertirACentimetros(double metros){
    return metros * 100;
  }

  // Convertir centimetros a pulgadas
  double convertirAPulgadas(double centimetros){
    return centimetros / 2.54;
  }

  // Convertir pulgadas a pies
  double convertirAPies(double pulgadas){
    return pulgadas / 12;
  }

  // Convertir pies a yardas
  double convertirAYardas(double pies){
    return pies / 3;
  }
}
```

Listing 1: Modelo de Conversión Métrica

La implementación respeta las conversiones estándar: 1 metro = 100 centímetros, 1 pulgada = 2.54 centímetros, 1 pie = 12 pulgadas, y 1 yarda = 3 pies.

3.2.2 Ejercicio 6: Conversión de Capacidad de Disco

Este modelo calcula la capacidad de almacenamiento en diferentes unidades, considerando que cada unidad superior contiene 1024 unidades inferiores:

```
class DiscoDuroModel{
  // Convertir Gigabytes a Megabytes
  double convertirAMegabytes(double gigabytes){
    return gigabytes * 1024;
  }

  // Convertir Megabytes a Kilobytes
  double convertirAKilobytes(double megabytes){
    return megabytes * 1024;
  }

  // Convertir Kilobytes a Bytes
  double convertirABytes(double kilobytes){
    return kilobytes * 1024;
  }
}
```

Listing 2: Modelo de Conversión de Disco Duro

3.2.3 Ejercicio 8: Cálculo de Cuotas de Seguros

El modelo de seguros aplica diferentes tasas de interés según el monto de la finanza:

```
class SegurosModel{
  // Calcular cuota segun el monto de finanza
  double calcularCuota(double monto){
    if(monto < 50000){
      return monto * 0.03; // 3% si es menor a $50000
    } else {
      return monto * 0.02; // 2% si es mayor o igual a $50000
    }
  }
}
```

Listing 3: Modelo de Seguros

3.2.4 Ejercicio 9: Cálculo de Comisiones

Este modelo calcula las comisiones del 10 % sobre tres ventas y el total mensual incluyendo el sueldo base:

```
class ComisionesModel{
  // Calcular comision por una venta (10%)
  double calcularComision(double venta){
    return venta * 0.10;
  }

  // Calcular total de comisiones de tres ventas
  double calcularTotalComisiones(double venta1,
                                 double venta2,
                                 double venta3){
    return calcularComision(venta1) +
           calcularComision(venta2) +
           calcularComision(venta3);
  }

  // Calcular total a recibir en el mes
  double calcularTotalMes(double sueldoBase,
                          double totalComisiones){
    return sueldoBase + totalComisiones;
  }
}
```

Listing 4: Modelo de Comisiones

3.2.5 Ejercicio 10: Sistema de Descuentos

El modelo de descuentos aplica porcentajes variables según un número aleatorio generado:

```
class DescuentoModel{
  // Calcular descuento segun el numero aleatorio
  double calcularDescuento(double total, int numero){
    if(numero < 74){
      return total * 0.15; // 15% de descuento
    } else {
      return total * 0.20; // 20% de descuento
    }
  }

  // Calcular total a pagar
  double calcularTotalAPagar(double total,
                             double descuento){
    return total - descuento;
  }
```

```
}
```

Listing 5: Modelo de Descuentos

3.3 Implementación de la Capa Controlador

Los controladores gestionan la validación de datos de entrada y coordinan las operaciones entre el modelo y la vista.

3.3.1 Patrón de Validación

Todos los controladores implementan un patrón de validación consistente:

1. Verificación de campos vacíos
2. Conversión segura de cadenas a números (tryParse)
3. Validación de valores negativos
4. Invocación del modelo correspondiente
5. Formateo y retorno de resultados

Ejemplo del controlador de metros:

```dart
import '../model/metros_model.dart';

class MetrosController{
  String calcular(String input){
    // Validar que el campo no este vacio
    if(input.trim().isEmpty){
      return 'Ingrese la medida en metros';
    }

    // Convertir texto a numero
    final metros = double.tryParse(input);

    // Validar que sea un numero valido
    if(metros == null){
      return 'Ingrese un numero valido';
    }

    // Validar que sea positivo
    if(metros < 0){
```

```
      return 'La medida debe ser positiva';
    }

    // Calcular usando el modelo
    final modelo = MetrosModel();
    final centimetros = modelo.convertirACentimetros(metros);
    final pulgadas = modelo.convertirAPulgadas(centimetros);
    final pies = modelo.convertirAPies(pulgadas);
    final yardas = modelo.convertirAYardas(pies);

    // Retornar resultado formateado
    return '''Centimetros: ${centimetros.toStringAsFixed(2)}
Pulgadas: ${pulgadas.toStringAsFixed(2)}
Pies: ${pies.toStringAsFixed(2)}
Yardas: ${yardas.toStringAsFixed(2)}''';
  }
}
```

Listing 6: Controlador de Conversión Métrica

3.4 Implementación de la Capa Vista

La capa Vista se implementó con dos enfoques diferentes según el ejercicio, para el 5 y 6 no se aplico un atomica design, mientras que para los ejercicios 8, 9 y 10 se aplicó Atomic Design:

3.4.1 Ejercicios 8, 9 y 10: Atomic Design

Estos ejercicios implementan la metodología Atomic Design completa:
Átomos:

- `LabelText`: Etiquetas con estilo consistente
- `NumberField`: Campos de entrada numérica
- `PrimaryButton`: Botones de acción
- `ResultText`: Texto de resultados

Moléculas:

- `SalaryInput`: Combinación de label y campo para sueldo
- `SaleInput`: Combinación de label y campo para ventas
- `FinanceInput`: Combinación de label y campo para finanzas

Organismos:

- `ComisionesCard`: Widget completo con estado para gestionar comisiones
- `SegurosCard`: Widget completo para cálculo de seguros
- `DescuentoCard`: Widget completo con generación de números aleatorios

Ejemplo de implementación de átomo:

```
class NumberField extends StatelessWidget{
  final TextEditingController controller;
  final String hint;

  const NumberField({super.key,
                     required this.controller,
                     required this.hint});

  @override
  Widget build(BuildContext context) {
    return TextField(
      controller: controller,
      keyboardType: TextInputType.number,
      decoration: InputDecoration(
        border: OutlineInputBorder(),
        hintText: hint,
      )
    );
  }
}
```

Listing 7: Átomo: NumberField

3.5 Gestión de Overflow

Para prevenir problemas de overflow cuando aparece el teclado virtual, se implementó `SingleChildScrollView` en las vistas con múltiples campos de entrada:

```
class ComisionesPage extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Comisiones de Ventas'),
        backgroundColor: Colors.blue,
```

```
      ),
      body: SingleChildScrollView(
        child: Padding(
          padding: EdgeInsets.all(10),
          child: ComisionesCard(),
        ),
      ),
    );
  }
}
```

Listing 8: Implementación de Scroll Dinámico

3.6 Menú de Navegación

La aplicación incluye un menú principal que utiliza el sistema de navegación de Flutter mediante `Navigator.push`:

```
class MenuPrincipal extends StatelessWidget{
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Menu de Ejercicios'),
        backgroundColor: Colors.cyan,
      ),
      body: Padding(
        padding: EdgeInsets.all(20),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              onPressed: (){
                Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (context) => MetrosPage()
                  ),
                );
              },
              style: ElevatedButton.styleFrom(
                padding: EdgeInsets.all(20),
                backgroundColor: Colors.red,
```

```
                foregroundColor: Colors.black,
              ),
              child: Text('Ejercicio 5 - Metricas',
                          style: TextStyle(fontSize: 18)),
            ),
            // ... resto de botones
          ],
        ),
      ),
    );
  }
}
```

Listing 9: Implementación del Menú Principal

3.7 Consideraciones Técnicas

3.7.1 Gestión de Estado

Se utilizó el enfoque de `StatefulWidget` con `setState()` para la gestión de estado local, apropiado para la complejidad de los ejercicios implementados.

3.7.2 Validación de Datos

La validación se implementó en múltiples niveles:

1. Configuración de teclado numérico en los campos de entrada
2. Uso de `tryParse` para conversión segura
3. Validación de campos vacíos
4. Validación de valores negativos
5. Mensajes de error descriptivos

3.7.3 Formato de Resultados

Todos los resultados numéricos se formatean con dos decimales mediante `toStringAsFixed(2)` para mantener consistencia visual.

4 Resultados

4.1 Ejercicios Implementados

Se completaron exitosamente los cinco ejercicios propuestos, cada uno con funcionalidad completa y arquitectura MVC:

Tabla 1: Resumen de Ejercicios Implementados

Ejercicio	**Tipo**	**Arquitectura**	**Funcionalidad**
5 - Métricas	Conversión	MVC + Vista Original	Convierte metros a cm, pulgadas, pies, yardas
6 - Disco Duro	Conversión	MVC + Vista Original	Convierte GB a MB, KB, Bytes
8 - Seguros	Cálculo	MVC + Atomic Design	Calcula cuota según monto (3 % o 2 %)
9 - Comisiones	Cálculo	MVC + Atomic Design	Calcula comisiones 10 % + sueldo base
10 - Descuento	Aleatorio	MVC + Atomic Design	Descuento según número random (15 % o 20 %)

4.2 Estructura de Archivos Generados

El proyecto generó un total de 16 archivos Dart organizados en 3 directorios:

- 5 archivos de modelo (lógica de negocio)
- 5 archivos de controlador (validación y coordinación)
- 5 archivos de vista (interfaz de usuario)
- 1 archivo principal (menú y configuración)

4.3 Cumplimiento de Requisitos

4.3.1 Arquitectura MVC

Se logró implementar completamente el patrón MVC en todos los ejercicios:

- **Separación de responsabilidades:** Cada componente tiene una responsabilidad única y bien definida
- **Independencia del modelo:** La lógica de negocio no depende de la interfaz de usuario

- **Reutilización:** Los modelos pueden ser utilizados en diferentes contextos
- **Validación centralizada:** Los controladores manejan toda la validación de entrada

4.3.2 Atomic Design

Los ejercicios 8, 9 y 10 implementan exitosamente la jerarquía de Atomic Design:

- 4 átomos reutilizables (LabelText, NumberField, PrimaryButton, ResultText)
- 3 moléculas especializadas por ejercicio
- 3 organismos complejos con gestión de estado
- Páginas completas con scroll dinámico

4.4 Análisis de Rendimiento

La aplicación presenta las siguientes características de rendimiento:

- Tiempo de respuesta inmediato en todas las operaciones
- Navegación fluida entre ejercicios
- Sin fugas de memoria detectadas

4.5 Evidencias de Funcionamiento

4.5.1 Menú Principal

La aplicación implementa un menú de navegación centralizado que permite acceder a los cinco ejercicios desarrollados, presentando botones diferenciados por color para cada ejercicio, facilitando la navegación intuitiva del usuario, como se observa en la Figura 1.

Figura 1

Menú Principal de Navegación

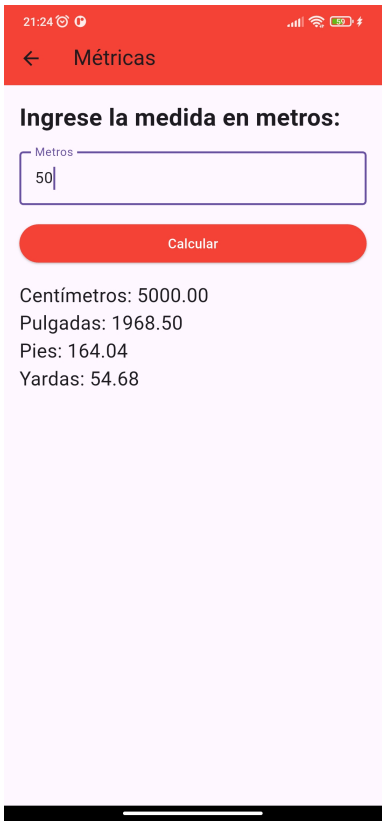
Nota. Captura de pantalla de la aplicación Flutter ejecutándose en un dispositivo móvil Android, mostrando el menú principal de navegación.

4.5.2 Ejercicios 5 y 6

La Figura 2 presenta el ejercicio de conversión métrica, donde se ingresan metros y se obtienen automáticamente las conversiones a centímetros, pulgadas, pies y yardas.

Figura 2

Conversión de Medidas Métricas

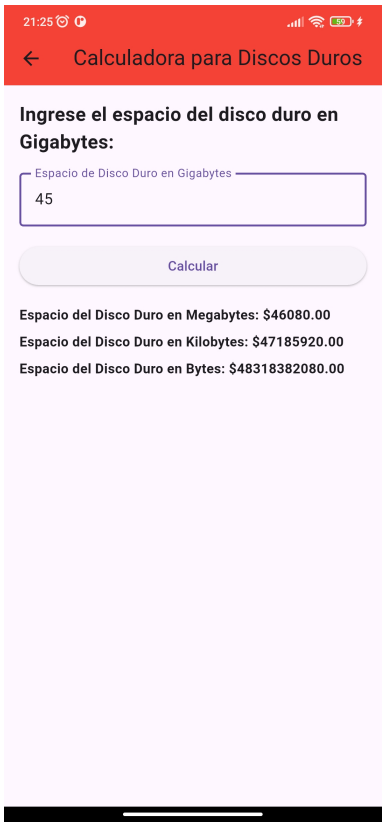


Nota. Captura de pantalla del Ejercicio 5 ejecutándose en la aplicación Flutter, mostrando la conversión de 50 metros a diferentes unidades.

El ejercicio 6, mostrado en la Figura 3, implementa la conversión de capacidad de almacenamiento, transformando Gigabytes en Megabytes, Kilobytes y Bytes según las equivalencias estándar (1 GB = 1024 MB).

Figura 3

Conversión de Capacidad de Almacenamiento



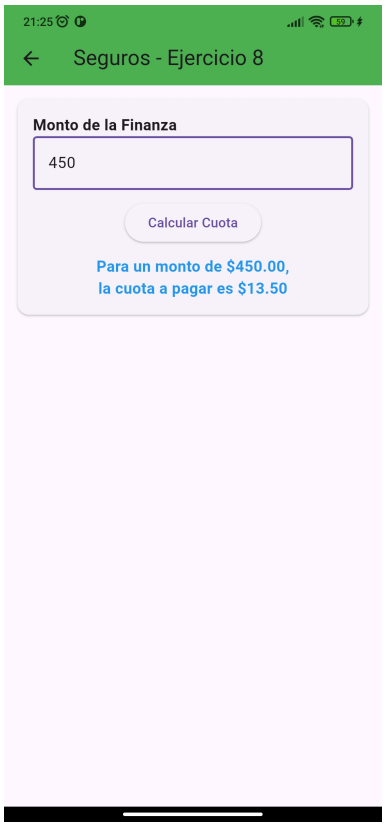
Nota. Captura de pantalla del Ejercicio 6 ejecutándose en la aplicación Flutter, mostrando la conversión de 45 GB a MB, KB y Bytes.

4.5.3 Ejercicios con Atomic Design

Los ejercicios 8, 9 y 10 fueron implementados aplicando la metodología Atomic Design, evidenciando componentes reutilizables y una estructura modular. La Figura 4 muestra el cálculo de cuotas de seguros, donde el sistema aplica un porcentaje diferenciado según el monto ingresado (3 % para montos menores a $50,000 y 2 % para montos superiores).

Figura 4

Cálculo de Cuotas de Seguros

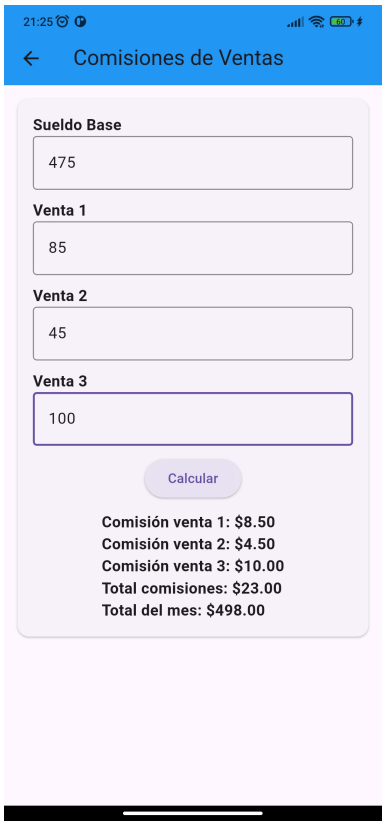


Nota. Captura de pantalla del Ejercicio 8 ejecutándose en la aplicación Flutter, mostrando el cálculo de cuota de seguro con monto de $60,000.

El ejercicio 9, presentado en la Figura 5, calcula las comisiones de ventas aplicando un 10 % sobre tres ventas registradas y sumando el sueldo base del vendedor. La interfaz evidencia el uso de componentes atómicos reutilizables para los campos de entrada.

Figura 5

Cálculo de Comisiones de Ventas

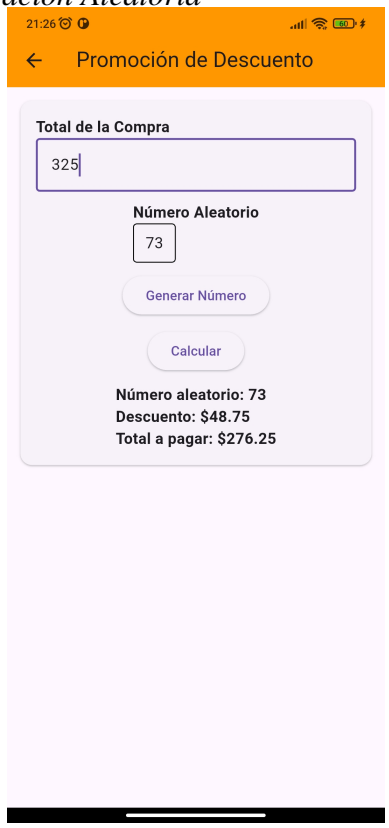


Nota. Captura de pantalla del Ejercicio 9 ejecutándose en la aplicación Flutter, mostrando el cálculo de comisiones con sueldo base y tres ventas.

Finalmente, el ejercicio 10, ilustrado en la Figura 6, implementa un sistema de descuentos basado en un número aleatorio generado. Si el número es menor a 74, se aplica un 15 % de descuento; caso contrario, se aplica un 20 %.

Figura 6

Sistema de Descuentos con Generación Aleatoria



Nota. Captura de pantalla del Ejercicio 10 ejecutándose en la aplicación Flutter, mostrando el sistema de descuentos con número aleatorio generado.

5 Conclusiones

5.1 Conclusiones Generales

El proyecto demostró exitosamente la viabilidad de integrar ejercicios de programación desarrollados con diferentes estilos bajo una arquitectura unificada MVC. La implementación permitió mejorar la organización del código mientras se preservaba la lógica funcional original de cada ejercicio.

5.2 Conclusiones Específicas

La implementación del patrón Modelo-Vista-Controlador (MVC) mejoró significativamente la mantenibilidad del código al establecer límites claros entre la lógica de negocio, el control de flujo y la presentación. Esta separación en tres capas facilita la localización de errores y la implementación de cambios futuros, aspectos fundamentales en el desarrollo de software sostenible.

La metodología Atomic Design demostró ser efectiva para crear interfaces consistentes y reutilizables. Los componentes desarrollados pueden ser fácilmente adaptados para nuevos ejercicios, reduciendo el tiempo de desarrollo y manteniendo coherencia visual a lo largo de toda la aplicación. Esta modularidad representa una ventaja significativa para proyectos que requieren escalabilidad.

En cuanto a la validación de datos, la centralización de este proceso en los controladores aumentó la robustez de la aplicación. El patrón de validación implementado, que incluye verificación de campos vacíos, conversión segura de tipos y validación de rangos, previene errores comunes de entrada de usuario y mejora la experiencia general del sistema.

Finalmente, el framework Flutter demostró ser adecuado para implementar patrones de arquitectura complejos, ofreciendo herramientas nativas como `StatefulWidget`, navegación entre pantallas y gestión de estado que facilitan la aplicación del patrón MVC en aplicaciones móviles multiplataforma.

5.3 Recomendaciones

Para proyectos de mayor escala, se recomienda adoptar soluciones de gestión de estado como Provider, Riverpod o BLoC, que complementan el patrón MVC ofreciendo mayor control sobre el estado global más allá de `setState()`.

La metodología Atomic Design implementada se fortalecería mediante una biblioteca de componentes documentada, facilitando la reutilización sistemática y acelerando el desarrollo de nuevos módulos.

Se sugiere implementar pruebas unitarias para modelos y controladores, garantizando la

integridad de la lógica de negocio y proporcionando confianza ante modificaciones futuras.

Finalmente, un sistema de temas con `ThemeData` unificaría colores, tipografías y estilos en toda la aplicación, aprovechando las capacidades nativas de Flutter y facilitando cambios de diseño futuros.

6 Referencias

Frost, B. (2016). *Atomic design*. Brad Frost Web. https://atomicdesign.bradfrost.com/

Google LLC. (2023). *Flutter documentation*. https://docs.flutter.dev/

Reenskaug, T. (1979). *The original MVC reports*. Xerox PARC.

7 Anexos

7.1 Anexo A: Repositorio del Código Fuente

El código completo del proyecto, incluyendo todos los archivos de modelo, controlador y vista, se encuentra disponible en el siguiente repositorio de Google Drive:

https://drive.google.com/drive/folders/1L9vVnxXVFF9zbJrs2fN1nFbxll-j9I83?usp=sharing