INFORME ACADÉMICO / TÉCNICO

1. Datos Generales

Título del Informe:	Actividad 2.1 Aplicación Calcular Sueldo de Operario
Autor(a):	Mesias Orlando Mariscal Bryan Roberto Quispe Romero 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:	26 de noviembre de 2025

Índice

1 Introducción

El presente informe documenta el desarrollo de una aplicación móvil para calcular aumentos salariales de operarios, implementando Clean Architecture y Atomic Design. El proyecto automatiza el cálculo de incrementos salariales basados en el salario actual y la antigüedad laboral del trabajador, demostrando la aplicación práctica de patrones arquitectónicos modernos.

Para la solución se utilizó Flutter como framework principal complementado con Provider para la gestión del estado. La arquitectura se estructuró en capas definidas: dominio, presentación y temas, permitiendo separación de responsabilidades y facilidad en el mantenimiento del código a largo plazo.

La implementación resultante proporciona tres pantallas principales: cálculo de aumento, historial de operaciones y generación de reportes PDF. Estas pantallas están interconectadas mediante un sistema centralizado de estado, demostrando las ventajas de los patrones arquitectónicos empleados en aplicaciones móviles funcionales.

1.1 Objetivos

1.1.1 Objetivo General

Desarrollar una aplicación móvil para el cálculo de aumentos salariales de operarios mediante la implementación de Clean Architecture y Atomic Design, utilizando Flutter y Provider como herramientas de desarrollo, para automatizar y registrar los procesos de incremento salarial.

1.1.2 Objetivos Específicos

1. Investigar los fundamentos teóricos de Clean Architecture, Atomic Design y gestión de estado con Provider para establecer las bases conceptuales del desarrollo.

2. Implementar la capa de dominio con entidades y casos de uso que encapsulen la lógica de negocio del cálculo de aumentos salariales.

3. Desarrollar la capa de presentación aplicando Atomic Design con componentes reutilizables organizados en átomos, moléculas y organismos.

4. Crear un sistema de historial que almacene y visualice todos los cálculos realizados durante la sesión de la aplicación.

2 Marco Teórico

2.1 Clean Architecture

Clean Architecture es un enfoque de diseño de software propuesto por Robert C. Martin que organiza el código en capas concéntricas con dependencias que apuntan hacia el interior. La capa más interna contiene las entidades y reglas de negocio, mientras que las capas externas manejan detalles de implementación como interfaces de usuario y bases de datos. Esta separación permite que el núcleo de la aplicación sea independiente de frameworks y herramientas externas, facilitando las pruebas y el mantenimiento del sistema.

2.2 Atomic Design

Atomic Design es una metodología creada por Brad Frost para construir sistemas de diseño de manera jerárquica y modular. Los átomos representan componentes básicos indivisibles como botones y campos de texto. Las moléculas combinan varios átomos para formar componentes funcionales. Los organismos agrupan moléculas y átomos en secciones completas de la interfaz. Esta metodología promueve la reutilización de componentes y mantiene la consistencia visual en toda la aplicación.

2.3 Flutter y Provider

Flutter es un framework de código abierto desarrollado por Google que permite crear aplicaciones multiplataforma desde una única base de código. Utiliza el lenguaje Dart y se caracteriza por su sistema de widgets reactivos. Provider es un paquete de gestión de estado recomendado por el equipo de Flutter que implementa el patrón ChangeNotifier, permitiendo que los widgets escuchen y reaccionen a cambios en el estado de la aplicación de manera eficiente y declarativa.

2.4 Casos de Uso

Los casos de uso representan las acciones específicas que un sistema puede realizar, encapsulando la lógica de negocio de manera independiente. En Clean Architecture, los casos de uso residen en la capa de dominio y orquestan el flujo de datos entre las entidades y las capas externas. Cada caso de uso debe tener una única responsabilidad, siguiendo el principio de responsabilidad única (SRP), lo que facilita su prueba y mantenimiento.

3 Desarrollo

3.1 Arquitectura del Proyecto

La estructura del proyecto sigue una organización basada en Clean Architecture con la siguiente distribución de carpetas principales:

- `lib/src/domain/`: Contiene las entidades y casos de uso
- `lib/src/presentation/`: Incluye vistas, viewmodels, rutas y widgets
- `lib/temas/`: Almacena la configuración de estilos y colores

3.2 Capa de Dominio

3.2.1 Archivo: src/domain/entities/operario.dart

Este archivo define la entidad principal que representa a un operario con sus atributos fundamentales. La clase `Operario` contiene dos propiedades: el sueldo actual como valor decimal y la antigüedad en años como valor entero.

```dart
class Operario {
  final double sueldo;
  final int antiguedad;

  Operario({required this.sueldo, required this.antiguedad});
}
```

Listing 1: Entidad Operario en operario.dart

La entidad utiliza el modificador `final` para garantizar la inmutabilidad de los datos una vez creada la instancia. El constructor con parámetros nombrados y requeridos asegura que siempre se proporcionen ambos valores al crear un operario.

3.2.2 Archivo: src/domain/entities/resultado_operario.dart

Define la estructura que almacena el resultado del cálculo salarial, incluyendo el monto del aumento y el sueldo final calculado.

```dart
class ResultadoOperario {
  final double aumento;
  final double sueldoFinal;

  ResultadoOperario({required this.aumento, required this.sueldoFinal});
}
```

Listing 2: Entidad ResultadoOperario en resultado_operario.dart

Esta entidad encapsula los resultados del proceso de cálculo, separando la lógica de presentación de los datos calculados. La inmutabilidad garantiza que los resultados no puedan ser modificados accidentalmente después de su creación.

3.2.3 Archivo: src/domain/usescases/calcular_operario_usecases.dart

Este archivo contiene el caso de uso principal que implementa la lógica de negocio para el cálculo de aumentos salariales. El método `ejecutar` recibe un objeto Operario y retorna un ResultadoOperario con los valores calculados.

```dart
class CalcularAumentoUseCase {
  ResultadoOperario ejecutar(Operario op){
    double aumento = 0;

    if(op.sueldo < 500){
      if(op.antiguedad >= 10){
        aumento = op.sueldo * 0.20;
      } else if(op.antiguedad >= 5){
        aumento = op.sueldo * 0.15;
      } else if(op.antiguedad >= 2){
        aumento = op.sueldo * 0.10;
      } else {
        aumento = op.sueldo * 0.05;
      }
    }
    final sueldoFinal = op.sueldo + aumento;
    return ResultadoOperario(aumento: aumento, sueldoFinal: sueldoFinal);
  }
}
```

Listing 3: Lógica de cálculo en calcular_operario_usecases.dart

La lógica de cálculo implementa las siguientes reglas de negocio: si el sueldo es menor a 500, se aplica un aumento escalonado según la antigüedad del operario. Para trabajadores con 10 años o más se aplica un 20 %, para aquellos con 5 a 9 años un 15 %, para 2 a 4 años un 10 %, y para menos de 2 años un 5 %. Si el sueldo es igual o mayor a 500, no se aplica ningún aumento. Esta separación de la lógica en un caso de uso independiente permite modificar las reglas sin afectar otras partes del sistema.

La validación en la capa de dominio es fundamental para garantizar la integridad de los datos. El caso de uso recibe los parámetros del operario y aplica las reglas de negocio sin necesidad de validaciones adicionales, confiando en que la capa de presentación ha verificado que los datos sean correctos.

3.3 Capa de Presentación - ViewModel

3.3.1 Archivo: src/presentation/viewmodel/operario_viewmodel.dart

El ViewModel actúa como intermediario entre la vista y el dominio, gestionando el estado de la aplicación mediante Provider. Implementa `ChangeNotifier` para notificar a los widgets cuando el estado cambia.

```dart
class OperarioViewModel extends ChangeNotifier{
  final CalcularAumentoUseCase _useCase;

  OperarioViewModel(this._useCase);

  ResultadoOperario? _resultado;
  ResultadoOperario? get resultado => _resultado;

  final List<ResultadoOperario> _historial = [];
  List<ResultadoOperario> get historial => _historial;

  void calcular(double sueldo, int antiguedad){
    final op = Operario(sueldo: sueldo, antiguedad: antiguedad);
    _resultado = _useCase.ejecutar(op);

    if (_resultado != null) {
      _historial.add(_resultado!);
    }
    notifyListeners();
  }
}
```

Listing 4: Gestión de estado en operario_viewmodel.dart

El ViewModel mantiene dos estados principales: el resultado actual del último cálculo y una lista de historial con todos los cálculos realizados. El método `calcular` crea una instancia de Operario, ejecuta el caso de uso, almacena el resultado en el historial y notifica a los listeners para actualizar la interfaz. Esta implementación permite que múltiples pantallas accedan al mismo historial de cálculos.

3.4 Capa de Presentacion - Widgets Atomic Design

3.4.1 Atomos

Los atomos son componentes basicos ubicados en `src/presentation/widgets/atoms/`. Se implementaron tres atomos principales:

Archivo: custom_text_field.dart - Campo de texto con validacion integrada:

```dart
class CustomTextField extends StatelessWidget {
  final String label;
  final String hint;
  final TextEditingController controller;
```

```
  final TextInputType keyboardType;
  final String? Function(String?)? validator;

  @override
  Widget build(BuildContext context) {
    return TextFormField(
      controller: controller,
      keyboardType: keyboardType,
      validator: validator,
      decoration: InputDecoration(
        labelText: label,
        hintText: hint,
        border: OutlineInputBorder(
          borderRadius: BorderRadius.circular(12),
        ),
        filled: true,
        fillColor: colorScheme.primary.withOpacity(0.05),
      ),
    );
  }
}
```

Listing 5: Campo de texto en custom_text_field.dart

Archivo: custom_button.dart - Boton estilizado reutilizable:

```
class CustomButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;
  final bool isEnabled;

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: double.infinity,
      height: 56,
      child: ElevatedButton(
        onPressed: isEnabled ? onPressed : null,
        style: ElevatedButton.styleFrom(
          backgroundColor: colorScheme.primary,
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(12),
          ),
        ),
        child: Text(text, style: TextStyle(fontSize: 18)),
      ),
    );
  }
}
```

Listing 6: Boton personalizado en custom_button.dart

Archivo: custom_radio_option.dart - Opcion de radio button personalizada:

```
class CustomRadioOption<T> extends StatelessWidget {
  final T value;
```

```
  final T groupValue;
  final String label;
  final ValueChanged<T?> onChanged;

  @override
  Widget build(BuildContext context) {
    final isSelected = value == groupValue;
    return InkWell(
      onTap: () => onChanged(value),
      child: Container(
        decoration: BoxDecoration(
          color: isSelected ? colorScheme.primary.withOpacity(0.1) : Colors.transparent,
          borderRadius: BorderRadius.circular(12),
          border: Border.all(
            color: isSelected ? colorScheme.primary : Colors.grey,
          ),
        ),
        child: Row(
          children: [
            Radio<T>(value: value, groupValue: groupValue, onChanged: onChanged),
            Text(label),
          ],
        ),
      ),
    );
  }
}
```

Listing 7: Radio option en custom_radio_option.dart

3.4.2 Moleculas

Las moléculas combinan átomos para crear componentes funcionales más complejos. El archivo `radio_button_group.dart` agrupa opciones de antigüedad con valores específicos que corresponden a los rangos de cálculo.

```
CustomRadioOption<int>(
  value: 1,
  groupValue: selectedValue,
  label: 'Menos de 2 anios',  // 5% de aumento
  onChanged: onChanged,
),
CustomRadioOption<int>(
  value: 2,
  groupValue: selectedValue,
  label: '2 a 4 anios',       // 10% de aumento
  onChanged: onChanged,
),
CustomRadioOption<int>(
  value: 5,
  groupValue: selectedValue,
  label: '5 a 9 anios',       // 15% de aumento
  onChanged: onChanged,
```

```
),
CustomRadioOption<int>(
  value: 10,
  groupValue: selectedValue,
  label: '10 anios o mas',    // 20% de aumento
  onChanged: onChanged,
),
```

Listing 8: Grupo de radio buttons en radio_button_group.dart

Cada opción tiene un valor numérico que representa la antigüedad mínima del rango, permitiendo que el caso de uso aplique el porcentaje correcto según la selección del usuario.

Archivo: info_card.dart - Tarjeta de informacion para mostrar resultados:

```
class InfoCard extends StatelessWidget {
  final String title;
  final String value;
  final IconData icon;
  final Color color;

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: const EdgeInsets.all(20),
      decoration: BoxDecoration(
        color: color.withOpacity(0.1),
        borderRadius: BorderRadius.circular(16),
        border: Border.all(color: color.withOpacity(0.3)),
      ),
      child: Column(
        children: [
          Row(children: [Icon(icon, color: color), Text(title)]),
          Text(value, style: TextStyle(fontSize: 28, fontWeight: FontWeight.bold, color: color)),
        ],
      ),
    );
  }
}
```

Listing 9: Tarjeta de informacion en info_card.dart

3.4.3 Archivo: src/presentation/widgets/organisms/salary_calculator_form.dart

Este organismo combina átomos y moléculas para crear el formulario completo de cálculo. Implementa un sistema robusto de validación de datos en dos niveles antes de enviar la información al ViewModel.

```
void _handleCalculate() {
  if (_formKey.currentState!.validate()) {
    final salary = double.parse(_salaryController.text);
    widget.onCalculate(salary, _selectedAntiguedad);
  }
```

```
}

// Validacion en el campo de texto
validator: (value) {
  if (value == null || value.isEmpty) {
    return 'Por favor ingrese un salario';
  }
  if (double.tryParse(value) == null) {
    return 'Por favor ingrese un numero valido';
  }
  if (double.parse(value) <= 0) {
    return 'El salario debe ser mayor a 0';
  }
  return null;
}
```

Listing 10: Validación del formulario en salary_calculator_form.dart

El primer nivel de validación ocurre en cada campo de texto mediante la función `validator`, verificando que el campo no esté vacío, que el valor sea numérico válido y que sea mayor a cero. El segundo nivel ocurre en el método `_handleCalculate`, que valida el estado completo del formulario mediante `_formKey.currentState!.validate()` antes de proceder. Solo cuando ambos niveles de validación pasan se envían los datos al ViewModel, garantizando la integridad de los datos en toda la cadena de procesamiento.

3.5 Sistema de Rutas

3.5.1 Archivo: src/presentation/routes/app_routes.dart

Define las rutas de navegación de la aplicación, conectando las tres pantallas principales del sistema.

```
class AppRoutes {
  static const String home = '/';
  static const String calculateIncrease = '/calculate';
  static const String history = '/history';
  static const String pdfReport = '/pdf_report';

  static Map<String, WidgetBuilder> routes = {
    home: (context) => const HomePage(),
    calculateIncrease: (context) => const CalculateIncreaseScreen(),
    history: (context) => const HistoryScreen(),
    pdfReport: (context) => const PdfReportScreen(),
  };
}
```

Listing 11: Configuración de rutas en app_routes.dart

La configuración centralizada de rutas permite una navegación organizada y facilita el mantenimiento. Cada ruta está asociada a una pantalla específica mediante un mapa de constructores de

widgets.

3.6 Pantallas de la Aplicacion

3.6.1 Archivo: src/presentation/view/home_page.dart

Pantalla principal con navegacion a las tres funcionalidades de la aplicacion.

```dart
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Actividad 2.1 App Calc')),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          ElevatedButton.icon(
            icon: const Icon(Icons.calculate),
            label: const Text('Pantalla 1: Calcular Aumento'),
            onPressed: () => Navigator.pushNamed(context, AppRoutes.calculateIncrease),
          ),
          ElevatedButton.icon(
            icon: const Icon(Icons.history),
            label: const Text('Pantalla 2: Historial'),
            onPressed: () => Navigator.pushNamed(context, AppRoutes.history),
          ),
          ElevatedButton.icon(
            icon: const Icon(Icons.picture_as_pdf),
            label: const Text('Pantalla 3: Reporte PDF'),
            onPressed: () => Navigator.pushNamed(context, AppRoutes.pdfReport),
          ),
        ],
      ),
    );
  }
}
```

Listing 12: Pantalla principal en home_page.dart

3.6.2 Archivo: src/presentation/view/calculate_increase_screen.dart

Pantalla de calculo que utiliza Consumer para escuchar cambios en el ViewModel y mostrar resultados.

```dart
class CalculateIncreaseScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<OperarioViewModel>(
      builder: (context, viewModel, child) {
        return Scaffold(
          appBar: AppBar(title: const Text('Calcular Aumento')),
          body: SingleChildScrollView(
```

```
            child: Column(
              children: [
                SalaryCalculatorForm(
                  onCalculate: (salary, antiguedad) {
                    viewModel.calcular(salary, antiguedad);
                  },
                ),
                if (viewModel.resultado != null)
                  Card(
                    child: Column(
                      children: [
                        InfoCard(
                          title: 'Aumento Salarial',
                          value: '\$ ${viewModel.resultado!.aumento.toStringAsFixed(2)}',
                        ),
                        InfoCard(
                          title: 'Salario Final',
                          value: '\$ ${viewModel.resultado!.sueldoFinal.toStringAsFixed(2)}',
                        ),
                      ],
                    ),
                  ),
              ],
            ),
          ),
        );
      },
    );
  }
}
```

Listing 13: Pantalla de calculo en calculate_increase_screen.dart

3.6.3 Archivo: src/presentation/view/history_screen.dart

Pantalla de historial que muestra todos los calculos realizados durante la sesion.

```
class HistoryScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<OperarioViewModel>(
      builder: (context, viewModel, child) {
        return Scaffold(
          appBar: AppBar(title: const Text('Historial de Calculos')),
          body: viewModel.historial.isEmpty
              ? const Center(child: Text('Aun no has realizado ningun calculo.'))
              : ListView.builder(
                  itemCount: viewModel.historial.length,
                  itemBuilder: (context, index) {
                    final resultado = viewModel.historial[index];
                    return Card(
                      child: ListTile(
                        leading: CircleAvatar(child: Text('${index + 1}')),
```

```
                        title: Text('Salario Final: \$ ${resultado.sueldoFinal.toStringAsFixed(2)}
    '),
                        subtitle: Text('Aumento: \$ ${resultado.aumento.toStringAsFixed(2)}'),
                      ),
                    );
                  },
                ),
        );
      },
    );
  }
}
```

Listing 14: Pantalla de historial en history_screen.dart

3.7 Sistema de Temas

3.7.1 Archivo: temas/esquema_color.dart

Define la paleta de colores centralizada de la aplicación con un esquema basado en tonos verdes.

```
class ColorApp{
  static const Color primaio = Color(0xFF2E7D32);
  static const Color secundario = Color(0xFF66BB6A);
  static const Color acento = Color(0xFF26C6DA);
  static const Color exito = Color(0xFF42A5F5);
  static const Color error = Color(0xFFEF5350);
  static const Color textoOscuro = Color(0xFF212121);
  static const Color textoClaro = Color(0xFFFFFFFF);
  static const Color fondo = Color(0xFFF1F8F3);
}
```

Listing 15: Definición de colores en esquema_color.dart

Los colores se definen como constantes estáticas permitiendo su acceso global sin necesidad de instanciar la clase. El color primario verde oscuro (0xFF2E7D32) establece la identidad visual de la aplicación, mientras que el secundario proporciona variación para elementos de acento.

3.7.2 Archivo: temas/tema_general.dart

Configura el tema global integrando todos los componentes de estilo en un objeto `ThemeData` que se aplica en el `MaterialApp`.

```
class TemaGeneral {
  static ThemeData claro = ThemeData(
      useMaterial3: true,
      colorScheme: ColorScheme.light(
        primary: ColorApp.primaio,
        secondary: ColorApp.secundario,
        surface: ColorApp.fondo,
```

```
        onPrimary: ColorApp.textoClaro,
        onSecondary: Colors.white,
      ),
      textTheme: TipografiaApp.texto,
      appBarTheme: TemaAppBar.estilo,
      elevatedButtonTheme: TemaBotones.botonPrincipal,
      outlinedButtonTheme: TemaBotones.botonSecundario,
      inputDecorationTheme: TemaFormulario.campoTexto,
      scaffoldBackgroundColor: ColorApp.fondo
  );
}
```

Listing 16: Configuración del tema en tema_general.dart

El tema utiliza Material Design 3 y combina configuraciones de múltiples archivos especializados. Esta estructura modular permite modificar aspectos específicos del diseño sin afectar otros componentes, manteniendo la coherencia visual en toda la aplicación.

3.8 Punto de Entrada de la Aplicación

3.8.1 Archivo: main.dart

El archivo principal configura Provider como gestor de estado y establece el tema y rutas iniciales de la aplicación.

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => OperarioViewModel(CalcularAumentoUseCase()),
      child: const MyApp(),
    ),
  );
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Aumento de Salario',
      debugShowCheckedModeBanner: false,
      theme: TemaGeneral.claro,
      initialRoute: AppRoutes.home,
      routes: AppRoutes.routes,
    );
  }
}
```

Listing 17: Configuración inicial en main.dart

El `ChangeNotifierProvider` envuelve toda la aplicación, inyectando el ViewModel con su dependencia del caso de uso. Esto permite que cualquier widget descendiente acceda al

estado compartido mediante `Provider.of` o `Consumer`. La configuración del MaterialApp establece el tema claro y define la ruta inicial como la página de inicio.

3.9 Dependencias del Proyecto

3.9.1 Archivo: pubspec.yaml

El archivo de configuración de dependencias define las librerías externas necesarias para el funcionamiento de la aplicación.

```
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.8
  provider: ^6.0.0          # Gestion de estado
  pdf: ^3.10.8              # Generacion de documentos PDF
  path_provider: ^2.1.4     # Acceso a directorios del sistema
  share_plus: ^10.0.0       # Compartir archivos
```

Listing 18: Dependencias en pubspec.yaml

Las dependencias clave incluyen: `provider` para la gestión reactiva del estado, `pdf` para la creación de documentos PDF, `path_provider` para acceder al directorio temporal del dispositivo, y `share_plus` para compartir archivos con otras aplicaciones.

3.10 Generación de Reportes PDF

3.10.1 Archivo: src/presentation/view/pdf_report_screen.dart

Esta pantalla implementa la funcionalidad de generación y exportación de reportes PDF utilizando las librerías `pdf`, `path_provider` y `share_plus`.

```
Future<Uint8List> _generatePdfBytes(List historial) async {
  final pdf = pw.Document();

  pdf.addPage(
    pw.Page(
      pageFormat: PdfPageFormat.a4,
      build: (pw.Context context) {
        return pw.Column(
          children: [
            pw.Text('Reporte de Calculos Salariales',
              style: pw.TextStyle(fontSize: 24)),
            pw.Table(
              border: pw.TableBorder.all(),
              children: [
                // Encabezados y filas de datos
                ...historial.map((resultado) => pw.TableRow(...)),
              ],
            ),
          ],
```

```
        );
      },
    ),
  );
  return pdf.save();
}

Future<File> _savePdfToTemp(Uint8List bytes) async {
  final output = await getTemporaryDirectory();
  final file = File('${output.path}/reporte_salarial.pdf');
  await file.writeAsBytes(bytes);
  return file;
}

// Compartir el PDF generado
await Share.shareXFiles(
  [XFile(file.path)],
  text: 'Reporte de Calculos Salariales',
);
```

Listing 19: Generación de PDF en pdf_report_screen.dart

El método `_generatePdfBytes` crea un documento PDF con formato A4, incluyendo un título, la fecha actual y una tabla con todos los cálculos del historial. El método `_savePdfToTemp` guarda el archivo en el directorio temporal del dispositivo mediante `getTemporaryDirectory()`, evitando problemas de permisos. Finalmente, `Share.shareXFiles` permite compartir el PDF con otras aplicaciones del dispositivo.

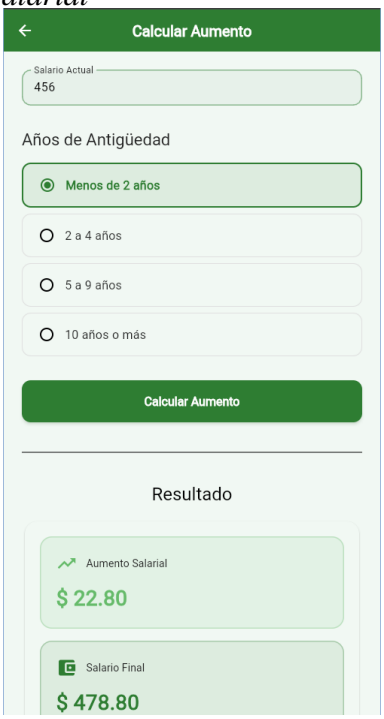
4 Resultados

Los resultados obtenidos durante el desarrollo del proyecto demuestran el cumplimiento satisfactorio de los objetivos planteados. La aplicación implementa exitosamente las tres pantallas requeridas con funcionalidad completa.

La Pantalla 1 (Calcular Aumento) presenta un formulario funcional que permite ingresar el salario actual y seleccionar la antigüedad del operario mediante radio buttons. El sistema valida los datos de entrada y muestra el resultado del cálculo inmediatamente en la misma pantalla, incluyendo el monto del aumento y el salario final.

Figura 1

Pantalla de Calculo de Aumento Salarial

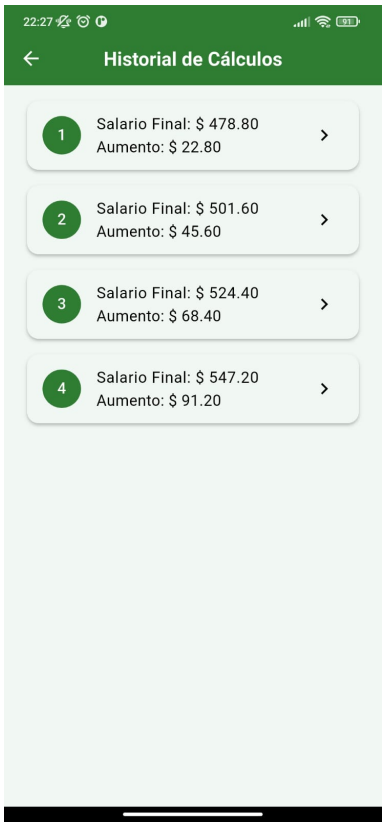


Nota. Captura de pantalla mostrando el formulario de calculo con campos de salario y seleccion de antiguedad.

La Pantalla 2 (Historial) muestra una lista organizada de todos los cálculos realizados durante la sesión. Cada registro presenta el número de operación, el salario final y el monto del aumento aplicado, permitiendo al usuario revisar las operaciones anteriores.

Figura 2

Pantalla de Historial de Calculos

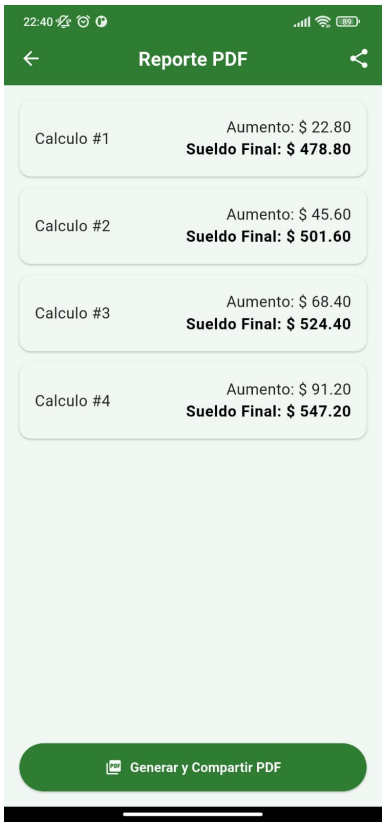


Nota. Captura de pantalla mostrando la lista de calculos realizados durante la sesion.

La Pantalla 3 (Reporte PDF) presenta la lista de cálculos en formato de reporte con opciones para generar y descargar el documento PDF, además de incluir funcionalidad para compartir el reporte mediante el botón en la barra de aplicación.

Figura 3

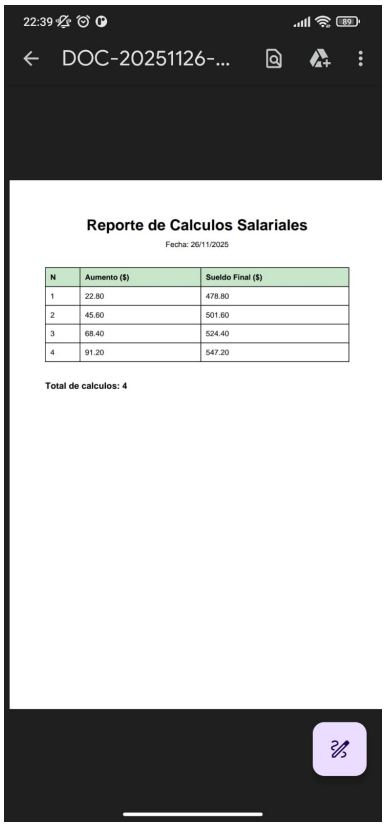
Pantalla de Generacion de Reporte PDF



Nota. Captura de pantalla mostrando la interfaz para generar y compartir el reporte PDF.

Figura 4

Vista Previa del PDF Generado



Nota. Captura de pantalla mostrando el documento PDF generado con los calculos salariales.

La arquitectura implementada demuestra una clara separación de responsabilidades, donde la capa de dominio permanece independiente de la interfaz de usuario. El sistema de temas proporciona una apariencia visual coherente con el esquema de colores verde definido.

5 Conclusiones

En conclusión, la investigación de los fundamentos teóricos de Clean Architecture, Atomic Design y Provider permitió establecer una base conceptual sólida que guió las decisiones de diseño durante todo el desarrollo, resultando en una estructura de código organizada y comprensible. Este conocimiento previo fue fundamental para comprender la separación de responsabilidades y aplicar correctamente cada patrón en el contexto del proyecto.

Se puede afirmar que la implementación de la capa de dominio con entidades inmutables y casos de uso especializados demostró ser efectiva para encapsular la lógica de negocio del cálculo de aumentos salariales, facilitando su modificación y prueba de manera independiente. La clase CalcularAumentoUseCase centraliza las reglas de negocio, permitiendo que cualquier cambio en los porcentajes de aumento se realice en un único punto del código.

En síntesis, el desarrollo de la capa de presentación siguiendo Atomic Design produjo componentes altamente reutilizables, donde los átomos, moléculas y organismos creados pueden emplearse en futuras pantallas sin modificaciones significativas. Esta metodología facilitó la construcción progresiva de la interfaz, comenzando desde elementos simples como campos de texto hasta formularios completos con validación integrada.

Finalmente, la creación del sistema de historial mediante Provider permitió compartir el estado entre las tres pantallas de manera eficiente, demostrando las ventajas de la gestión de estado centralizada en aplicaciones Flutter. El ViewModel mantiene una lista persistente de todos los cálculos realizados, accesible desde cualquier pantalla sin necesidad de pasar datos manualmente entre widgets.

6 Recomendaciones

Se recomienda profundizar en el estudio de patrones arquitectónicos adicionales como Repository Pattern y Dependency Injection para complementar la comprensión de Clean Architecture en proyectos de mayor escala.

Para futuras versiones, se sugiere implementar persistencia de datos mediante SQLite o Hive para mantener el historial de cálculos entre sesiones de la aplicación, extendiendo la funcionalidad de la capa de dominio.

Se aconseja expandir la biblioteca de componentes Atomic Design creando variantes adicionales de átomos y moléculas que puedan adaptarse a diferentes contextos de uso dentro de la aplicación.

Se propone ampliar la funcionalidad de generación de PDF agregando opciones de personalización como selección de rango de fechas, filtros por monto de aumento y exportación en formatos adicionales como Excel o CSV.

7 Referencias Bibliográficas

Flutter. (2024). Provider package. Pub.dev. https://pub.dev/packages/provider

Frost, B. (2016). Atomic Design. Brad Frost. https://atomicdesign.bradfrost.com/

Google. (2024). Flutter documentation. Flutter.dev. https://docs.flutter.dev/

Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

8 Anexos

El código fuente completo del proyecto se encuentra disponible en el siguiente repositorio de GitHub:

`https://github.com/AMVMesias/Desarrollo-Movil/tree/main/2P/App%20Calcular%20Sueldo`

El repositorio incluye toda la estructura del proyecto con los archivos de configuración, código fuente organizado según Clean Architecture, sistema de temas y documentación adicional.