Arquitectura de Software para Aplicaciones Multiplataforma y Patrones de Diseño

Análisis de casos de estudio y discusión

Multiplataforma Patrones de Diseño Casos de Estudio

Docente: Ing. Milton Accostupa A.

Introducción a Arquitectura Multiplataforma

¿Qué es?

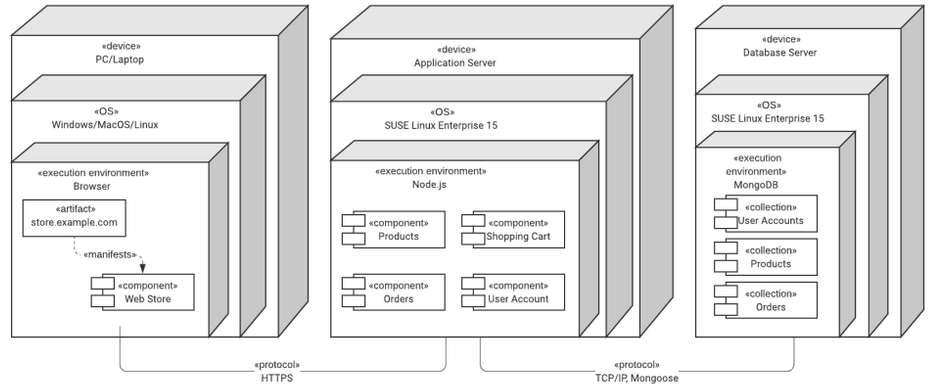
Solución estructurada para crear aplicaciones que funcionan en múltiples plataformas con una base de código compartida.

Importancia de la Separación de Capas

- Responsabilidades claras
- Testabilidad mejorada

Beneficios Clave

- **Reutilización de Código**
- **Mantenimiento Eficiente**
- **Desarrollo Acelerado**



Frameworks Multiplataforma Populares

Flutter

Lenguaje Dart
Sintaxis moderna y orientada a objetos

Widget-Based Architecture
Componentes reutilizables y composables

Rendimiento Nativo
Compilación AOT para máxima velocidad

UI Personalizada
Renderizado propio sin dependencias nativas

React Native

Lenguaje JavaScript
Ecosistema amplio y conocido

Componentes Nativos
Uso de elementos nativos de cada plataforma

Gran Comunidad
Soporte extenso y librerías disponibles

Hot Reloading
Desarrollo rápido con actualización en tiempo real

Introducción a Patrones de Diseño Arquitectónicos

¿Qué son los Patrones de Diseño?

Soluciones reutilizables y probadas a problemas comunes en el diseño de software que establecen una estructura organizativa para la aplicación.

Importancia Clave

Proporcionan una estructura que facilita el desarrollo, mantenimiento y escalabilidad de aplicaciones multiplataforma.

Objetivos Principales

Separación de Responsabilidades

Cada componente tiene una función específica y bien definida, reduciendo la complejidad y acoplamiento.

Testabilidad

Facilita la creación de pruebas unitarias y de integración al aislar componentes individuales.

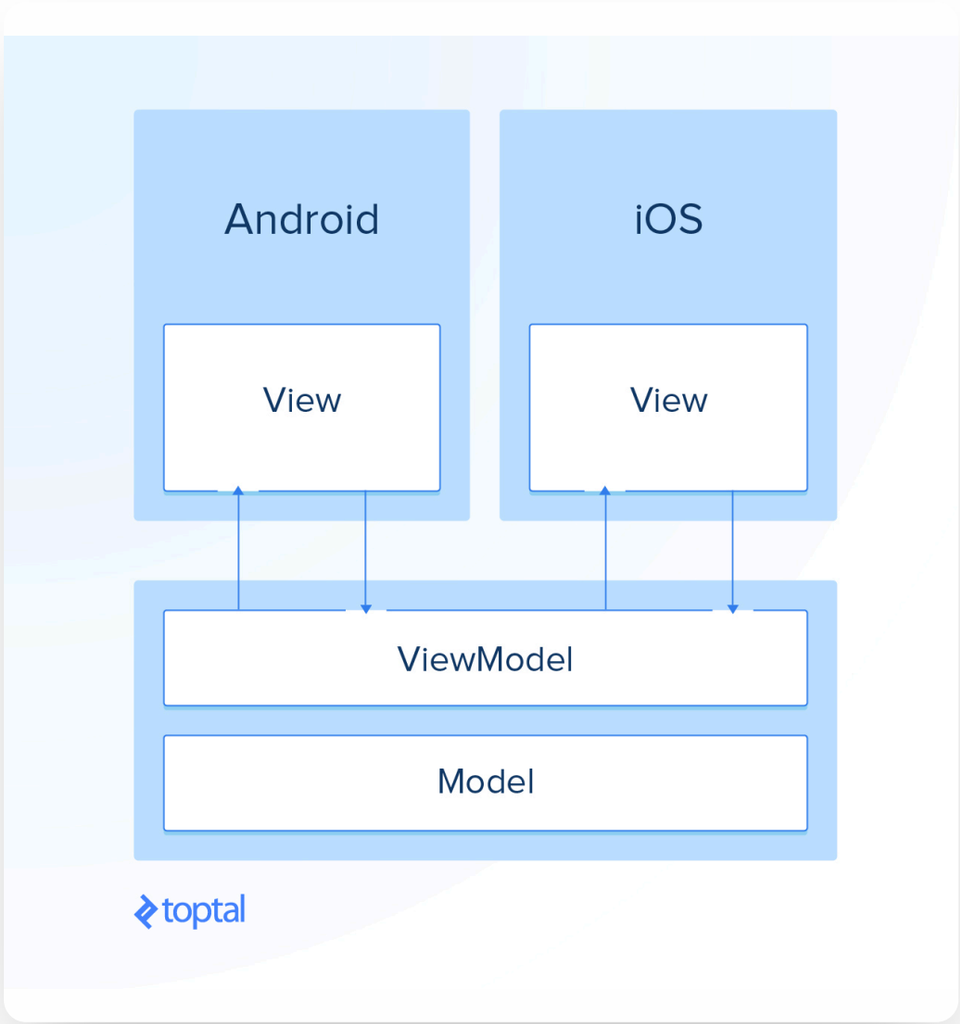
Mantenibilidad

Permite realizar cambios en una parte del sistema sin afectar otras, mejorando la gestión del código.

Escalabilidad

Permite que la aplicación crezca de manera ordenada y controlada sin comprometer su calidad.

Patrón MVVM (Model-View-ViewModel)



¿Qué es MVVM?

Patrón arquitectónico que separa la interfaz de usuario de la lógica de negocio y datos mediante un ViewModel intermediario.

Model

Datos y lógica de negocio independiente de la UI

View

Interfaz de usuario que observa el ViewModel

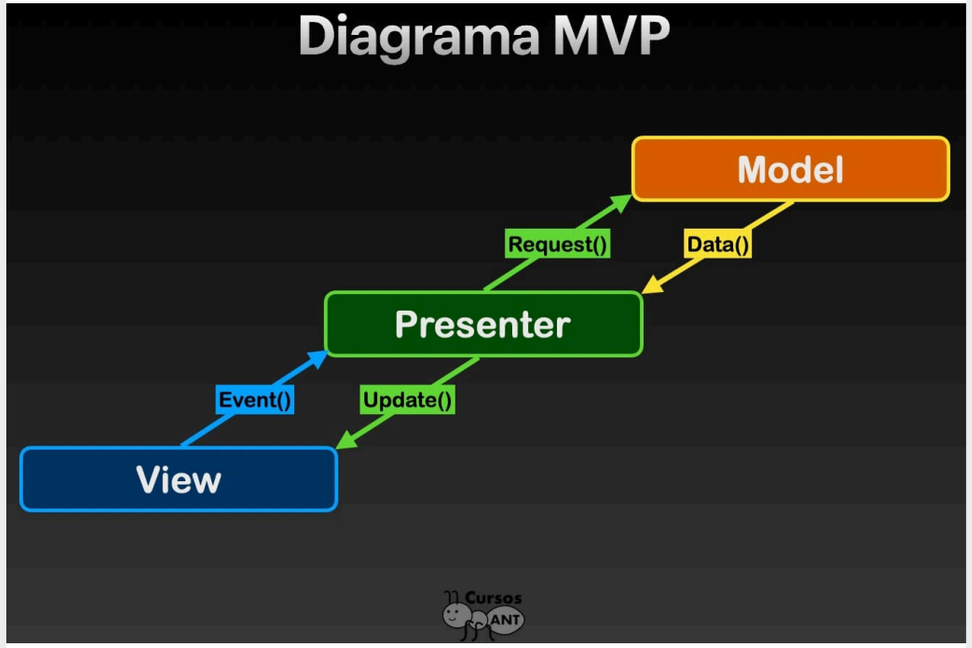
ViewModel

Mediador que expone datos y acciones a la View

Flujo Bidireccional

La View observa cambios en el ViewModel mediante data binding. El ViewModel actualiza el Model y notifica cambios.

Patrón MVP (Model-View-Presenter)



¿Qué es MVP?

Patrón arquitectónico donde el Presenter actúa como intermediario entre la View y el Model, manejando la lógica de presentación.

Model

Capa de datos y lógica de negocio

View

Interfaz de usuario que delega acciones al Presenter

Presenter

Controla la lógica y actualiza la View directamente

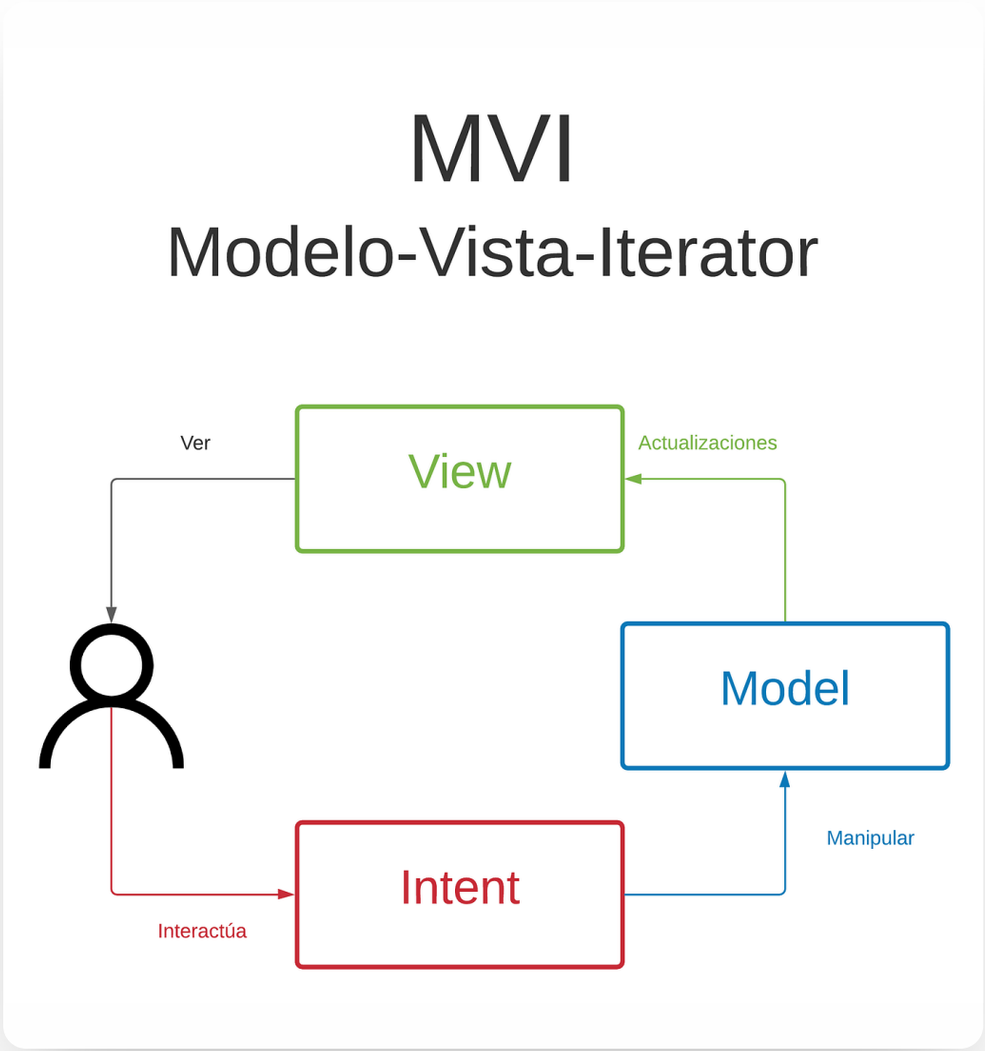
Flujo Unidireccional

View → Presenter → Model → Presenter → View

Diferencia MVVM

No hay data binding bidireccional

Patrón MVI (Model-View-Intent)



¿Qué es MVI?

Arquitectura unidireccional con flujo cíclico y estado inmutable, ideal para aplicaciones reactivas.

Model

Estado inmutable de la aplicación

View

Renderiza el estado y emite intents

Intent

Acciones del usuario que modifican el estado

Flujo Cíclico

Intent → Model → View → Intent

Predecible

Estado predecible y fácil de depurar

Comparación MVVM vs MVP vs MVI

MVVM

Model-View-ViewModel

Flujo Bidireccional
Data binding automático

Complejidad Media
Equilibrio ideal

Testabilidad Alta
ViewModels aislados

Ideal para:
Apps modernas, frameworks con binding

✓ Curva de aprendizaje moderada

MVP

Model-View-Presenter

Flujo Unidireccional
Control centralizado

Complejidad Baja
Fácil de implementar

Testabilidad Muy Alta
View completamente aislada

Ideal para:
Apps Android tradicionales, proyectos legacy

✓ Máxima testabilidad

MVI

Model-View-Intent

Flujo Cíclico Unidireccional
Estado predecible

Complejidad Alta
Curva pronunciada

Estado Inmutable
Debugging simplificado

Ideal para:
Apps reactivas, estados complejos

✓ Predecibilidad máxima

Caso de Estudio: Arquitectura en Flutter

Patrón BLoC en Flutter

Implementación del patrón **BLoC** (Business Logic Component), similar a MVI, con estado inmutable y flujo unidireccional usando Streams.

Estructura

- lib/blocs/
- lib/models/
- lib/screens/
- lib/repositories/

Implementación

- Streams para eventos
- State inmutable
- BlocBuilder para UI

Ventajas

Flujo predecible, alta testabilidad, separación clara

Desafíos

Curva de aprendizaje inicial, boilerplate adicional

Desarrollo multiplataforma con Flutter y Dart

Caso de Estudio: Arquitectura en React Native

Redux / MobX State Management

Implementación de patrones **MVVM/MVI** usando Redux para gestión de estado global con flujo unidireccional predecible.

Flujo Redux

- → UI → Dispatch Action
- → Reducer → New State
- → Store → UI Update

Arquitectura

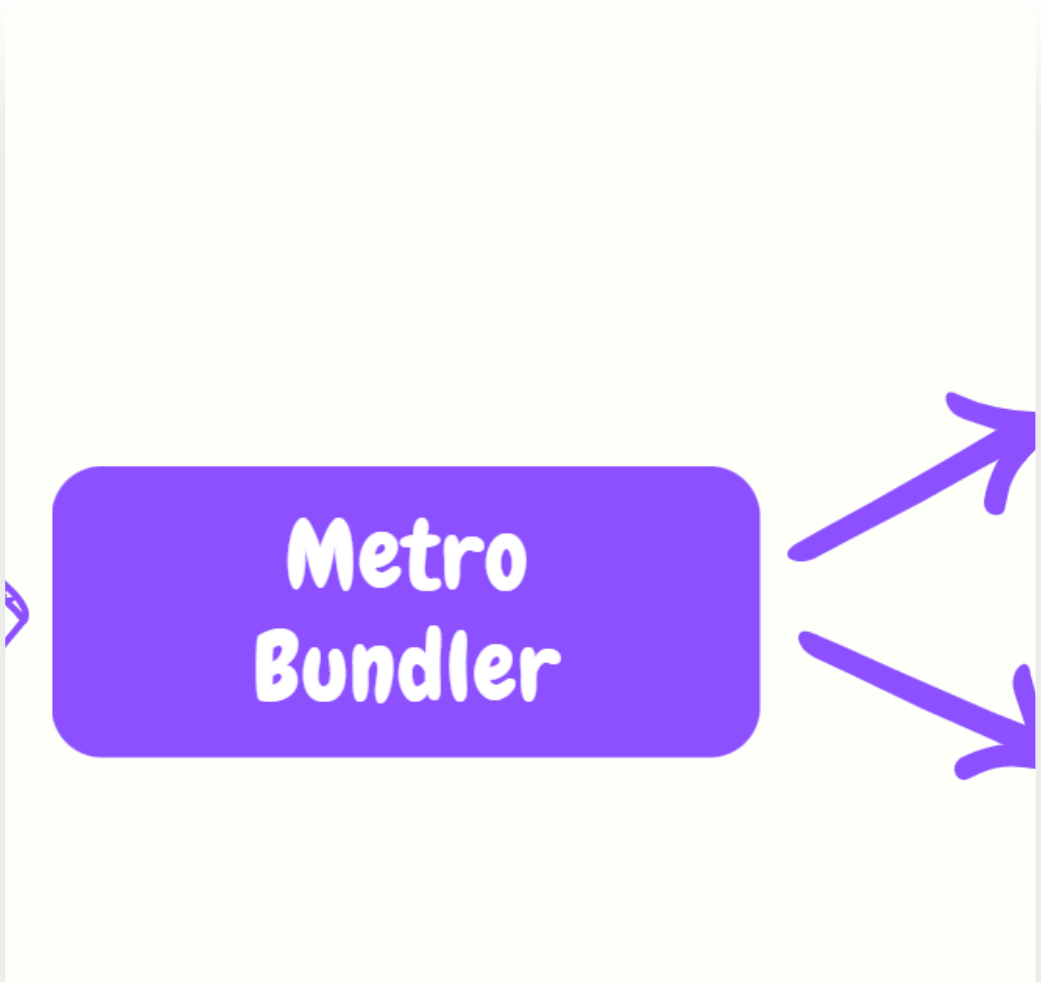
- Actions/Reducers
- Store centralizado
- Selectores optimizados

Estado Global

Store único para toda la aplicación

Time-Travel Debug

Navegación entre estados anteriores



Flujo de datos en React Native con Redux

Discusión: ¿Cuándo usar cada patrón?

MVVM

Tamaño del Proyecto
Mediano a grande

Equipo de Desarrollo
Experiencia con frameworks modernos

Testabilidad
Alta con ViewModels aislados

Complejidad de UI
Moderada con data binding

Ideal para apps modernas con frameworks que soportan data binding

MVP

Tamaño del Proyecto
Pequeño a mediano

Equipo de Desarrollo
Equipos pequeños o junior

Testabilidad
Muy alta, View completamente aislada

Complejidad de UI
Baja a moderada, sin binding

Perfecto para máxima testabilidad y proyectos legacy

MVI

Tamaño del Proyecto
Grande y complejo

Equipo de Desarrollo
Experiencia en programación reactiva

Testabilidad
Alta con estados predecibles

Complejidad de UI
Alta con estados complejos

Ideal para apps reactivas con estados inmutables

Conclusiones y Mejores Prácticas

Puntos Clave

- Elegir patrón según **necesidades del proyecto**
- Priorizar **testabilidad y mantenibilidad**
- Considerar **experiencia del equipo**
- Evaluar **complejidad de UI**

Recomendaciones Finales

Empezar simple: No sobre-diseñar desde el inicio

Refactorizar: Adaptar arquitectura según evolución del proyecto

Aprender continuamente: Los patrones evolucionan con el tiempo

Recursos Adicionales

Documentación Oficial
Flutter.dev, React Native Docs

Cursos y Tutoriales
Pluralsight, Udemy, Coursera

Artículos Técnicos
Medium, Dev.to, Android Developers

Proyectos Open Source
Ejemplos prácticos en GitHub

Comunidades
Stack Overflow, Reddit, Discord

La arquitectura correcta depende del contexto

No existe una solución universal, evalúa y adapta