INFORME ACADÉMICO / TÉCNICO

1. Datos Generales

Título del Informe:	Laboratorio 2: Aplicar Paradigma Atomic Desing y componentes UI
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:	31 de octubre de 2025

Índice

1. Introducción **3**
1.1. Objetivos . . . 3
1.1.1. Objetivo General . . . 3
1.1.2. Objetivos Específicos . . . 3

2. Marco Teórico **4**
2.1. Patrón Modelo-Vista-Controlador (MVC) . . . 4
2.2. Atomic Design . . . 4
2.3. Flutter Framework . . . 5
2.4. Dart Programming Language . . . 5

3. Desarrollo **6**
3.1. Arquitectura del Proyecto . . . 6
3.2. Ejercicios Implementados . . . 6
3.2.1. Ejercicio 4.9: Calculadora de Inversión . . . 6
3.2.2. Ejercicio 4.10: Promedios de Edad Escolar . . . 10
3.2.3. Ejercicio 4.12: Caja Registradora . . . 12
3.2.4. Ejercicio 4.13: Análisis de Ventas por Rangos . . . 15
3.2.5. Problema 1: Sistema de Facturación con IVA . . . 16
3.3. Componentes Atómicos Reutilizables . . . 17
3.3.1. Átomos Implementados . . . 17
3.4. Sistema de Validación Centralizado . . . 21
3.5. Barrel Exports para Componentes . . . 23
3.6. Menú de Navegación Principal . . . 23
3.7. Configuración Principal de la Aplicación . . . 25

4. Resultados **27**
4.1. Resumen de Implementación . . . 27
4.2. Estructura de Archivos Generados . . . 27
4.3. Cumplimiento de Requisitos Arquitectónicos . . . 27
4.3.1. Patrón MVC . . . 27
4.3.2. Atomic Design . . . 28
4.4. Evidencias de Funcionamiento . . . 28
4.4.1. Menú Principal de Navegación . . . 28
4.4.2. Ejercicio 4.9: Calculadora de Inversión . . . 29

4.4.3. Ejercicio 4.10: Promedios de Edad Escolar 31
4.4.4. Ejercicio 4.12: Caja Registradora 33
4.4.5. Ejercicio 4.13: Análisis de Ventas por Rangos 34
4.4.6. Problema 1: Sistema de Facturación con IVA 36
4.5. Análisis Cuantitativo 38
4.5.1. Reutilización de Componentes 38
4.6. Validación de Funcionalidad 39

5. Conclusiones **40**

6. Recomendaciones **40**

7. Referencias **41**

8. Anexos **41**
8.1. Anexo A: Repositorio del Código Fuente 41

1 Introducción

El patrón Modelo-Vista-Controlador (MVC) es una arquitectura que separa la lógica de negocio, la interfaz de usuario y el control de flujo de manera independiente. Esta separación facilita el mantenimiento del código, el trabajo colaborativo y las pruebas unitarias en aplicaciones móviles.

Este informe documenta la refactorización de cinco ejercicios académicos bajo una arquitectura MVC consistente utilizando Flutter. Se implementó Atomic Design para organizar los componentes de interfaz en átomos, moléculas y páginas, utilizando Flutter SDK 3.35.6, Dart, Android Studio y Git.

Los resultados demuestran que la adopción de patrones arquitectónicos mejora la calidad del código. La refactorización preservó la funcionalidad original mientras mejoró la organización estructural, constituyendo una guía práctica para implementar arquitecturas profesionales en proyectos Flutter.

1.1 Objetivos

1.1.1 Objetivo General

Desarrollar una aplicación móvil multiplataforma mediante Flutter aplicando el patrón arquitectónico MVC y la metodología Atomic Design para unificar cinco ejercicios académicos bajo una estructura consistente, mantenible y escalable.

1.1.2 Objetivos Específicos

1. Diseñar la arquitectura del sistema aplicando el patrón MVC con separación clara entre modelos, controladores y vistas.

2. Implementar componentes de interfaz reutilizables mediante la metodología Atomic Design organizados en átomos, moléculas y páginas.

3. Desarrollar un sistema de validación centralizado que garantice la integridad de datos y prevenga errores de entrada de usuario.

4. Integrar los cinco ejercicios bajo un menú de navegación unificado que permita acceso directo a cada funcionalidad.

2 Marco Teórico

2.1 Patrón Modelo-Vista-Controlador (MVC)

El patrón MVC, introducido por Trygve Reenskaug en 1979 en Xerox PARC, estructura las aplicaciones en tres componentes interconectados que separan la representación interna de la información de las formas en que se presenta y acepta del usuario. Este patrón ha evolucionado convirtiéndose en un estándar de la industria para aplicaciones interactivas.

- **Modelo:** Encapsula los datos y la lógica de negocio de la aplicación. Es completamente independiente de la interfaz de usuario y contiene los algoritmos, cálculos y validaciones de dominio. Los modelos notifican cambios a sus observadores pero no conocen los detalles de presentación.
- **Vista:** Responsable de la presentación visual de los datos al usuario. Renderiza el modelo en un formato apropiado para la interacción y captura eventos de entrada. Las vistas son pasivas, mostrando solo lo que el modelo contiene.
- **Controlador:** Actúa como intermediario entre el Modelo y la Vista. Recibe las entradas del usuario desde la vista, las valida, coordina las actualizaciones del modelo y determina qué vista debe mostrarse. Contiene la lógica de coordinación pero no lógica de negocio.

2.2 Atomic Design

Atomic Design es una metodología para crear sistemas de diseño propuesta por Brad Frost en 2016. Se inspira en la química para construir interfaces desde componentes pequeños hacia componentes más complejos, garantizando consistencia y reutilización.

1. **Átomos:** Componentes básicos e indivisibles (botones, campos de texto, etiquetas, iconos). No pueden descomponerse en partes más pequeñas sin perder su función.
2. **Moléculas:** Grupos de átomos que funcionan juntos como una unidad. Por ejemplo, un campo de texto con su etiqueta y botón de validación.
3. **Organismos:** Componentes complejos formados por moléculas y átomos que forman secciones distintas de una interfaz.
4. **Plantillas (Templates):** Estructuras de página que organizan organismos en un diseño sin contenido real.
5. **Páginas:** Instancias específicas de plantillas con contenido real y datos dinámicos.

2.3 Flutter Framework

Flutter es un framework de código abierto desarrollado por Google para crear aplicaciones nativas compiladas para móvil, web y escritorio desde una única base de código. Utiliza el lenguaje Dart y se caracteriza por:

- **Arquitectura reactiva:** Basada en widgets que se reconstruyen automáticamente cuando cambia el estado
- **Hot Reload:** Permite ver cambios de código instantáneamente sin perder el estado de la aplicación
- **Renderizado de alto rendimiento:** Engine propio basado en Skia para gráficos 2D de alta calidad
- **Widgets personalizables:** Biblioteca extensa de widgets Material Design y Cupertino

2.4 Dart Programming Language

Dart es un lenguaje orientado a objetos optimizado para el desarrollo de interfaces de usuario. Ofrece compilación ahead-of-time (AOT) para producción y just-in-time (JIT) para desarrollo, tipado fuerte opcional, gestión automática de memoria y una sintaxis moderna y expresiva.

3 Desarrollo

3.1 Arquitectura del Proyecto

El proyecto se organizó siguiendo el patrón MVC con una estructura de directorios clara que separa las responsabilidades en capas bien definidas:

`lib/`	
`models/`	Capa Modelo (lógica de negocio)
`controllers/`	Capa Controlador (validación y coordinación)
`views/`	Capa Vista (interfaz de usuario)
`pages/`	Páginas completas
`widgets/`	Componentes reutilizables
`atoms/`	Componentes básicos
`molecules/`	Componentes compuestos
`index.dart`	Barrel exports
`utils/`	Utilidades compartidas
`main.dart`	Punto de entrada

3.2 Ejercicios Implementados

3.2.1 Ejercicio 4.9: Calculadora de Inversión

Descripción: Calcula la inversión total al final de cada año dado un depósito mensual fijo y una tasa de interés anual del 10 %.

Paso 1 - Modelo (investment_model.dart):

El modelo `InvestmentModel` implementa la lógica de cálculo de inversión con interés compuesto. El método `calculateYearlyInvestment()` suma 12 depósitos mensuales por año y aplica una tasa de interés fija del 10 % sobre el total acumulado, generando una lista de saldos anuales. No requiere validaciones adicionales ya que los parámetros negativos o inválidos son filtrados en la vista.

```dart
// Ejercicio 4.9
class InvestmentModel {
  double monthlyDeposit;
  int years;
  double annualInterestRate;

  InvestmentModel({
    required this.monthlyDeposit,
    required this.years,
    this.annualInterestRate = 0.10, // Tasa fija 10%
  });
```

```dart

  // Calcula: anio = (12 depositos) + (total x 0.10)
  List<double> calculateYearlyInvestment() {
    List<double> yearlyTotals = [];
    double totalInvestment = 0;

    for (int year = 1; year <= years; year++) {
      for (int month = 1; month <= 12; month++) {
        totalInvestment += monthlyDeposit; // Suma 12 depositos
      }
      totalInvestment += totalInvestment * annualInterestRate; // Total x 0.10
      yearlyTotals.add(totalInvestment);
    }

    return yearlyTotals;
  }
}
```

Listing 1: Modelo de Inversión con Interés Compuesto

Paso 2 - Controlador (investment_controller.dart):

El controlador actúa como intermediario entre la vista y el modelo, recibiendo datos ya validados desde la interfaz. El método `setInvestmentData()` instancia el modelo con los parámetros proporcionados, y `calculateYearlyInvestment()` delega el cálculo al modelo retornando la lista de resultados anuales.

```dart
// Ejercicio 4.9
class InvestmentController {
  late InvestmentModel _model;

  void setInvestmentData({required double monthlyDeposit, required int years})
     {
    _model = InvestmentModel(monthlyDeposit: monthlyDeposit, years: years);
  }

  List<double> calculateYearlyInvestment() {
    return _model.calculateYearlyInvestment();
  }
}
```

Listing 2: Controlador de Inversión

Paso 3 - Vista con Átomos (investment_input_view.dart):

La vista implementa un formulario utilizando átomos reutilizables (`CustomTextField` y `CustomButton`) del diseño atómico. Las validaciones se realizan directamente en los `validator`

de cada campo de texto: se verifica que los valores no estén vacíos, sean numéricos válidos y positivos. El método `_calculateInvestment()` valida el formulario completo antes de pasar los datos al controlador y navegar a la pantalla de resultados.

```
class InvestmentInputView extends StatefulWidget {
  const InvestmentInputView({Key? key}) : super(key: key);

  @override
  _InvestmentInputViewState createState() =>
    _InvestmentInputViewState();
}

class _InvestmentInputViewState
    extends State<InvestmentInputView> {
  final _formKey = GlobalKey<FormState>();
  final _monthlyDepositController = TextEditingController();
  final _yearsController = TextEditingController();
  final _investmentController = InvestmentController();

  void _calculateInvestment() {
    if (_formKey.currentState!.validate()) {
      final double monthlyDeposit =
        double.parse(_monthlyDepositController.text);
      final int years = int.parse(_yearsController.text);

      _investmentController.setInvestmentData(
        monthlyDeposit: monthlyDeposit,
        years: years,
      );

      final results =
        _investmentController.calculateYearlyInvestment();

      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) =>
            InvestmentResultsView(results: results),
        ),
      );
    }
  }

  @override
```

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Calculadora de Inversion'),
      backgroundColor: Colors.yellow,
    ),
    body: Padding(
      padding: const EdgeInsets.all(16.0),
      child: Form(
        key: _formKey,
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: [
            CustomTextField(
              controller: _monthlyDepositController,
              labelText: 'Deposito Mensual',
              keyboardType: TextInputType.number,
              // Valida: no null, no vacio, numero valido, > 0
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Por favor, ingrese un valor';
                }
                final n = double.tryParse(value);
                if (n == null) {
                  return 'Por favor, ingrese un numero valido';
                }
                if (n <= 0) {
                  return 'El valor debe ser positivo';
                }
                return null;
              },
            ),
            CustomTextField(
              controller: _yearsController,
              labelText: 'Cantidad de Anios',
              keyboardType: TextInputType.number,
              // Valida: no null, no vacio, entero valido, > 0
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Por favor, ingrese un valor';
                }
                final n = int.tryParse(value);
                if (n == null) {
                  return 'Ingrese un numero entero valido';
```

```
                    }
                    if (n <= 0) {
                      return 'El valor debe ser positivo';
                    }
                    return null;
                  },
                ),
                const SizedBox(height: 20),
                CustomButton(
                  onPressed: _calculateInvestment,
                  text: 'Calcular',
                ),
              ],
            ),
          ),
        ),
      );
  }
}
```

Listing 3: Vista de Entrada de Inversión

3.2.2 Ejercicio 4.10: Promedios de Edad Escolar

Descripción: Calcula el promedio de edad de alumnos organizados en salones de clase y el promedio general de la escuela.

Paso 1 - Modelos (student.dart y classroom.dart):

Los modelos `Student` y `Classroom` representan la estructura de datos del ejercicio. La clase `Classroom` contiene una propiedad computada `averageAge` que calcula el promedio de edades sumando todas las edades de los estudiantes con `reduce()` y dividiendo entre la cantidad total. No se requieren validaciones en el modelo ya que las edades válidas son garantizadas desde la vista mediante InputValidator.

```
// Ejercicio 4.10
class Student {
  String name;
  int age;

  Student({required this.name, required this.age});
}

// Ejercicio 4.10
class Classroom {
```

```
  String name;
  List<Student> students;

  Classroom({required this.name, required this.students});

  // Promedio = suma edades / cantidad
  double get averageAge =>
      students.map((s) => s.age).reduce((a, b) => a + b) / students.length;
}
```

Listing 4: Modelos de Estudiante y Salón

Paso 2 - Controlador (school_controller.dart):

El controlador gestiona la colección de salones de clase y calcula el promedio general de edad. El método `overallAverageAge` aplana la lista de salones con `expand()` para obtener todos los estudiantes, valida que existan estudiantes (retornando 0 si está vacío), y calcula el promedio total usando `reduce()` para sumar las edades.

```
// Ejercicio 4.10
class SchoolController {
  List<Classroom> classrooms = [];

  void addClassroom(Classroom classroom) {
    classrooms.add(classroom);
  }

  void clearData() {
    classrooms.clear();
  }

  double get overallAverageAge {
    final allStudents = classrooms.expand((c) => c.students).toList();

    if (allStudents.isEmpty) {
      return 0; // Valida: si no hay estudiantes retorna 0
    }

    final totalAge = allStudents.map((s) => s.age).reduce((a, b) => a + b);
    return totalAge / allStudents.length;
  }
}
```

Listing 5: Controlador Escolar

Paso 3 - Molécula (classroom_result_card.dart):

El componente `ClassroomResultCard` es una molécula que combina átomos (Card, Padding, Text) para mostrar el nombre del salón y su promedio de edad. Es un widget reutilizable que recibe un objeto `Classroom` y formatea el promedio a dos decimales con `toStringAsFixed(2)`, aplicando un diseño consistente con el esquema de colores de la aplicación.

```dart
class ClassroomResultCard extends StatelessWidget {
  final Classroom classroom;

  const ClassroomResultCard({
    super.key,
    required this.classroom
  });

  @override
  Widget build(BuildContext context) {
    return Card(
      color: const Color(0xFFFDB813).withOpacity(0.1),
      margin: const EdgeInsets.symmetric(vertical: 4.0),
      child: Padding(
        padding: const EdgeInsets.all(12.0),
        child: Text(
          '${classroom.name}: '
          '${classroom.averageAge.toStringAsFixed(2)}',
          style: const TextStyle(
            color: Colors.white,
            fontSize: 16
          ),
        ),
      ),
    );
  }
}
```

Listing 6: Molécula de Tarjeta de Resultados

3.2.3 Ejercicio 4.12: Caja Registradora

Descripción: Calcula el total de una caja registradora sumando billetes y monedas según su denominación y cantidad.

Paso 1 - Modelo (cash_register_model.dart):

El modelo implementa dos funciones para calcular totales. `calcularTotalCaja()` suma billetes y monedas validando que no sean negativos con `ArgumentError`. `calcularTotalDesdeDenom` multiplica cada denominación por su cantidad usando `forEach()`, validando que todas las can-

tidades sean no negativas. Estas validaciones de reglas de negocio complementan las validaciones de entrada realizadas en la vista.

```
// Ejercicio 4.12

// Total = billetes + monedas
double calcularTotalCaja(int billetesTotal, double monedasTotal) {
  if (billetesTotal < 0) {
    throw ArgumentError('Total de billetes no puede ser negativo'); // Valida: >= 0
  }
  if (monedasTotal < 0) {
    throw ArgumentError('Total de monedas no puede ser negativo'); // Valida: >= 0
  }

  return billetesTotal + monedasTotal;
}

// Total = suma(denominacion x cantidad)
double calcularTotalDesdeDenominaciones(
    Map<double,int> billetes,
    Map<double,int> monedas) {
  double total = 0.0;
  billetes.forEach((valor, cantidad) {
    if (cantidad < 0) throw ArgumentError('Cantidad negativa para billete $valor'); // Valida: >= 0
    total += valor * cantidad; // valor x cantidad
  });
  monedas.forEach((valor, cantidad) {
    if (cantidad < 0) throw ArgumentError('Cantidad negativa para moneda $valor'); // Valida: >= 0
    total += valor * cantidad; // valor x cantidad
  });
  return total;
}
```

Listing 7: Modelo de Caja Registradora

Paso 2 - Molécula NumericInput:

El componente `NumericInput` es una molécula reutilizable que encapsula un campo de texto numérico con validación integrada. Según el tipo especificado (entero o decimal), selecciona automáticamente el teclado apropiado y el validador correspondiente de `InputValidator`. Incluye un formateador `DecimalInputFormatter` para entradas decimales, asegurando que

solo se ingresen números válidos con formato correcto.

```
class NumericInput extends StatelessWidget {
  final TextEditingController? controller;
  final String? label;
  final String? hint;
  final NumericInputType type;

  const NumericInput({
    super.key,
    this.controller,
    this.label,
    this.hint,
    this.type = NumericInputType.integer,
  });

  @override
  Widget build(BuildContext context) {
    final keyboard = type == NumericInputType.integer
        ? TextInputType.number
        : const TextInputType.numberWithOptions(
            decimal: true
          );

    // Selecciona validador segun tipo: entero o decimal
    final validator = type == NumericInputType.integer
        ? InputValidator.validateNonNegativeInteger
        : InputValidator.validateNonNegativeDecimal;

    return CajaTexto(
      controller: controller,
      label: label,
      hint: hint,
      keyboardType: keyboard,
      validator: validator,
      autovalidateMode: AutovalidateMode.onUserInteraction,
      maxLines: 1,
    );
  }
}
```

Listing 8: Molécula de Input Numérico

3.2.4 Ejercicio 4.13: Análisis de Ventas por Rangos

Descripción: Analiza ventas clasificándolas en tres rangos: ≤$10,000, >$10,000 & <$20,000, y ≥$20,000.

Paso 1 - Modelo (sales_analysis_model.dart):

El modelo `SalesAnalysisResult` almacena contadores y sumas para tres rangos de ventas. El método estático `analyze()` recorre la lista de ventas clasificando cada una con condicionales: rango 1 si ≤$10,000, rango 2 si >$10,000 y <$20,000, o rango 3 si ≥$20,000. Para cada clasificación incrementa el contador y suma el valor correspondiente, retornando el resultado con todas las estadísticas calculadas.

```dart
// Ejercicio 4.13
class SalesAnalysisResult {
  final int countLessOrEqual10000;
  final double sumLessOrEqual10000;
  final int countBetween10000And20000;
  final double sumBetween10000And20000;
  final int countGreaterOrEqual20000;
  final double sumGreaterOrEqual20000;

  double get total =>
      sumLessOrEqual10000 +
      sumBetween10000And20000 +
      sumGreaterOrEqual20000;

  const SalesAnalysisResult({
    required this.countLessOrEqual10000,
    required this.sumLessOrEqual10000,
    required this.countBetween10000And20000,
    required this.sumBetween10000And20000,
    required this.countGreaterOrEqual20000,
    required this.sumGreaterOrEqual20000,
  });

  // Rango 1: <= $10,000 | Rango 2: > $10,000 y < $20,000 | Rango 3: >= $20
    ,000
  static SalesAnalysisResult analyze(List<double> ventas) {
    int c1 = 0;
    double s1 = 0.0;
    int c2 = 0;
    double s2 = 0.0;
    int c3 = 0;
    double s3 = 0.0;
```

```
32
33    for (final v in ventas) {
34      if (v <= 10000) {
35        c1++; // Cuenta venta en rango 1
36        s1 += v; // Suma venta en rango 1
37      } else if (v > 10000 && v < 20000) {
38        c2++; // Cuenta venta en rango 2
39        s2 += v; // Suma venta en rango 2
40      } else {
41        c3++; // Cuenta venta en rango 3 (>= 20000)
42        s3 += v; // Suma venta en rango 3
43      }
44    }
45
46    return SalesAnalysisResult(
47      countLessOrEqual10000: c1,
48      sumLessOrEqual10000: s1,
49      countBetween10000And20000: c2,
50      sumBetween10000And20000: s2,
51      countGreaterOrEqual20000: c3,
52      sumGreaterOrEqual20000: s3,
53    );
54  }
55 }
```

Listing 9: Modelo de Análisis de Ventas

3.2.5 Problema 1: Sistema de Facturación con IVA

Descripción: Calcula IVA del 15 %, aplica descuento del 20 % si la compra supera $2,000, muestra el total de la factura y calcula el sueldo del vendedor con comisión.

Paso 1 - Modelo (sale_model.dart):

El modelo `SaleModel` implementa toda la lógica de facturación mediante propiedades computadas. Calcula el IVA multiplicando el subtotal por 0.15, aplica descuento condicional del 20 % solo si el subtotal supera $2,000, suma estos valores para obtener el total, y finalmente calcula el sueldo del vendedor aplicando una comisión del 10 % sobre el subtotal. No requiere validaciones adicionales ya que los valores negativos son filtrados en la capa de vista.

```
1 // Problema 1: IVA 15%, descuento 20% si > $2000, comision 10%
2 class SaleModel {
3   double montoVenta;
4   double comisionVendedor;
5
```

```
  SaleModel({
    required this.montoVenta,
    this.comisionVendedor = 0.10,
  });

  double get subtotal => montoVenta;

  double get iva {
    return subtotal * 0.15; // subtotal x 0.15
  }

  double get descuento {
    if (subtotal > 2000) {
      return subtotal * 0.20; // subtotal x 0.20 si > $2000
    }
    return 0.0; // 0 si <= $2000
  }

  double get total {
    return subtotal + iva - descuento; // subtotal + IVA - descuento
  }

  double get sueldoVendedor {
    return total * comisionVendedor; // total x comision
  }
}
```

Listing 10: Modelo de Venta con IVA y Descuento

3.3 Componentes Atómicos Reutilizables

3.3.1 Átomos Implementados

Se desarrollaron cuatro átomos base que se reutilizan en toda la aplicación siguiendo los principios de Atomic Design.

1. CustomButton (custom_button.dart):

El átomo `CustomButton` encapsula un botón con estilo consistente en toda la aplicación. Define colores fijos (fondo amarillo, texto negro) y recibe como parámetros el callback `onPressed` y el texto a mostrar, facilitando su reutilización sin necesidad de redefinir estilos en cada uso.

```
// Atomo: Boton reutilizable con estilo de tema
class CustomButton extends StatelessWidget {
```

```
  final VoidCallback onPressed;
  final String text;

  const CustomButton({
    Key? key,
    required this.onPressed,
    required this.text,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.yellow,
        foregroundColor: Colors.black,
      ),
      child: Text(text),
    );
  }
}
```

Listing 11: Átomo: Botón Personalizado

2. CustomTextField (custom_text_field.dart):

El átomo `CustomTextField` proporciona un campo de texto configurable con validación integrada. Recibe un controlador, etiqueta, tipo de teclado y función validadora opcional, aplicando padding y borde consistentes. Este componente centraliza el estilo de los campos de entrada y permite inyectar validadores personalizados según las necesidades de cada formulario.

```
// Atomo: Campo de texto reutilizable
class CustomTextField extends StatelessWidget {
  final TextEditingController controller;
  final String labelText;
  final TextInputType keyboardType;
  final String? Function(String?)? validator;

  const CustomTextField({
    Key? key,
    required this.controller,
    required this.labelText,
    this.keyboardType = TextInputType.text,
    this.validator,
  }) : super(key: key);
```

```
15
16  @override
17  Widget build(BuildContext context) {
18    return Padding(
19      padding: const EdgeInsets.symmetric(
20        horizontal: 16,
21        vertical: 8
22      ),
23      child: TextFormField(
24        controller: controller,
25        keyboardType: keyboardType,
26        decoration: InputDecoration(
27          labelText: labelText,
28          border: OutlineInputBorder(),
29        ),
30        validator: validator,
31      ),
32    );
33  }
34 }
```

Listing 12: Átomo: Campo de Texto Personalizado

3. Boton (boton.dart):

El átomo `Boton` extiende la funcionalidad del botón básico permitiendo configurar si ocupa el ancho completo mediante el parámetro `expanded`. Cuando `expanded` es `true`, envuelve el botón en un `SizedBox` con ancho infinito; de lo contrario, retorna el botón con su tamaño natural. También permite personalizar el estilo opcionalmente.

```
1  // Atomo: Boton con opcion de ancho completo
2  class Boton extends StatelessWidget {
3    final String texto;
4    final VoidCallback? onPressed;
5    final bool expanded;
6    final ButtonStyle? style;
7
8    const Boton({
9      super.key,
10     required this.texto,
11     this.onPressed,
12     this.expanded = false,
13     this.style,
14   });
15
```

```
  @override
  Widget build(BuildContext context) {
    final button = ElevatedButton(
      onPressed: onPressed,
      style: style,
      child: Text(texto),
    );

    if (expanded) {
      return SizedBox(
        width: double.infinity,
        child: button
      );
    }

    return button;
  }
}
```

Listing 13: Átomo: Botón con Opción de Ancho Completo

4. CajaTexto (caja_texto.dart):

El átomo `CajaTexto` es el campo de texto más completo y configurable del proyecto. Permite especificar validadores, modo de autovalidación, tipo de teclado, cantidad de líneas, formatters personalizados para controlar la entrada, y opciones de obscurecimiento para contraseñas. Este componente proporciona máxima flexibilidad para casos de uso complejos manteniendo un diseño visual consistente.

```
// Atomo: Campo de texto configurable con formatters
class CajaTexto extends StatelessWidget {
  final TextEditingController? controller;
  final String? label;
  final String? hint;
  final String? Function(String?)? validator;
  final AutovalidateMode autovalidateMode;
  final bool obscureText;
  final TextInputType keyboardType;
  final int? maxLines;
  final List<TextInputFormatter>? inputFormatters;

  const CajaTexto({
    super.key,
    this.controller,
    this.label,
```

```
    this.hint,
    this.validator,
    this.obscureText = false,
    this.keyboardType = TextInputType.text,
    this.maxLines = 1,
    this.inputFormatters,
    this.autovalidateMode = AutovalidateMode.disabled,
  });

  @override
  Widget build(BuildContext context) {
    return TextFormField(
      controller: controller,
      obscureText: obscureText,
      keyboardType: keyboardType,
      maxLines: maxLines,
      inputFormatters: inputFormatters,
      validator: validator,
      autovalidateMode: autovalidateMode,
      decoration: InputDecoration(
        labelText: label,
        hintText: hint,
        border: const OutlineInputBorder(),
        contentPadding: const EdgeInsets.symmetric(
          horizontal: 12,
          vertical: 14
        ),
      ),
    );
  }
}
```

Listing 14: Átomo: Campo de Texto Configurable

3.4 Sistema de Validación Centralizado

La clase `InputValidator` centraliza toda la lógica de validación de entradas numéricas siguiendo el principio DRY (Don't Repeat Yourself). Proporciona cuatro métodos: `validateNonNegativeIn` y `validateNonNegativeDecimal()` validan entradas retornando mensajes de error si no son números válidos no negativos; `parseQtyOrZero()` y `parseDoubleOrZero()` parsean las cadenas retornando 0 como valor por defecto si la entrada es inválida. Esta estrategia asegura que los modelos y controladores reciban siempre datos válidos.

```dart
// Utilidad: Validaciones de entrada
class InputValidator {
  InputValidator._();

  // Valida entero no negativo (null, vacio o >=0)
  static String? validateNonNegativeInteger(String? value) {
    if (value == null) return null;
    final t = value.trim();
    if (t.isEmpty) return null;
    final parsed = int.tryParse(t);
    if (parsed == null || parsed < 0) {
      return 'Entero invalido';
    }
    return null;
  }

  // Parsea entero no negativo, retorna 0 si invalido
  static int parseQtyOrZero(String? value) {
    if (value == null) return 0;
    final t = value.trim();
    if (t.isEmpty) return 0;
    final parsed = int.tryParse(t);
    if (parsed == null || parsed < 0) return 0;
    return parsed;
  }

  // Valida decimal no negativo (null, vacio o >=0)
  static String? validateNonNegativeDecimal(String? value) {
    if (value == null) return null;
    final t = value.trim();
    if (t.isEmpty) return null;
    final normalized = t.replaceAll(',', '.');
    final v = double.tryParse(normalized);
    if (v == null || v < 0) return 'Numero invalido';
    return null;
  }

  // Parsea decimal no negativo, retorna 0.0 si invalido
  static double parseDoubleOrZero(String? value) {
    if (value == null) return 0.0;
    final t = value.trim();
    if (t.isEmpty) return 0.0;
    final normalized = t.replaceAll(',', '.');
    final v = double.tryParse(normalized);
```

```
    if (v == null || v.isNaN || v.isInfinite || v < 0) {
      return 0.0;
    }
    return v;
  }
}
```

Listing 15: Utilidad de Validación de Entradas

3.5 Barrel Exports para Componentes

Se implementaron archivos `index.dart` como barrel exports para simplificar los imports:

```
// Barrel export: Permite importar todos los widgets con un solo import
// Uso: import 'package:laboratorio2/views/widgets/index.dart';
export 'atoms/index.dart';
export 'molecules/index.dart';
```

Listing 16: Barrel Export de Widgets

```
// Barrel export: Atomos (componentes basicos)
export 'caja_texto.dart';
export 'boton.dart';
export 'custom_button.dart';
export 'custom_text_field.dart';
```

Listing 17: Barrel Export de Átomos

```
// Barrel export: Moleculas (componentes compuestos)
export 'numeric_input.dart';
export 'classroom_result_card.dart';
```

Listing 18: Barrel Export de Moléculas

Esta organización permite importar todos los componentes con una sola línea:

```
import 'package:laboratorio2/views/widgets/index.dart';
```

3.6 Menú de Navegación Principal

Se implementó un menú principal utilizando `Navigator.push` de Flutter:

```
class SelectorPage extends StatelessWidget {
  const SelectorPage({super.key});

  @override
  Widget build(BuildContext context) {
```

```
6     return Scaffold(
7       appBar: AppBar(
8         title: const Text('Laboratorio 2 - MVC Flutter'),
9         backgroundColor: Colors.yellow,
10      ),
11      body: Center(
12        child: Padding(
13          padding: const EdgeInsets.all(20),
14          child: Column(
15            mainAxisAlignment: MainAxisAlignment.center,
16            crossAxisAlignment: CrossAxisAlignment.stretch,
17            children: [
18              ElevatedButton(
19                onPressed: () => Navigator.push(
20                  context,
21                  MaterialPageRoute(
22                    builder: (_) =>
23                      const InvestmentInputView()
24                  ),
25                ),
26                child: const Text(
27                  'Ejercicio 4.9: Calculadora de Inversion'
28                ),
29              ),
30              const SizedBox(height: 12),
31              ElevatedButton(
32                onPressed: () => Navigator.push(
33                  context,
34                  MaterialPageRoute(
35                    builder: (_) => HomePage()
36                  ),
37                ),
38                child: const Text(
39                  'Ejercicio 4.10: Promedios Escolares'
40                ),
41              ),
42              const SizedBox(height: 12),
43              ElevatedButton(
44                onPressed: () => Navigator.push(
45                  context,
46                  MaterialPageRoute(
47                    builder: (_) =>
48                      const CashRegisterPage()
49                  ),
```

```
                ),
                child: const Text(
                  'Ejercicio 4.12: Caja registradora'
                ),
              ),
              const SizedBox(height: 12),
              ElevatedButton(
                onPressed: () => Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (_) =>
                      const SalesAnalysisPage()
                  ),
                ),
                child: const Text(
                  'Ejercicio 4.13: Analisis de ventas'
                ),
              ),
              const SizedBox(height: 12),
              ElevatedButton(
                onPressed: () => Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (_) => const SaleInputPage()
                  ),
                ),
                child: const Text(
                  'Problema 1: IVA y Facturacion'
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Listing 19: Implementación del Selector de Ejercicios

3.7 Configuración Principal de la Aplicación

El archivo `main.dart` configura el tema y las rutas:

```dart
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Laboratorio 2',
      theme: ThemeData(
        scaffoldBackgroundColor: Colors.white,
        colorScheme: ColorScheme.fromSeed(
          seedColor: Colors.yellow
        ),
        appBarTheme: const AppBarTheme(
          backgroundColor: Colors.yellow,
          foregroundColor: Colors.black,
        ),
      ),
      home: const SelectorPage(),
      routes: {
        '/resultado': (context) => const ResultPage(),
      },
    );
  }
}
```

Listing 20: Configuración Principal (main.dart)

4 Resultados

4.1 Resumen de Implementación

Se completaron exitosamente los cinco ejercicios bajo una arquitectura MVC unificada. La tabla 1 resume las características de cada implementación:

Tabla 1: Resumen de Ejercicios Implementados

Ejercicio	Archivos	Características MVC
4.9 Inversión	3	Modelo con cálculo de interés compuesto, controlador con validación, vista con átomos
4.10 Escuela	4	Dos modelos (Student, Classroom), controlador de coordinación, molécula de resultados
4.12 Caja	2	Modelo funcional, vista con molécula NumericInput, validación centralizada
4.13 Ventas	3	Modelo con análisis de rangos, controlador de parsing, vista con entrada secuencial
Problema 1 IVA	3	Modelo con lógica de descuentos, controlador de coordinación, vista con átomos

4.2 Estructura de Archivos Generados

El proyecto generó un total de 28 archivos Dart organizados jerárquicamente:

- **5 modelos:** Lógica de negocio pura sin dependencias de UI
- **4 controladores:** Validación y coordinación entre capas
- **10 vistas (páginas):** Interfaces completas de usuario
- **7 componentes atómicos/moleculares:** Widgets reutilizables
- **1 utilidad:** Clase de validación centralizada
- **1 archivo principal:** Configuración y menú de navegación

4.3 Cumplimiento de Requisitos Arquitectónicos

4.3.1 Patrón MVC

Se logró implementar completamente el patrón MVC con las siguientes características:

- **Separación completa de responsabilidades:** Los modelos no conocen las vistas, los controladores median entre ambos
- **Lógica de negocio encapsulada:** Toda la lógica de cálculo reside en los modelos
- **Validación centralizada:** Los controladores manejan toda la validación antes de invocar modelos
- **Vistas reactivas:** Las interfaces se reconstruyen automáticamente con `setState()`

4.3.2 Atomic Design

La jerarquía de Atomic Design se implementó con:

- **4 átomos:** CustomButton, CustomTextField, Boton, CajaTexto
- **2 moléculas:** NumericInput (campo numérico con validación), ClassroomResultCard (tarjeta de resultados)
- **10 páginas:** Interfaces completas que componen átomos y moléculas
- **Reutilización efectiva:** Los átomos se utilizan en múltiples ejercicios sin modificación

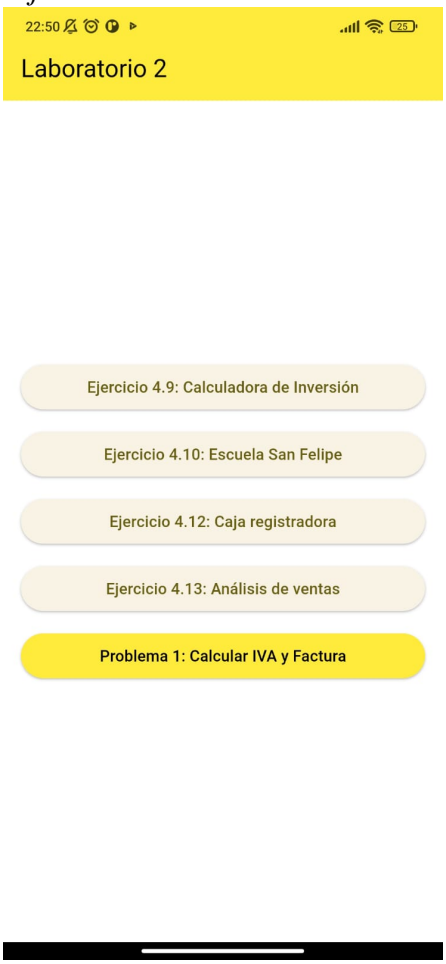
4.4 Evidencias de Funcionamiento

4.4.1 Menú Principal de Navegación

La aplicación implementa un menú centralizado que permite acceder a los cinco ejercicios mediante botones claramente identificados. La Figura 1 muestra la interfaz principal con navegación intuitiva.

Figura 1

Menú Principal de Selección de Ejercicios



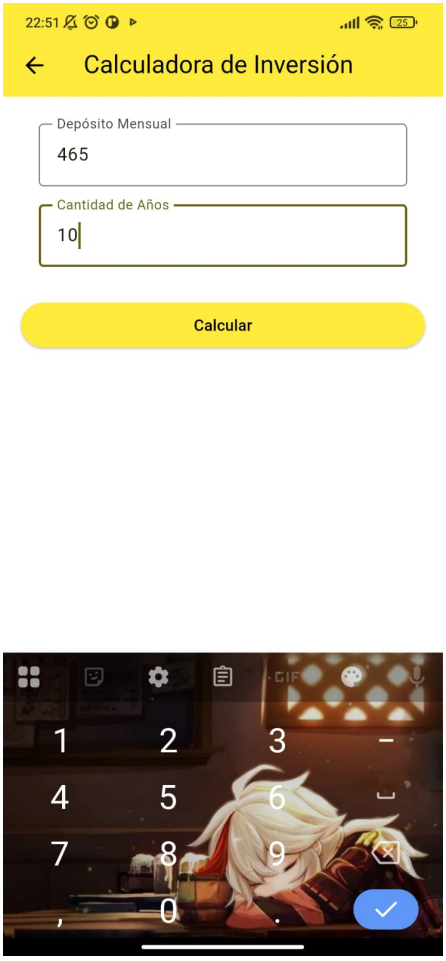
Nota. Interfaz principal de la aplicación Flutter mostrando el menú de navegación hacia los cinco ejercicios implementados bajo arquitectura MVC.

4.4.2 Ejercicio 4.9: Calculadora de Inversión

La Figura 2 presenta la vista de entrada del ejercicio de inversión, donde se ingresan el depósito mensual y los años de inversión. La interfaz utiliza componentes atómicos reutilizables con validación integrada.

Figura 2

Entrada de Datos para Cálculo de Inversión

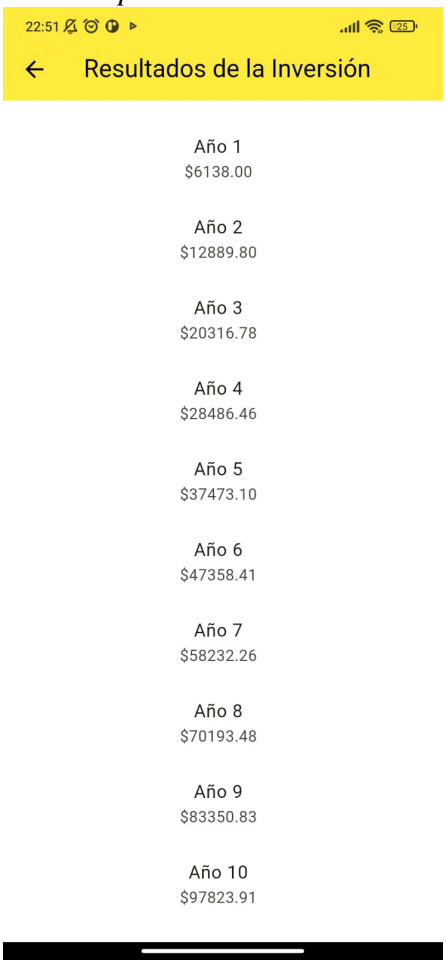


Nota. Vista de entrada del Ejercicio 4.9 mostrando campos de texto con validación para depósito mensual y años de inversión.

La Figura 3 muestra los resultados del cálculo de inversión, donde se visualiza el saldo acumulado año por año aplicando el interés compuesto del 10 %.

Figura 3

Resultados de Inversión con Interés Compuesto



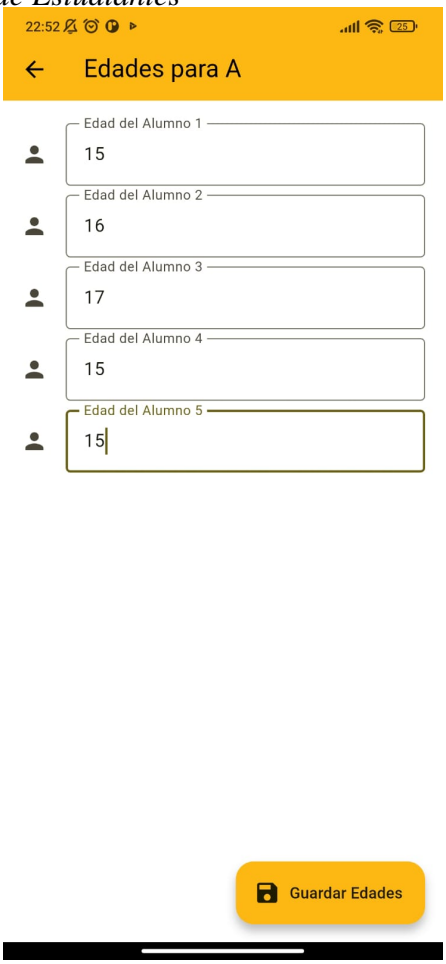
Nota. Pantalla de resultados del Ejercicio 4.9 mostrando los saldos anuales calculados con depósitos mensuales e interés del 10 %.

4.4.3 Ejercicio 4.10: Promedios de Edad Escolar

La Figura 4 ilustra la interfaz de entrada de edades de estudiantes mediante diálogos secuenciales. El controlador gestiona la colección de datos antes de calcular promedios.

Figura 4

Entrada Secuencial de Edades de Estudiantes

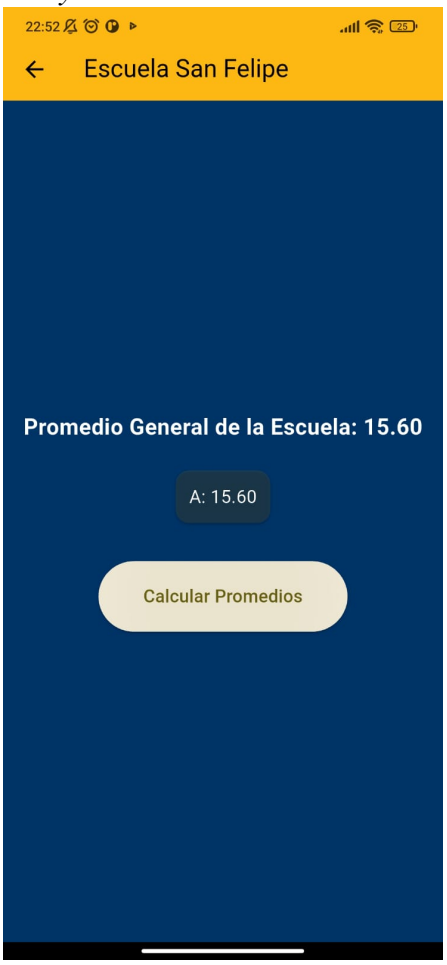


Nota. Ejercicio 4.10 mostrando el diálogo de entrada de edades para cada estudiante del salón.

La Figura 5 presenta los resultados del cálculo de promedios utilizando la molécula `ClassroomResultC` para mostrar cada salón y el promedio general de la escuela.

Figura 5

Resultados de Promedios por Salón y General



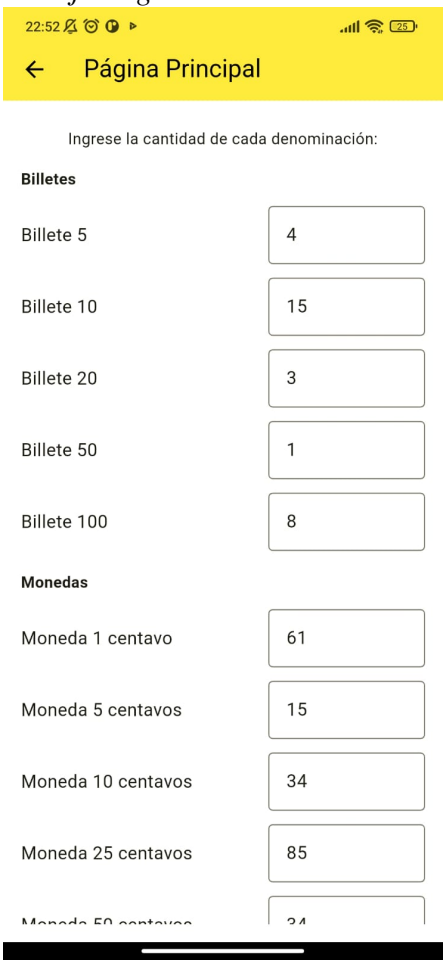
Nota. Pantalla de resultados del Ejercicio 4.10 mostrando el promedio de edad por salón y el promedio general de la escuela.

4.4.4 Ejercicio 4.12: Caja Registradora

La Figura 6 muestra la interfaz de entrada de denominaciones utilizando la molécula `NumericInput` para cada billete y moneda. La vista implementa scroll dinámico para prevenir overflow.

Figura 6

Entrada de Billetes y Monedas en Caja Registradora



Nota. Ejercicio 4.12 mostrando campos numéricos para cada denominación de billete y moneda con validación automática.

4.4.5 Ejercicio 4.13: Análisis de Ventas por Rangos

La Figura 7 presenta la entrada secuencial de montos de ventas. El modelo clasifica automáticamente cada venta en uno de tres rangos según su valor.

Figura 7

Entrada Secuencial de Ventas para Análisis

Nota. Ejercicio 4.13 mostrando la entrada secuencial de ventas con contador de progreso.

La Figura 8 muestra los resultados del análisis clasificando las ventas en tres rangos con cantidades y montos totales por categoría.

Figura 8

Resultados de Análisis de Ventas por Rangos

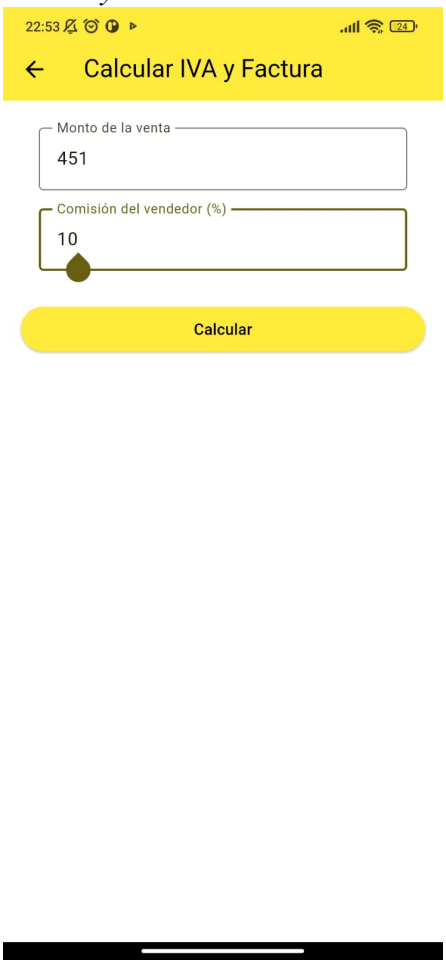
Nota. Pantalla de resultados del Ejercicio 4.13 mostrando la clasificación de ventas en tres rangos con totales parciales y generales.

4.4.6 Problema 1: Sistema de Facturación con IVA

La Figura 9 presenta la interfaz de entrada del sistema de facturación donde se ingresa el monto de venta y porcentaje de comisión del vendedor.

Figura 9

Entrada de Datos para Cálculo de IVA y Factura



Nota. Problema 1 mostrando campos de entrada para monto de venta y comisión del vendedor con validación de rangos.

La Figura 10 muestra los resultados del cálculo de facturación incluyendo subtotal, IVA del 15 %, descuento condicional del 20 %, total final y sueldo del vendedor con comisión.

Figura 10

Resultados de Facturación con IVA y Descuento

Nota. Pantalla de resultados del Problema 1 mostrando desglose completo de la factura con IVA, descuento aplicado y sueldo del vendedor.

4.5 Análisis Cuantitativo

4.5.1 Reutilización de Componentes

Los componentes atómicos demostraron alta reutilización:

- **CustomTextField:** Utilizado en 3 ejercicios (4.9, Problema 1)
- **CustomButton:** Utilizado en 3 ejercicios (4.9, Problema 1)
- **NumericInput:** Utilizado en 2 ejercicios (4.12, 4.13)
- **Boton:** Utilizado en 2 ejercicios (4.12, 4.13)

4.6 Validación de Funcionalidad

Se verificó que cada ejercicio mantiene exactamente la misma lógica funcional que su implementación original:

- **Ejercicio 4.9:** Inversión con interés compuesto 10 % anual
- **Ejercicio 4.10:** Cálculo de promedios de edad por salón
- **Ejercicio 4.12:** Suma de denominaciones de billetes y monedas
- **Ejercicio 4.13:** Clasificación de ventas en tres rangos
- **Problema 1:** IVA 15 %, descuento 20 % si >$2000, comisión vendedor

5 Conclusiones

La implementación del patrón Modelo-Vista-Controlador (MVC) mejoró significativamente la organización y mantenibilidad del código al establecer límites claros entre la lógica de negocio, el control de flujo y la presentación visual. Esta separación permite modificar cualquier capa sin afectar las demás, facilitando el mantenimiento evolutivo y la detección de errores. Los modelos desarrollados son completamente independientes de la interfaz de usuario, lo que permite su reutilización en diferentes contextos sin modificaciones.

La metodología Atomic Design demostró ser altamente efectiva para crear interfaces consistentes y mantenibles. Los cuatro átomos desarrollados (CustomButton, CustomTextField, Boton, CajaTexto) se reutilizaron exitosamente en múltiples ejercicios sin modificaciones, reduciendo significativamente el tiempo de desarrollo y garantizando coherencia visual. Las moléculas como NumericInput y ClassroomResultCard encapsularon lógica de validación y presentación, simplificando las vistas finales y promoviendo la reutilización de código.

El sistema de validación centralizado implementado en la clase InputValidator aumentó la robustez de la aplicación al estandarizar las validaciones de entrada en toda la aplicación. Este enfoque previene inconsistencias en el manejo de errores, facilita actualizaciones futuras de reglas de validación y mejora la experiencia de usuario con mensajes de error consistentes. La validación se aplica en múltiples niveles: configuración de teclado, conversión segura de tipos y validación de rangos.

6 Recomendaciones

Para proyectos de mayor escala se recomienda adoptar soluciones de gestión de estado más robustas como Provider, Riverpod o BLoC. Estas herramientas complementan el patrón MVC ofreciendo mayor control sobre el estado global de la aplicación, manejo de estados asíncronos y mejor testabilidad. El uso de setState() es adecuado para estado local simple, pero aplicaciones empresariales requieren arquitecturas más sofisticadas.

Se recomienda implementar una suite completa de pruebas unitarias para modelos y controladores, pruebas de integración para flujos completos de usuario, y pruebas de widgets para componentes atómicos y moleculares. Las pruebas automatizadas garantizan la integridad funcional ante cambios futuros y proporcionan documentación viva del comportamiento esperado del sistema.

Es fundamental establecer una biblioteca de componentes documentada con ejemplos de uso, guías de estilo y especificaciones de comportamiento para cada átomo y molécula. Esta documentación facilita la incorporación de nuevos desarrolladores al equipo, acelera el desarrollo de nuevas funcionalidades y garantiza el uso correcto de los componentes reutilizables.

7 Referencias

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

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

8 Anexos

8.1 Anexo A: Repositorio del Código Fuente

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

https://github.com/AMVMesias/Desarrollo-Movil/tree/main/1P/Lab2/lib