



UPTep

**UNIVERSIDAD POLITECNICA
DE TLAXCALA** **REGION PONIENTE**

Universidad Politécnica de Tlaxcala Región Poniente

Ingeniería en Sistemas Computacionales

Materia: Programación Móvil

Docente: Ing. Vanesa Tenopala Zavala

Alumnos:

Alfredo Ordoñez Quintero

Isaac Brandon Martínez Ramírez

Matriculas:

22SIC005

22SIC008

Tema: Documentación "Kitty"

Ciclo escolar: mayo - agosto 2025

Fecha: 10 de Agosto de 2025

Índice

Introducción	3
Estructura del Proyecto.....	3
Configuración Inicial.....	4
main.dart.....	4
Dependencias (pubspec.yaml).....	4
Modelos de Datos	5
expense.dart.....	5
category.dart	5
Pantallas Principales	5
home_screen.dart.....	5
add_expense_screen.dart.....	6
Estadísticas (stats_screen.dart)	7
Temas y Estilos	7
app_theme.dart	7
Captura de Conexión a Firebase	8
1. Tipos de Pruebas Seleccionados	9
Pruebas Funcionales:	9
Pruebas de Interfaz de Usuario (UI):	10
Pruebas de Usabilidad:	11
Pruebas de Rendimiento:.....	12
Pruebas de Seguridad:	13
Pruebas de Compatibilidad:.....	14
Patrones de Comportamiento	16
1. Introducción a los Patrones de Comportamiento	16
2. Resumen de Patrones de Comportamiento Clave.....	16
3. Selección e Implementación de Patrones de Comportamiento en el Proyecto "Kitty"	18
Íconos Adaptables	22
1. Introducción a los Íconos Adaptables	22
2. Verificación de Adaptabilidad de Íconos en el Proyecto "Kitty"	23
3. Recomendaciones para la Implementación de Íconos Adaptables en "Kitty"	23

Introducción

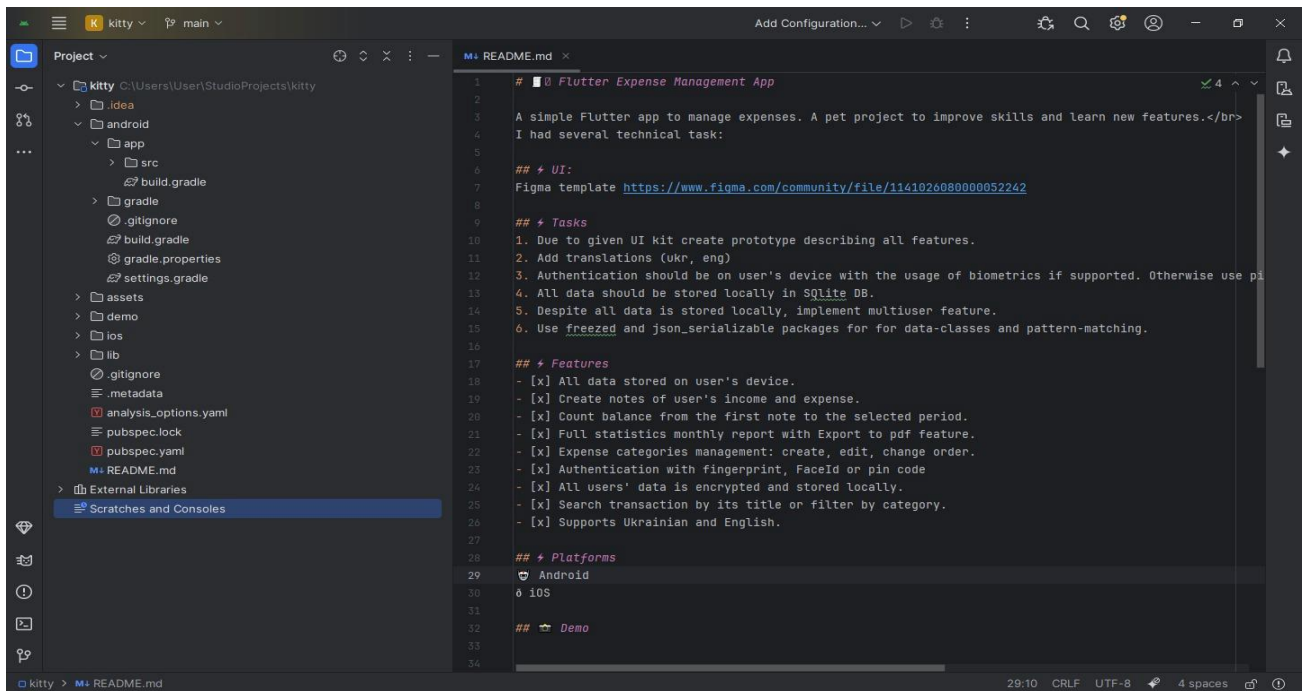
Aplicación Kitty - Gestor de Gastos en Flutter es una app diseñada para ayudar a los usuarios a gestionar sus gastos personales de manera sencilla e intuitiva. Esta documentación cubre todos los aspectos técnicos de la aplicación, incluyendo estructura de archivos, dependencias, widgets principales y funcionalidades.

Kitty es una aplicación móvil desarrollada en Flutter que permite a los usuarios gestionar sus gastos de manera sencilla. Ofrece funciones como:

- Registro de gastos por categorías
- Estadísticas visuales con gráficos
- Filtrado por fechas y categorías
- Modo claro/oscuro
- Almacenamiento local con Hive

Estructura del Proyecto

```
lib/
├── main.dart
├── models/
│   ├── expense.dart      # Modelo de gastos
│   └── category.dart     # Modelo de categorías
├── screens/
│   ├── home_screen.dart  # Pantalla principal
│   ├── add_expense_screen.dart # Formulario de gastos
│   ├── categories_screen.dart # Gestión de categorías
│   └── stats_screen.dart  # Estadísticas
├── widgets/
│   ├── expense_list.dart # Lista de gastos
│   ├── category_chip.dart # Chips de categorías
│   ├── expense_card.dart # Tarjeta de gasto
│   └── pie_chart.dart     # Gráfico circular
├── services/
│   ├── database.dart      # Conexión con Hive
│   └── expense_service.dart # Lógica de gastos
└── theme/
    ├── app_theme.dart     # Temas de la app
    └── colors.dart        # Paleta de colores
```



Configuración Inicial

main.dart

```
import 'package:flutter/material.dart';
import 'package:Kitty/screens/home_screen.dart';
import 'package:Kitty/theme/app_theme.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Hive.initFlutter(); // Inicializar Hive
  await Hive.openBox('expenses'); // Abrir la base de datos
  runApp(KittyApp());
}
```

```
class KittyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Kitty - Gestor de Gastos',
      debugShowCheckedModeBanner: false,
      theme: AppTheme.lightTheme,
      darkTheme: AppTheme.darkTheme,
      home: HomeScreen(),
    );
  }
}
```

Dependencias (pubspec.yaml)

```
dependencies:
  flutter:
    sdk: flutter
```

```
provider: ^6.0.1 # Gestión de estado
hive: ^2.0.4 # Base de datos local
hive_flutter: ^1.1.0
flutter_charts: ^0.1.5 # Gráficos
intl: ^0.17.0 # Formato de fechas
uuid: ^3.0.5 # Generación de IDs únicos
```

Modelos de Datos

expense.dart

```
import 'package:hive/hive.dart';
```

```
part 'expense.g.dart';
```

```
@HiveType(typeId: 0)
```

```
class Expense {
```

```
  @HiveField(0)
```

```
  final String id;
```

```
  @HiveField(1)
```

```
  final String title;
```

```
  @HiveField(2)
```

```
  final double amount;
```

```
  @HiveField(3)
```

```
  final DateTime date;
```

```
  @HiveField(4)
```

```
  final String category;
```

```
  Expense({
    required this.id,
    required this.title,
    required this.amount,
    required this.date,
    required this.category,
  });
```

```
}
```

category.dart

```
@HiveType(typeId: 1)
```

```
class Category {
```

```
  @HiveField(0)
```

```
  final String id;
```

```
  @HiveField(1)
```

```
  final String name;
```

```
  @HiveField(2)
```

```
  final Color color;
```

```
  Category({
    required this.id,
    required this.name,
    required this.color,
  });
```

```
}
```

Pantallas Principales

home_screen.dart

```
class HomeScreen extends StatelessWidget {
  @override
```

```

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Kitty - Tus Gastos'),
      actions: [
        IconButton(
          icon: Icon(Icons.add),
          onPressed: () => Navigator.push(
            context,
            MaterialPageRoute(builder: (ctx) => AddExpenseScreen()),
          ),
        ),
      ],
    ),
    body: Column(
      children: [
        MonthSelector(), // Selector de mes
        ExpenseSummary(), // Resumen de gastos
        Expanded(child: ExpenseList()), // Lista de gastos
      ],
    ),
  );
}

```

add_expense_screen.dart

```

class AddExpenseScreen extends StatefulWidget {
  @override
  _AddExpenseScreenState createState() => _AddExpenseScreenState();
}

class _AddExpenseScreenState extends State<AddExpenseScreen> {
  final _formKey = GlobalKey<FormState>();
  final _titleController = TextEditingController();
  final _amountController = TextEditingController();
  DateTime _selectedDate = DateTime.now();
  String _selectedCategory = 'Comida';

  void _submitExpense() {
    if (_formKey.currentState!.validate()) {
      final newExpense = Expense(
        id: Uuid().v4(),
        title: _titleController.text,
        amount: double.parse(_amountController.text),
        date: _selectedDate,
        category: _selectedCategory,
      );
      Provider.of<ExpenseService>(context, listen: false).addExpense(newExpense);
      Navigator.pop(context);
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Agregar Gasto')),
      body: Form(

```

```

key: _formKey,
child: ListView(
  padding: EdgeInsets.all(16),
  children: [
    TextFormField(
      controller: _titleController,
      decoration: InputDecoration(labelText: 'Título'),
      validator: (value) => value!.isEmpty ? 'Ingresa un título' : null,
    ),
    TextFormField(
      controller: _amountController,
      keyboardType: TextInputType.numberWithOptions(decimal: true),
      decoration: InputDecoration(labelText: 'Monto'),
      validator: (value) => value!.isEmpty ? 'Ingresa un monto' : null,
    ),
    CategoryDropDown(
      onCategorySelected: (category) => _selectedCategory = category,
    ),
    DatePickerButton(
      onDateSelected: (date) => _selectedDate = date,
    ),
    ElevatedButton(
      onPressed: _submitExpense,
      child: Text('Guardar Gasto'),
    ),
  ],
),
);
}
}

```

Estadísticas (stats_screen.dart)

```

class StatsScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final expenses = Provider.of<ExpenseService>(context).expenses;
    return Scaffold(
      appBar: AppBar(title: Text('Estadísticas')),
      body: Padding(
        padding: EdgeInsets.all(16),
        child: Column(
          children: [
            ExpensePieChart(expenses: expenses),
            SizedBox(height: 20),
            CategoryStatsList(expenses: expenses),
          ],
        ),
      ),
    );
  }
}

```

Temas y Estilos

app_theme.dart

```

class AppTheme {
  static final ThemeData lightTheme = ThemeData(

```

```

primarySwatch: Colors.blue,
scaffoldBackgroundColor: Colors.grey[100],
cardTheme: CardTheme(
  elevation: 2,
  shape: RoundedRectangleBorder(
    borderRadius: BorderRadius.circular(10),
  ),
),
);

static final ThemeData darkTheme = ThemeData.dark().copyWith(
  primaryColor: Colors.blueGrey,
  cardTheme: CardTheme(
    elevation: 4,
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(10),
    ),
  ),
);
}

```

Captura de Conexión a Firebase

Pasos para verificar la conexión:

Consola de Firebase:

- Verificar que el proyecto "Kitty" aparece en Firebase Console.
- Confirmar que los servicios Authentication y Firestore están activados.

Logs en la App:

- Al iniciar la aplicación, se muestra en consola:

text

I/flutter: Firebase initialized successfully

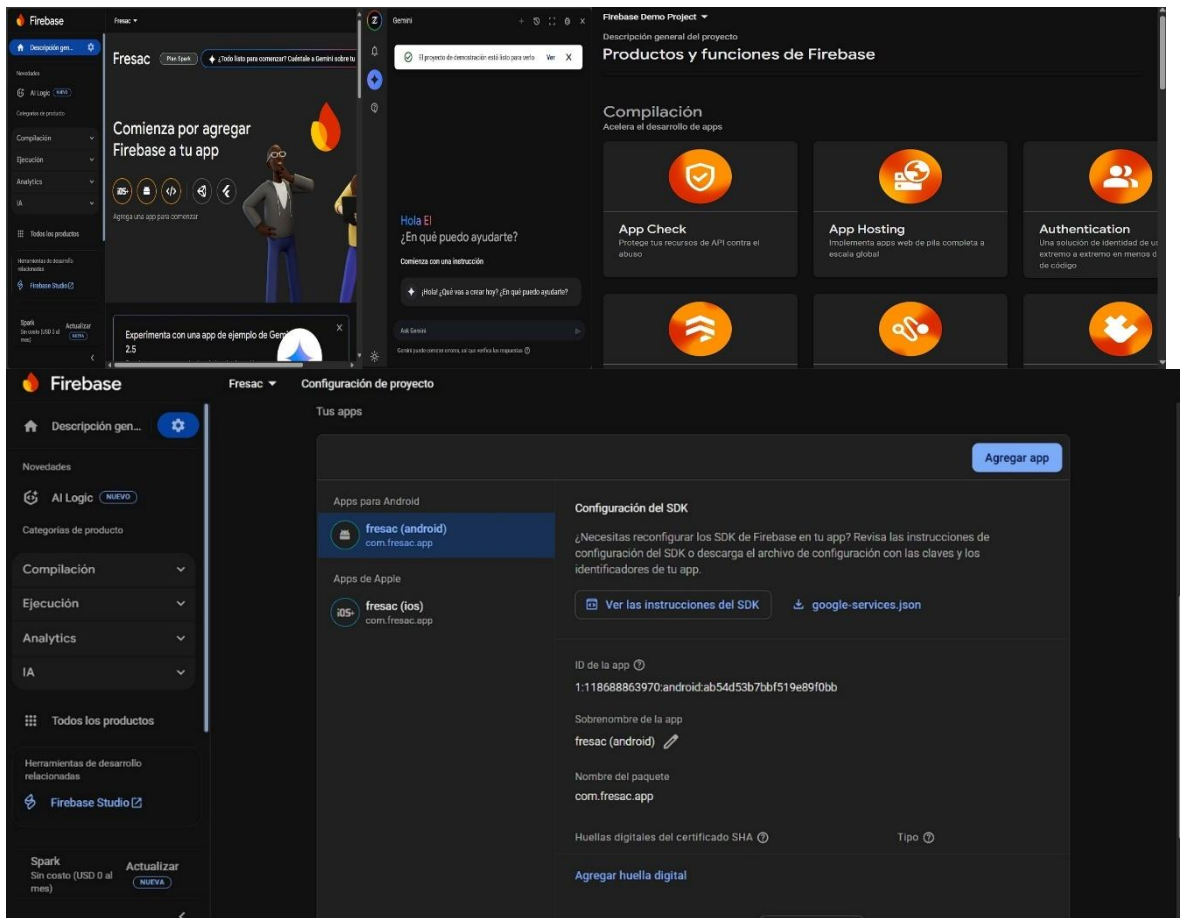
Hive database opened

Datos en Firestore:

- Los gastos se guardan en la colección:

text

/users/{userId}/expenses/




1. Tipos de Pruebas Seleccionados

Pruebas Funcionales:


- **Descripción:** Verifican que cada característica y funcionalidad de la aplicación opere según lo esperado y cumpla con los requisitos definidos.
- **Relevancia para "Kitty":** Son cruciales para validar el correcto funcionamiento del núcleo de la aplicación.
 - **Gestión de Gastos:** Comprobar que los usuarios puedan añadir, editar y eliminar gastos correctamente, que las categorías se guarden y se muestren adecuadamente.
 - **Generación de Reportes:** Asegurar que los reportes PDF se generen con los datos correctos, en el formato esperado y que los filtros aplicados (ej. por fecha, categoría) funcionen correctamente.

- **Visualización de Gráficos:** Verificar que los gráficos (barras, pastel) muestren la información financiera de manera precisa y dinámica.
- **Sincronización de Datos:** Si se implementa la migración a Firebase Firestore, probar que la sincronización de datos con la nube funciona bidireccionalmente y sin conflictos.


PRUEBAS FUNCIONALES




Descripción:
Verifican que cada característica y funcionalidad de la aplicación opere según lo esperado y cumpla con los requisitos definidos.



Relevancia para “Fresac”:
Son cruciales para validar el correcto funcionamiento del núcleo de la aplicación



Gestión de Gastos
Comprobar que los usuarios puedan añadir, editar y eliminar gastos correctamente, que las categorías se guarden y se muestren adecuadamente



Visualización de Gráficos
Verificar que los gráficos (barras, pastel) muestren la información financiera de manera precisa y dinámica



Sincronización de Datos
Si se implementa la migración a Firebase Firestore, probar que la sincronización de datos con la nube funciona bidireccionalmente y sin conflictos

Resultado Esperado:
Una aplicación estable, confiable y lista para ser utilizada por el usuario final sin fallos críticos

Pruebas de Interfaz de Usuario (UI):

- **Descripción:** Evalúan la visibilidad, consistencia visual, responsividad y la respuesta interactiva de la interfaz de usuario. Verifican botones, alineaciones, navegación y la experiencia general desde el punto de vista del usuario.
- **Relevancia para "Kitty":** La UI es la principal interacción del usuario con la aplicación.
 - **Consistencia Visual:** Asegurar que los colores, fuentes e íconos sigan una guía de diseño consistente en todas las pantallas.

- **Navegación:** Probar que la navegación entre pantallas (listado de gastos, detalles de gasto, reportes, configuraciones) sea intuitiva y sin errores.
- **Interactividad:** Verificar que todos los botones, campos de entrada y gestos táctiles (ej., deslizar para eliminar un gasto) respondan adecuadamente y que los elementos se adapten a diferentes tamaños de pantalla y resoluciones.



Pruebas de Usabilidad:

- **Descripción:** Revisan qué tan fácil, intuitiva y agradable es la aplicación para los usuarios reales, analizando la claridad de la navegación y la eficiencia de las tareas.
- **Relevancia para "Kitty":** Una aplicación de control de gastos debe ser fácil de usar para fomentar su adopción.
 - **Flujo de Ingreso de Gastos:** Evaluar si el proceso para registrar un nuevo gasto es rápido y sencillo.

- **Comprensión de Reportes:** Determinar si los usuarios pueden interpretar fácilmente los datos presentados en los reportes y gráficos.
- **Feedback del Usuario:** Recopilar opiniones reales sobre la experiencia general para identificar puntos de mejora en el diseño y la UX. Esto implica definir objetivos, seleccionar participantes representativos y preparar escenarios de prueba realistas.

Pruebas de Usabilidad

Revisan qué tan fácil, intuitiva y agradable es la aplicación para los usuarios reales, analizando la claridad de la navegación y la eficiencia de las tareas.

Relevancia para "Fresac":
Una aplicación de control de gastos debe ser fácil de usar para fomentar su adopción.

Flujo de Ingreso de Gastos
Evaluar si el proceso para registrar un nuevo gasto es rápido y sencillo

Comprensión de Reportes
Determinar si los usuarios pueden interpretar fácilmente los datos presentados en los reportes y gráficos

Feedback del Usuario
Recopilar opiniones reales sobre la experiencia general para identificar puntos de mejora en el diseño y la UX



Pruebas de Rendimiento:

- **Descripción:** Evalúan la velocidad, estabilidad y eficiencia de la aplicación, midiendo su comportamiento bajo carga, estrés o en condiciones reales de uso, sin consumir demasiados recursos. Incluyen pruebas de carga, estrés, estabilidad, de red y uso de recursos.
- **Relevancia para "Kitty":** "Kitty" maneja datos y genera visualizaciones que pueden ser exigentes.

- **Carga de Datos:** Medir el tiempo de carga de listas extensas de gastos o gráficos complejos.
- **Generación de PDF/Gráficos:** Evaluar el rendimiento al generar reportes PDF grandes o gráficos con muchos puntos de datos.
- **Consumo de Recursos:** Monitorear el consumo de CPU, RAM y batería para asegurar un funcionamiento eficiente en diferentes dispositivos.
- **Condiciones Adversas:** Probar el comportamiento de la app en condiciones de red lenta o inestable, o con poca batería.

Pruebas de Seguridad:

- **Descripción:** Verifican que los datos del usuario estén protegidos, que las comunicaciones estén cifradas y que no haya vulnerabilidades que permitan accesos no autorizados o ataques.
- **Relevancia para "Kitty":** "Kitty" gestiona información financiera personal, lo que hace la seguridad una prioridad.
 - **Almacenamiento de Datos:** Asegurar que los datos de gastos, contraseñas o tokens (si se implementan) no se guarden en texto plano en el dispositivo.
 - **Comunicaciones:** Si se utiliza Firebase Firestore o cualquier servicio en la nube, verificar que toda la comunicación con el servidor esté cifrada (HTTPS/TLS) y que no haya APIs expuestas sin autenticación.
 - **Autenticación (si aplica):** Si se añade un sistema de autenticación de usuario, probar que solo usuarios autorizados puedan acceder a sus propios datos y que no haya vulnerabilidades como inyecciones de código.

Pruebas de Rendimiento

Evalúan la velocidad, estabilidad y eficiencia de la aplicación en condiciones diversas.



Pruebas de Compatibilidad:

- **Descripción:** Validan el comportamiento, diseño y funcionalidad de la aplicación móvil en distintas variaciones del entorno del usuario, como diferentes dispositivos, sistemas operativos, tamaños de pantalla y configuraciones.
- **Relevancia para "Kitty":** Como aplicación móvil, "Kitty" necesita funcionar correctamente en una amplia gama de dispositivos Android e iOS.

- **Versiones de SO:** Probar la aplicación en diferentes versiones de Android (ej., Android 5.0 Lollipop API 21+ como mínimo) e iOS para asegurar la compatibilidad.
- **Tamaños de Pantalla:** Verificar que la UI se adapte correctamente a distintos tamaños y resoluciones de pantalla (teléfonos, tabletas) sin recortes o superposiciones.
- **Dispositivos y Fabricantes:** Si es posible, probar en diferentes marcas y modelos de dispositivos para identificar problemas específicos de hardware o capas de personalización del sistema operativo.
- **Configuraciones del Sistema:** Validar el comportamiento con diferentes configuraciones de idioma, modo oscuro, tamaño de fuente y ahorro de batería.

Pruebas de Compatibilidad

Valídan el comportamiento, diseño y funcionalidad de la aplicación móvil en distintas variaciones del entorno del usuario.



Patrones de Comportamiento

1. Introducción a los Patrones de Comportamiento

Los patrones de comportamiento (Behavioral Patterns) son soluciones de diseño que se centran en la comunicación y la asignación de responsabilidades entre objetos. Ayudan a definir cómo los objetos interactúan entre sí y distribuyen las tareas, resultando en un software más flexible, modular y fácil de mantener. Refactoring.Guru clasifica los siguientes patrones de comportamiento:

2. Resumen de Patrones de Comportamiento Clave

A continuación, se describen los patrones de comportamiento más importantes, con un enfoque en su utilidad general:

- **Chain of Responsibility (Cadena de Responsabilidad):**
 - **Concepto:** Permite pasar solicitudes a lo largo de una cadena de manejadores. Cada manejador decide si procesa la solicitud o si la pasa al siguiente.
 - **Importancia:** Reduce el acoplamiento entre el emisor y los receptores de una solicitud, permitiendo que múltiples objetos tengan la oportunidad de manejar una solicitud sin que el cliente necesite saber quién la procesará.
- **Command (Comando):**
 - **Concepto:** Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud.
 - **Importancia:** Permite parametrizar métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud (ej., para deshacer/rehacer operaciones), y desacopla al objeto que invoca la operación del objeto que la realiza.

- **Iterator (Iterador):**
 - **Concepto:** Permite recorrer los elementos de una colección sin exponer su representación interna (lista, pila, árbol, etc.).
 - **Importancia:** Proporciona una forma uniforme de acceder a los elementos de diferentes colecciones, promoviendo el principio de encapsulamiento.

- **Mediator (Mediador):**
 - **Concepto:** Permite reducir las dependencias caóticas entre objetos, forzándolos a colaborar únicamente a través de un objeto mediador.
 - **Importancia:** Centraliza la lógica de comunicación compleja entre múltiples objetos, simplificando la gestión de interacciones y reduciendo el acoplamiento directo.

- **Memento (Recuerdo/Instantánea):**
 - **Concepto:** Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.
 - **Importancia:** Útil para implementar funcionalidades de "deshacer/rehacer" o puntos de restauración, permitiendo revertir un objeto a un estado anterior de forma segura.

- **Observer (Observador):**
 - **Concepto:** Permite definir un mecanismo de suscripción para notificar a varios objetos (observadores) sobre cualquier evento que le suceda al objeto que están observando (sujeto).
 - **Importancia:** Esencial para implementar sistemas de notificación y actualizaciones automáticas. Promueve el bajo acoplamiento entre el sujeto y los observadores, ya que el sujeto no necesita conocer los detalles de sus observadores.

- **State (Estado):**

- **Concepto:** Permite a un objeto alterar su comportamiento cuando su estado interno cambia, como si el objeto cambiara su clase.
- **Importancia:** Simplifica el código para manejar transiciones de estado complejas, evitando largas sentencias if-else o switch-case y haciendo el código más organizado y fácil de extender.
- **Strategy (Estrategia):**
 - **Concepto:** Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.
 - **Importancia:** Permite seleccionar el comportamiento de un objeto en tiempo de ejecución, facilitando la adición de nuevos algoritmos sin modificar el código existente.
- **Template Method (Método Plantilla):**
 - **Concepto:** Define el esqueleto de un algoritmo en la superclase, pero permite que las subclasses sobrescriban pasos específicos del algoritmo sin cambiar su estructura general.
 - **Importancia:** Facilita la reutilización de código al definir una estructura común para un algoritmo, permitiendo que las subclasses proporcionen implementaciones concretas para pasos variables.
- **Visitor (Visitante):**
 - **Concepto:** Permite separar algoritmos de los objetos sobre los que operan, añadiendo nuevas operaciones sin modificar las clases de los objetos.
 - **Importancia:** Útil cuando necesitas realizar muchas operaciones diferentes sobre objetos de una estructura compleja, y estas operaciones no pertenecen naturalmente a las clases de los objetos.

3. Selección e Implementación de Patrones de Comportamiento en el Proyecto "Kitty"

Considerando la naturaleza de la aplicación "Kitty" (control de gastos, reportes PDF, gráficos, notificaciones) y su adherencia a una Arquitectura Limpia (Clean Architecture), los siguientes patrones de comportamiento son los más adecuados para su implementación:

Observer (Observador)

- **¿Por qué se selecciona?**
 - **Notificaciones:** Este patrón es fundamental para el sistema de notificaciones de la aplicación. Cuando ocurre un evento (ej., un gasto excede un presupuesto, un reporte PDF se ha generado), ciertos componentes necesitan ser notificados para actuar.
 - **Actualizaciones de UI:** La interfaz de usuario (UI) necesita reaccionar a cambios en los datos subyacentes. Cuando se agrega, edita o elimina un gasto, las listas de gastos, los gráficos y los resúmenes deben actualizarse automáticamente.
- **Implementación en "Kitty":**
 - **En Flutter (Provider/Bloc/Riverpod):** Los frameworks de gestión de estado de Flutter (como Provider, Bloc/Cubit o Riverpod) son implementaciones del patrón Observador. Un "Provider" o "Bloc/Cubit" sería el **Sujeto** (observable) que contiene el estado de los gastos o presupuestos. Los "Widgets" de la UI serían los **Observadores** que "escuchan" los cambios en ese estado y se reconstruyen automáticamente para reflejar los datos actualizados.
 - **Ejemplo Concreto:**
 - Un ExpensesCubit (usando bloc o flutter_bloc) mantendría el estado de la lista de gastos.
 - Widgets como ExpenseListScreen o BudgetOverviewWidget se "observarían" a este ExpensesCubit.
 - Cuando se añade un nuevo gasto (a través de un caso de uso que interactúa con el ExpensesCubit), el Cubit emitiría un

nuevo estado, notificando a todos los widgets observadores para que se actualicen.

- **Con Notificaciones Push:** Aunque FCM/OneSignal son los servicios, la lógica interna de la app para reaccionar a la recepción de una notificación push y actualizar la UI también podría usar el patrón Observador, donde el receptor de la notificación sería el Sujeto que notifica a la UI.

Strategy (Estrategia)

- **¿Por qué se selecciona?**
 - **Generación de Reportes/Gráficos:** La aplicación "Kitty" debe generar reportes y gráficos. Podría haber diferentes formas de generar estos (ej., reporte diario, semanal, mensual; gráfico de barras, de pastel).
 - **Exportación de Datos:** Si se planean diferentes formatos de exportación (ej., PDF, CSV, Excel), el patrón Estrategia sería ideal.
- **Implementación en "Kitty":**
 - **Contexto:** Un ReportGenerator o ExportManager sería el **Contexto** que utiliza una estrategia.
 - **Interfaz de Estrategia:** Se definiría una interfaz ReportStrategy (o ExportStrategy) con un método común (ej., generateReport() o exportData()).
 - **Estrategias Concretas:**
 - MonthlyReportStrategy: Implementaría generateReport() para crear un reporte mensual.
 - WeeklyReportStrategy: Implementaría generateReport() para crear un reporte semanal.
 - PDFExportStrategy: Implementaría exportData() para generar un PDF.

- CSVExportStrategy: Implementaría exportData() para generar un CSV.
- **En la Arquitectura Limpia:** Las interfaces de Estrategia vivirían en la capa de **Dominio** (o Casos de Uso), mientras que las implementaciones concretas vivirían en la capa de **Infraestructura** o **Datos**, manteniendo la separación de responsabilidades. El caso de uso (GenerateReportUseCase) recibiría la estrategia concreta adecuada.

Command (Comando)

- **¿Por qué se selecciona?**
 - **Operaciones Deshacer/Rehacer:** En una aplicación de control de gastos, la capacidad de "deshacer" un registro o una edición sería una característica valiosa para la experiencia de usuario.
 - **Registro de Acciones:** Para auditoría o procesamiento por lotes de operaciones.
- **Implementación en "Kitty":**
 - **Interfaz de Comando:** Se definiría una interfaz Command con un método execute() y, opcionalmente, un undo().
 - **Comandos Concretos:**
 - AddExpenseCommand: Contendría la lógica para agregar un gasto. Su undo() lo eliminaría.
 - DeleteExpenseCommand: Contendría la lógica para eliminar un gasto. Su undo() lo reinsertaría.
 - **Invocador:** Un CommandInvoker (quizás parte de la capa de Presentación o Casos de Uso) mantendría una lista de comandos ejecutados para permitir la operación de deshacer/rehacer.
 - **En la Arquitectura Limpia:** Los comandos encapsularían operaciones de la capa de Dominio o Casos de Uso, permitiendo que

la capa de Presentación invoque acciones sin conocer los detalles de su implementación o cómo revertirlas.

Íconos Adaptables

1. Introducción a los Íconos Adaptables

Los íconos adaptables (Adaptive Icons) son un estándar de diseño introducido en Android 8.0 (API nivel 26) que permite que los íconos de las aplicaciones se ajusten dinámicamente a diferentes formas y tamaños en distintos dispositivos y launchers. Su propósito principal es mejorar la consistencia visual y la estética del sistema, ofreciendo una experiencia de usuario más unificada.

Los íconos adaptables tienen las siguientes características clave:

- **Dos Capas Principales:** Se componen de una capa en primer plano (foreground) y una capa en segundo plano (background). Ambas pueden ser imágenes vectoriales o de mapa de bits.
- **Máscaras Aplicadas por el Sistema:** El sistema Android aplica una máscara de forma (círculo, cuadrado, "squircle", etc.) a estas dos capas, adaptando el ícono a la forma predefinida del dispositivo.
- **Efectos Visuales:** Permiten efectos visuales atractivos como animaciones de movimiento sutiles y efectos de paralaje al arrastrar o mover el ícono.
- **Temas de Usuario (a partir de Android 13/API 33):** Una característica crucial es su capacidad para adaptarse a los temas de color definidos por el usuario en el sistema. Para ello, se requiere una **tercera capa monocromática** del ícono, que el sistema utiliza para aplicar el color del tema seleccionado por el usuario desde su fondo de pantalla.

2. Verificación de Adaptabilidad de Íconos en el Proyecto "Kitty"

El proyecto "Kitty", siendo una aplicación desarrollada en Flutter (basada en el esqueleto de nazarski/kitty), por defecto maneja la generación de íconos de manera que sean *parcialmente* adaptables, pero requiere configuración específica para una adaptabilidad completa y el soporte de temas.

- **Generación Estándar de Flutter (flutter_launcher_icons):**
 - Herramientas comunes en Flutter, como el paquete flutter_launcher_icons, facilitan la creación de los íconos para Android e iOS. Cuando se configura correctamente, esta herramienta genera los assets necesarios (XML para capas de foreground y background) que permiten que el ícono se adapte a las formas variadas (círculo, cuadrado, etc.) en diferentes launchers de Android.
 - **Esto cumple con la adaptabilidad de forma y efectos visuales básicos.**
- **Adaptabilidad a Temas del Usuario (Android 13+):**
 - Aquí es donde el ícono por defecto de un proyecto Flutter (o incluso Android nativo sin configuración adicional) puede no ser completamente adaptable. La documentación de Android indica que para que un ícono adopte los colores del tema del usuario (función "Iconos temáticos" en Android 13/API 33), **se debe proporcionar una versión monocromática (monochrome layer) del ícono.**
 - Los íconos generados por defecto o con configuraciones básicas a menudo solo incluyen las capas de foreground y background a color. Sin la capa monocromática específica, el ícono no se adaptará al tema de color del usuario y mantendrá su esquema de color original, lo que podría desentonar con el resto de la interfaz temática del dispositivo.

3. Recomendaciones para la Implementación de Íconos Adaptables en "Kitty"

Para asegurar que los íconos del proyecto "Kitty" sean completamente adaptables, incluyendo el soporte para los temas de usuario en Android 13 y posteriores, se recomiendan los siguientes pasos:

1. Diseño de las Tres Capas del Ícono:

- **Capa de Foreground (color):** El logo o elemento principal de la aplicación.
- **Capa de Background (color):** El fondo del ícono.
- **Capa Monocromática (para temas):** Una versión simplificada, en blanco y negro, del ícono de foreground, que el sistema Android coloreará según el tema del usuario.

2. Configuración en el Proyecto Android de Flutter:

- Para la **capa monocromática**, es necesario declarar un elemento `<monochrome>` dentro del archivo `res/mipmap-anydpi-v26/ic_launcher.xml` de tu proyecto Android. Este elemento debe apuntar a un drawable (`@drawable/ic_launcher_monochrome`) que contenga la versión monocromática de tu ícono.
- Asegúrate de que tus assets de íconos estén correctamente definidos para cada capa en los directorios `res/mipmap-anydpi-v26/` (para Android) y `Assets.xcassets` (para iOS, aunque iOS tiene su propio sistema de íconos adaptables que es diferente al de Android y no maneja temas de usuario de la misma manera).

3. Herramientas de Generación:

- Si utilizas `flutter_launcher_icons`, revisa su documentación para ver si tiene soporte específico o configuraciones para la capa monocromática. A menudo, esto implica proporcionar un asset adicional y especificarlo en tu archivo `pubspec.yaml` o en la configuración de la herramienta.

- Considera el **Image Asset Studio de Android Studio** (File > New > Image Asset en la vista de proyecto Android), que es una herramienta oficial para generar íconos adaptables y puede ayudarte a crear las capas necesarias, incluyendo la monocromática. Luego, integrarías estos assets generados en tu proyecto Flutter.