icono TRAINING CONSULTING

logiRAIL

FORMACIÓN EN NUEVAS TECNOLOGÍAS

ACCIÓN FORMATIVA: IOS

Formador

David Jardón
Formador

Tengo más de 12 años de experiencia en el desarrollo de aplicaciones móviles nativas en **Android** e **iOS**.

He trabajado en múltiples proyectos para empresas como **BBVA**, **Santander**, **Telefónica**, **Bankinter**, **Mutua Madrileña**, **Ilunion**, **Adif** e **Inditex**.

Soy Co-Founder y CEO de **Dust Summit**, startup de desarrollo de software especializada en proyectos mobile y outsourcing de perfiles de desarrollo.

Programa

1. **Introducción**
    a. Información de la formación
    b. Presentaciones

2. **Conocimientos previos Swift + iOS**
    a. Conceptos básicos de Swift
    b. Conceptos previos de iOS

3. **Patrones de diseño**
    a. ¿Qué son?
    b. Principios, conceptos clave y ejemplos

4. **SOLID + CLEAN**
    a. ¿Qué es SOLID?
    b. ¿Qué es Clean?
    c. ¿Cómo aplicamos estos conceptos?

Programa

5. **MVC & MVVM**
    a. Definición y ventajas
    b. Ejemplo

6. **VIPER**
    a. Definición y ventajas
    b. Ejemplo

7. **Inyección dependencias**
    a. Conceptos clave
    b. Ejemplo aplicación completa VIPER
    c. Test Unitarios

8. **SwiftUI & UIKit**
    a. Empezar con SwiftUI
    b. Componentes SwiftUI
    c. Incluir SwiftUI en UIKit
    d. Incluir UIKit en SwiftUI

Introducción

Información formación

Información

- **36 horas** de formación en **iOS.**
- **Días 23, 24, 27, 28, 29, 30 de Noviembre y 1 de diciembre de 2023**
- Horario de **09:30 a 14:40** con dos descansos de 20 minutos.
- Activar la cámara durante toda la formación es muy recomendado.

- Usaremos **Xcode** para programar y realizaremos pruebas con iPhone y/o el simulador

- **Dudas** a través del **chat** por el canal abierto o de forma privada
- Todas las dudas privadas se resolverán de forma abierta para todos de forma anónima
- En cualquier momento durante la formación se podrán preguntar todas las dudas, pidiendo antes **turno de palabra**

¡Muy importante participar activamente durante toda la formación!

Objetivos

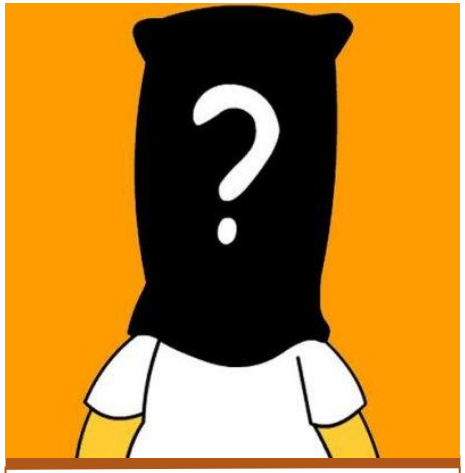
1. **Diferenciar patrones de diseño:** Conocer distintos patrones de diseño en un proyecto y cómo podemos desarrollarlos teniendo en cuenta sus ventajas y limitaciones.

2. **Implementar arquitecturas MVVM:** Saber identificar cómo distribuir y desarrollar el código de una aplicación teniendo en cuenta la metodología CLEAN.

3. **Importancia de patrones de diseño en testing:** Comprender cómo la arquitectura de un proyecto, el código, la lógica y todo el desarrollo realizado influye al realizar test unitarios.

4. **Migración de UIKit a SwiftUI:** Aprender a utilizar las herramientas disponibles para integrar SwiftUI en UIKit, UIKit en SwiftUI y facilitar la migración a proyectos con SwiftUI.

Sobre vosotros



MR. X
Alumno

Cuando sea tu turno:

1. Dí tu nombre
2. Cuenta tu experiencia previa en programación
3. Detalla tus conocimientos de iOS

4. Cuenta algo sobre tí que sea mentira y algo que sea verdad.
   ¡Tendremos que adivinar qué es mentira!

icono TRAINING CONSULTING

logiRAIL

Swift + iOS

Conocimientos previos

Previously on iOS…

En este punto deberías tener claros los siguientes conceptos:

1. **Programación funcional**
2. **Closures**
3. **Protocolos**
4. **Protocolo + Delegado**
5. **MVVM**
6. **Clean**

Previously on iOS…

PROGRAMACIÓN FUNCIONAL

- La programación funcional es un paradigma de programación que se basa en el uso de funciones puras y evita el estado compartido y los efectos secundarios.
- Swift es un lenguaje que admite la programación funcional y proporciona herramientas para trabajar con datos de manera funcional.
- Las funciones como **map, reduce, filter y flatMap** son ejemplos de funciones de alto nivel en Swift que operan en colecciones (arrays, diccionarios, etc.).

https://developer.apple.com/documentation/swift/array/map(_:)-87c4d

Previously on iOS…

CLOSURES

- Son similares a las funciones, pero pueden capturar y almacenar referencias a variables y constantes del contexto en el que fueron creados.
- Pueden **tomar parámetros y devolver valores**, o no tomar ni devolver nada.
- Los closures **se pueden pasar como argumentos a funciones**.
- Un closure se puede ejecutar después de que la función que lo ha recibido haya finalizado. Estos, deben marcarse explícitamente con ***@escaping*** en la declaración de parámetros.

https://docs.swift.org/swift-book/documentation/the-swift-programming-language/closures/

Previously on iOS…

PROTOCOLOS

- Los protocolos son una forma de **definir** una serie de **métodos, propiedades** y requisitos que un tipo (clase, estructura o enumeración) debe cumplir.
- Proporcionan un **conjunto de reglas** y una interfaz que los tipos conformantes deben seguir, es un “contrato” entre componentes.
- El uso de **protocolos permite** el **polimorfismo**, lo que significa que un objeto puede ser tratado de manera uniforme sin importar su tipo concreto.
- Los **protocolos pueden heredar** de otros protocolos.

Previously on iOS…

PROTOCOLO + DELEGADO

- El patrón del protocolo delegado es una **técnica de diseño** que **permite** que un **objeto delegue** (pase) la **responsabilidad** de ciertas tareas o eventos **a otro objeto**.
- El protocolo delegado se utiliza en escenarios donde un objeto necesita informar eventos o solicitar acciones a otro objeto.
- Permite que los **objetos se comuniquen sin acoplarlos** directamente.
- Mejora la **claridad del código y la separación de responsabilidades**.

Previously on iOS…

MVVM

- MVVM es un **patrón de diseño** que separa la lógica de presentación de la interfaz de usuario en aplicaciones iOS.
- Se compone de tres componentes principales: **Model**, **View** y **ViewModel**.
- Permite una clara **separación** de las **responsabilidades** en la aplicación.
- Facilita la **actualización** y el **mantenimiento** del **código** al reducir las dependencias.
- MVVM es especialmente útil en aplicaciones iOS que requieren una **interfaz de usuario dinámica y compleja**.

Previously on iOS…

CLEAN

- CLEAN es un principio arquitectónico que promueve la **separación de capas** en el desarrollo de aplicaciones iOS.
- Se compone de cinco componentes principales: **Capas, Reglas, Entidades, Adaptadores y Controladores**.
- CLEAN permite una clara **separación de las responsabilidades** en la aplicación.
- Definir las capas principales de la aplicación, como la **capa de presentación, la capa de lógica de negocio y la capa de datos**.

icono TRAINING CONSULTING

logiRAIL

Patrones de diseño

Conceptos y ejemplos

Patrones de diseño

¿Qué son?

- Son una solución general, reutilizable y aplicable a diferentes problemas de diseño de software.
- Se entienden como "***plantillas***" que identifican problemas en el sistema y proporcionan soluciones apropiadas a problemas generales.
- Se han creado y definido a través de la prueba y el error, facilitan la **claridad del código y la separación de responsabilidades**.

Patrones de diseño

¿Qué aportan?

- Permiten identificar condiciones de error y problemas en el código.
- Representan soluciones que funcionan ya que han sido probados por más desarrolladores.
- Facilitan mantener código de calidad al facilitar:
  - Mantenibilidad.
  - Escalabilidad.
  - Testabilidad.
  - Reusabilidad.

Patrones de diseño

¿Cuáles existen?

- **Patrones creacionales**: Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.
- **Patrones estructurales**: Explican cómo ensamblar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura.
- **Patrones de comportamiento**: Tratan con algoritmos y la asignación de responsabilidades entre objetos.

Patrones de diseño

Patrones creacionales

- **Factory Method:** Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

- **Builder:** Nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

- **Singleton:** Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Patrones de diseño

Patrones estructurales

- **Adapter:** Permite la colaboración entre objetos con interfaces incompatibles. Convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

- **Proxy:** Proporciona un sustituto para otro objeto. Un proxy controla el acceso al objeto original, permitiendo hacer algo antes o después de que la solicitud llegue al objeto original.

Patrones de diseño

Patrones comportamiento

- **Iterator:** Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, diccionario, etc…)
- **Observer:** Definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.
- **State:** Un objeto alterar su comportamiento cuando su estado interno cambia.

Patrones de diseño

En programación mobile los patrones de diseños más utilizados, con algunas variaciones, suelen ser los siguientes:

- **MVC**: Modelo Vista Controlador. Es el patrón más simple.
- **VIPER:** View, Interactor, Presenter, Entity y Router. Es el más complejo de implementar y mantener.
- **MVVM:** Modelo Vista Vista-Modelo. Es el más utilizado y sencillo de desarrollar y mantener.

Patrones de diseño

¿Cómo elegir un patrón de diseño?

- **Requisitos del proyecto:** Comprende las necesidades y limitaciones del proyecto.
- **Habilidades del equipo:** Evalúa la experiencia y familiaridad del equipo de desarrollo con los diferentes patrones arquitectónicos.
- **Escalabilidad esperada:** Mide el crecimiento previsto de la aplicación en términos de características, usuarios y rendimiento.

Patrones de diseño

¡CUIDADO!

- Los patrones intentan dar soluciones a problemas generales. Ésto es visto por muchos como un dogma y se convierten en fanáticos, implementan los patrones “al pie de la letra”, sin adaptarlos al contexto del proyecto particular.

- *“Si lo único que tienes es un martillo, todo te parecerá un clavo”*. Cuando no tienes suficientes conocimientos puedes querer aplicar patrones de diseño en cualquier situación.

¡No busques soluciones a problemas que no tienes!

icono TRAINING CONSULTING

logiRAIL

SOLID + CLEAN

Conceptos y ejemplos

SOLID

SOLID es un estándar que define ciertas reglas que deben seguirse para que el código sea legible y lógico, que se puedan crear extensiones con facilidad y que sea testable.

- Siguiendo los principios SOLID, también harás que tu código tenga una **alta cohesión y bajo acoplamiento**.

- El software tiene una alta cohesión cuando tiene **un único propósito**, y todas sus clases, métodos y propiedades definen una funcionalidad única, relacionada con una sola cosa.

- El acoplamiento en programación orientada a objetos se refiere al grado de **independencia** que tienen dos porciones de código entre sí.

SOLID

SOLID - Single Responsibility Principle

- Un objeto debería tener una, y sólo una **única razón** para **cambiar** su **estado** o valor.
- Una clase o función debe ser **responsable de una sola cosa**.
- Las clases y funciones deben tener nombres muy específicos que definan su objetivo y ser lo más pequeñas posibles.

SOLID

SOLID - Open-Closed Principle

- Una clase (o entidad de cualquier tipo) **no debería ser modificada** para añadir una nueva funcionalidad.
- Si es necesario hacer cambios estos deben añadirse como una **extensión**.
- Esto impide que el sistema se rompa al añadir **comportamiento** y que sea **estable**.

SOLID

SOLID - Liskov Substitution Principle

- Una clase que implementa una interfaz debe ser capaz de sustituir cualquier referencia que implemente esa misma interfaz.

- El código debe funcionar siempre **independientemente de** que **instancia** de clase se use, cuando estas implementan la misma interfaz.

SOLID

SOLID - Interface Segregation Principle

- Un cliente no debe depender de métodos que no usa o de métodos que no está llamando.
- **Cambiar métodos** de una clase **no** debe **afectar** a **otras clases** que no son dependientes.
- Es mejor tener muchas **interfaces pequeñas** que una grande.

SOLID

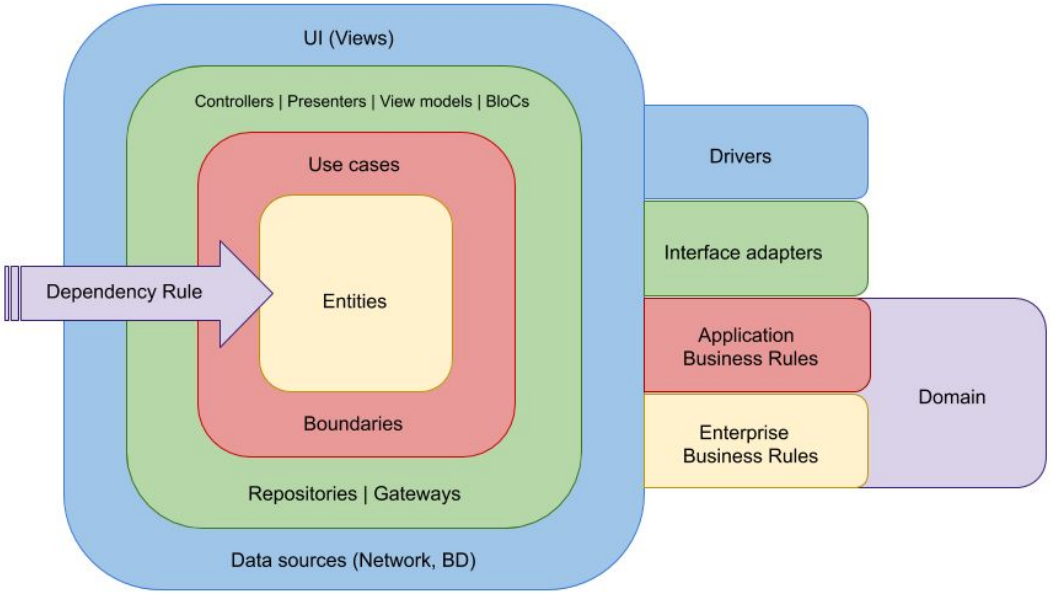
SOLID - Dependency Inversion Principle

- Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones.
- **Las abstracciones no deberían depender de los detalles**. Los detalles deben depender de abstracciones.
- **Módulos de alto nivel**: se refieren a los objetos que definen el qué es y qué hace tu programa. Aquellos que contienen la lógica de negocio y cómo interactúa.
- **Módulos de bajo nivel**: son aquellos objetos que no están directamente relacionados con la lógica de negocio del programa

CLEAN

- CLEAN es un principio arquitectónico que promueve la **separación de capas** en el desarrollo de aplicaciones iOS.
- Clean architecture es un conjunto de principios cuya finalidad principal es ocultar los detalles de implementación a la lógica de dominio de la aplicación.
- CLEAN permite una clara **separación de las responsabilidades** en la aplicación.
- Definir las capas principales de la aplicación, como la **capa de presentación, la capa de lógica de negocio y la capa de datos**.

CLEAN


UI (Views)
Controllers | Presenters | View models | BloCs
Use cases
Entities
Boundaries
Repositories | Gateways
Data sources (Network, BD)
Dependency Rule
Drivers
Interface adapters
Application Business Rules
Enterprise Business Rules
Domain

CLEAN

Se compone de cinco componentes principales: **Capas, Reglas, Entidades, Adaptadores y Controladores**.

- **Capas (Layers):** Representan la separación lógica de los componentes de la aplicación.
- **Reglas (Rules):** Definen las restricciones y normas de la aplicación.
- **Entidades (Entities) :**Contienen los datos y el estado de la aplicación.
- **Adaptadores (Adapters):** Conectan las capas y convierten datos en un formato adecuado.
- **Controladores (Controllers):** Gestionan la interacción con la interfaz de usuario y la lógica de flujo de la aplicación.

CLEAN

Tambíen podemos ver estas capas bajo la siguiente agrupación:

- **Dominio**: Entidades y casos de uso. Es el corazón de una aplicación y tiene que estar totalmente aislado de cualquier dependencia ajena a la lógica o los datos de negocio.

- **Adaptadores:** Se van a encargar de transformar la información y como es representada en los detalles de implementación o frameworks, drivers a como la entiende el dominio.

- **Detalles de implementación**: Son todos aquellos frameworks, librerías que se suelen utilizar en una aplicación para mostrar o almacenar información, por ejemplos iOS.

icono TRAINING CONSULTING

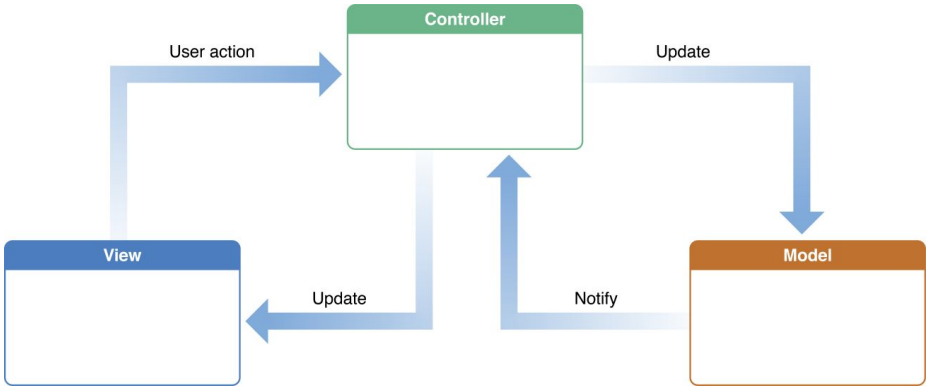
logiRAIL

MVC

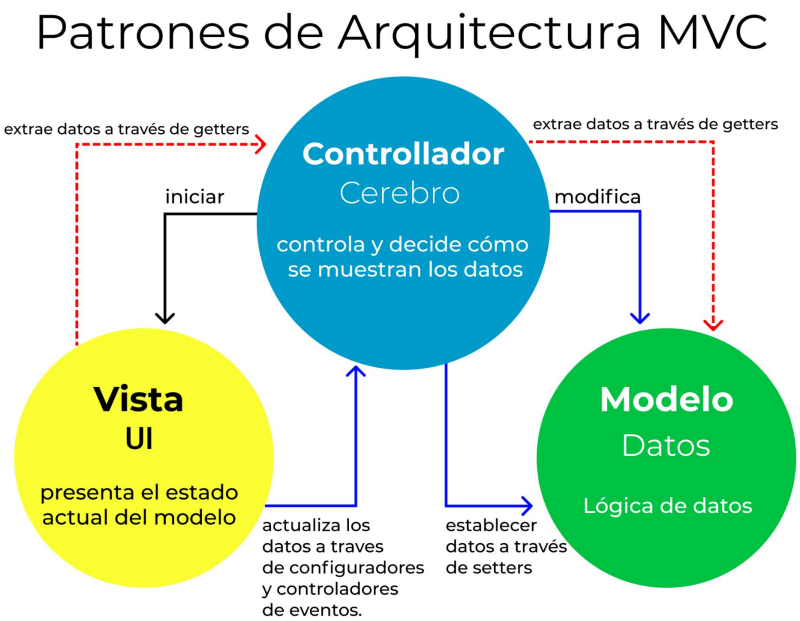
Conceptos

MVC

MVC parte de las iniciales de **Modelo-Vista-Controlador** (Model-View-Controller, en inglés), que son las capas o grupos de componentes en los que se separará el código en el proyecto.



MVC


Patrones de Arquitectura MVC
extrae datos a través de getters
Controllador
Cerebro
controla y decide cómo se muestran los datos
extrae datos a través de getters
iniciar
modifica
Vista
UI
presenta el estado actual del modelo
Modelo
Datos
Lógica de datos
actualiza los datos a traves de configuradores y controladores de eventos.
establecer datos a través de setters

MVC

MVC - Modelo

- **Encapsula** los datos de la aplicación
- Los objetos modelo no tienen conexión con los objetos de la View que presentan los datos.
- Modelos de datos **DTO** (Data Transfer Objects).
- Gestionar el **almacenamiento** y **recuperación** de **datos** del dominio.
- Contiene la representación del dominio, lógica de negocio y la persistencia de datos

MVC

MVC - Vista

- Cualquier objeto de la aplicación que el usuario pueda ver, **interfaz del usuario**.
- Un objeto View debe saber mostrarse al usuario y responder a acciones.
- Solo realiza funciones relacionadas con la interfaz de usuario.
- **No** implementa la **lógica** de negocio.
- No tiene contacto directo con la capa Model.

MVC

MVC - Controlador

- Actúa de **intermediario** entre la capa de View y la de Model.
- Recibe los cambios del modelo y de la vista.
- Realiza tareas de **configuración** y **coordinación** entre los datos y las vistas.
- Se podría decir que es el **cerebro del proyecto**, decide lo que sucederá en cada momento.

MVC

El uso del patrón MVC ofrece ventajas sobre otras maneras de desarrollar aplicaciones con interfaz de usuario.

- **Separación** de **responsabilidades** impuesta por el uso del patrón MVC.
- Proyectos más limpios, **simples**, más fácilmente **mantenibles** y más **robustos**.
- **Reutilizar** los desarrollos y asegurando **consistencia** entre ellos.
- Facilidad para realización de **pruebas unitarias**.

MVC

No todo es perfecto, su uso presenta también algunas desventajas:

- La división impuesta por el patrón MVC obliga a mantener un **mayor número de archivos**, incluso para tareas simples.
- **Curva de aprendizaje**. Dependiendo del punto de partida, el salto a MVC puede resultar un cambio radical y su adopción requerirá cierto esfuerzo.
- Peligro de generar clases con gran número de líneas de código difíciles de mantener.
- Problema de **Massive View Controller**. El ViewController tiende a aunar tanto lógica de interfaz como de negocio.

MVC

Massive View Controller

- Se incrementan las funcionalidades de un ViewController acumulando responsabilidad en el mismo.
- Los ViewController pasan de tener menos de 300 líneas de código a más de 1.000.
- Funcionalidades imposibles de mantener, de reutilizar y de testear.

icono TRAINING CONSULTING

logiRAIL

MVVM

Conceptos y Ejemplos

MVVM

MVVM es un **patrón de diseño** que separa la lógica de presentación de la interfaz de usuario en aplicaciones iOS. Se compone de tres componentes principales: **Model**, **View** y **ViewModel**.

MVVM

- Permite una clara **separación** de las **responsabilidades** en la aplicación.
- Facilita la **actualización** y el **mantenimiento** del **código** al reducir las dependencias.
- MVVM es especialmente útil en aplicaciones iOS que requieren una **interfaz de usuario dinámica y compleja**.

MVVM

Modelo (Model)

- Representa los datos y la lógica de negocio de la aplicación.
- Es independiente de la interfaz de usuario.

Vista (View)

- Muestra la interfaz de usuario y responde a las acciones del usuario.
- Debe ser lo más pasiva posible.

ViewModel

- Actúa como intermediario entre el Modelo y la Vista.
- Contiene la lógica de presentación y prepara los datos para ser mostrados en la Vista.

MVVM

MVVM - Modelo

- Representa la **capa de datos** y/o la **lógica de negocio**.
- También denominado como el **objeto del dominio**.
- El modelo contiene la información, pero nunca las acciones o servicios que la manipulan.
- En ningún caso tiene dependencia alguna con la vista.

MVVM

MVVM - View

- Representar la información a través de los elementos visuales que la componen.
- Las vistas en MVVM son activas, contienen comportamientos, eventos y enlaces a datos.
- Se puede utilizar **bindings**, es decir, vinculación automática entre los datos del modelo y los de la vista, de manera que cuando cambie alguno se modifique automáticamente el otro.

MVVM

MVVM - ViewModel

- Es un actor **intermediario** entre el **modelo** y la **vista**.
- Contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz.
- ViewModel contiene el **estado de la View** y se comunica con ella.
- La View sabe quién es su ViewModel pero el **ViewModel no sabe con qué View** se comunica.

icono TRAINING CONSULTING

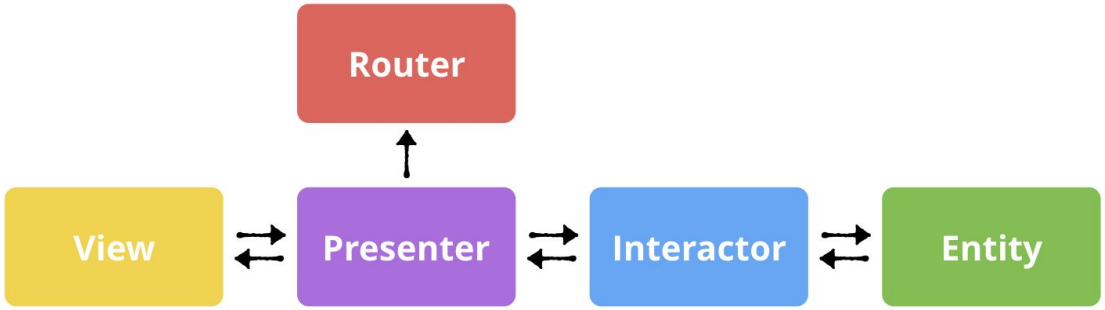
logiRAIL

VIPER

Conceptos y Ejemplos

VIPER

- VIPER: **View**, **Interactor**, **Presenter**, **Entity** y **Router**.
- Separa la lógica de visualización de la lógica del modelo de datos.
- Solo el Presenter habla con la View, y solo el Interactor habla con el Model/Entity.
- El Presenter y el Interactor se coordinan entre sí.



VIPER

- Crear interfaces más claras y mejor definidas, independientes de otros módulos.
- Hace que sea más fácil cambiar la manera en la que la interfaz presenta varios módulos de usuarios.
- Los módulos de Viper son independientes y funcionan muy bien para equipos de gran tamaño.
- Reduce el número de conflictos dentro del equipo de desarrollo.
- Hace que el codebase sea siempre parecido. Lo que hace leer el código de otras personas más rápido.

VIPER

- **View**: La responsabilidad de la vista es enviar las acciones del usuario al presentar y mostrar lo que le comunica el Presenter.
- **Presenter**: Su responsabilidad es convertir datos desde el Interactor en acciones de usuario, crear un modelo vista y llevarlo hacia el View para mostrarlo.
- **Router**: Tiene toda la lógica de navegación en el proyecto.
- **Interactor**: Es la columna vertebral del proyecto, ya que contiene la lógica de negocio. Su implementación es totalmente independiente de la interfaz del usuario.
- **Entity**: Contiene el modelo básico de objetos usados por el Interactor. Tiene parte de responsabilidades de la capa modelo en otras arquitecturas.

icono TRAINING CONSULTING

logiRAIL

Inyección Dependencias

Tests Unitarios

Inyección Dependencias

La inyección de dependencias es un patrón que permite **proporcionar** a una clase todas las **dependencias** que necesita desde el exterior, en lugar de crearlas internamente.

- Permite una mayor flexibilidad.
- Facilita las pruebas unitarias al separar las dependencias de la lógica de negocio.
- Desacoplamiento: Reduce la dependencia directa entre componentes

¡Cuidado! La inyección de dependencias con Storyboards es un poco limitada. Existen librerías para ello como por ejemplo SwinjectStoryboard

Inyección Dependencias

- En lugar de crear la dependencia internamente en la clase, la inyectamos a través del constructor.
- Esto permite utilizar diferentes implementaciones de dependencia según sea necesario.
- Lo ideal es utilizar protocolos, como hemos visto en patrones de diseño.

```
class MiClase {
    private let dependencia: MiDependencia

    init(dependencia: MiDependencia) {
        self.dependencia = dependencia
    }

    // Métodos que utilizan la dependencia
}
```

Inyección Dependencias

- Los patrones de diseño y la inyección de dependencias mejoran la calidad, mantenibilidad y escalabilidad del código.
- Los patrones de diseño proporcionan soluciones probadas a problemas comunes.
- La inyección de dependencias reduce el acoplamiento y facilita las pruebas unitarias.

Test Unitarios

- Las pruebas pueden clasificarse en tres categorías principales: **pruebas unitarias, pruebas de integración y pruebas funcionales**.
- Las pruebas funcionales evalúan si la aplicación cumple con las especificaciones funcionales y se enfocan en la experiencia del usuario.
- Para realizar pruebas funcionales, podemos utilizar **frameworks como XCTest (iOS)**.
- En **XCTest**, se puede configurar un **simulador de iOS** para ejecutar pruebas en una aplicación.

https://developer.apple.com/documentation/xctest/defining_test_cases_and_test_methods

Test Unitarios

- Una "**prueba unitaria**" es un método de prueba de software que inicializa una pequeña parte de nuestro proyecto y verifica su comportamiento de forma independiente a otras partes.

- La mayoría de las pruebas unitarias comunes constan de **tres fases**:
  - Primero, se inicializa una pequeña parte del código que se desea probar (también conocida como el "sistema bajo prueba", o SUT) .
  - Se activa el sistema bajo prueba (generalmente llamando a un método).
  - Finalmente, se observa el comportamiento resultante. Si el comportamiento observado es coherente con las expectativas, la prueba unitaria pasa, de lo contrario, falla, lo que indica que hay un problema en algún lugar del sistema bajo prueba.

- Estas tres fases de prueba unitaria también son conocidas como **Arreglar (Arrange), Actuar (Act) y Afirmar (Assert), o simplemente AAA**.

Test Unitarios

- La **cobertura de código** mide la cantidad de código fuente que es ejecutada por las pruebas (Test).
- Proporciona una métrica de qué tan bien se están probando todas las partes del código y como mínimo debería ser de un **80% del código** para ser representativo.
- Para medir la cobertura de código, utilizamos **herramientas como Xcode Code Coverage** o plugins de terceros.
- En Xcode, se puede generar un informe de cobertura de código después de ejecutar pruebas unitarias.

Test Unitarios

- **Ejemplo:** Definición de una clase de test con las funciones *setUp* y *tearDown* para configurar el test que se va a ejecutar.

```swift
import XCTest

class MiAppTests: XCTestCase {

    // Esta función se ejecuta antes de cada prueba
    override func setUp() {
        super.setUp()
        // Configurar cualquier estado necesario antes de las pruebas
    }

    // Esta función se ejecuta después de cada prueba
    override func tearDown() {
        // Limpiar o restablecer cualquier estado necesario después de las pruebas
        super.tearDown()
    }
}
```

Test Unitarios

- **Ejemplo:** Definición de una clase de test con las funciones *setUp* y *tearDown* para configurar el test que se va a ejecutar y con el test de una función sumar con ***testSumar***.

```swift
// Ejemplo de prueba para la función de suma
func testSumar() {
    // Arrange (preparar)
    let num1 = 5
    let num2 = 7

    // Act (actuar)
    let resultado = sumar(num1, num2)

    // Assert (afirmar)
    XCTAssertEqual(resultado, 12, "La suma debe ser igual a 12")
}

// Función de ejemplo que suma dos números
func sumar(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

Test Unitarios

Ventajas de las pruebas unitarias:

- Las pruebas unitarias reducen o eliminan posibles errores en nuevas y existentes características.
- Las pruebas unitarias te obligan a planificar antes de programar.
- Las pruebas unitarias actúan como un mecanismo de seguridad contra realizar un cambio en una parte del código y no saber qué va a romperse, fallar o duplicarse, entre otras cosas.

Desventajas de las pruebas unitarias:

- Aumentan la cantidad de código que necesita ser escrito.
- No hay forma de probar cada camino lógico o encontrar defectos de integración y sistema.
- Aumentarán el tiempo de desarrollo.

Test Unitarios

- **Ejemplo:** Utilizamos **XCTestExpectation** para esperar a que la llamada de red se complete antes de evaluar el resultado.

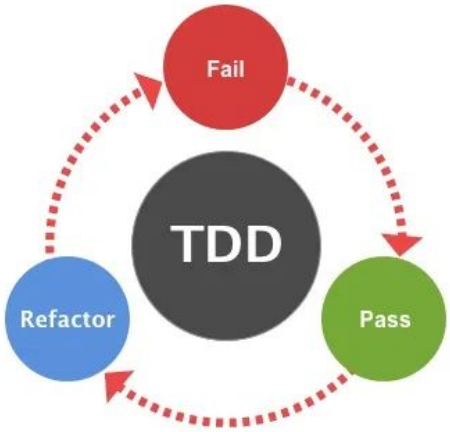
```swift
// Ejemplo de prueba asíncrona
func testAsyncCall() {
    let expectation = XCTestExpectation(description: "Llamada asíncrona completada")

    // Supongamos que aquí realizamos una llamada asíncrona
    SampleClass.callAsync { (respuesta) in
        // En este cierre, evaluamos la respuesta
        if respuesta == "Success" {
            // Si la respuesta es "Éxito", la prueba pasa
            expectation.fulfill()
        } else {
            // Si la respuesta es diferente, la prueba falla
            XCTFail("La llamada asíncrona no tuvo éxito")
        }
    }

    // Esperamos 5 segundos para que se complete la llamada
    // asíncrona (ajustar este tiempo según sea necesario)
    wait(for: [expectation], timeout: 5.0)
}
```

Test Unitarios

TDD

- En una **Prueba Unitaria, escribes código para probar tu código**. Primero escribes el código y luego escribes pruebas para verificar tu lógica y el código que escribiste.

- En el Desarrollo Guiado por Pruebas **(TDD), primero escribes pruebas que fallan y luego escribes código** para que las pruebas pasen, y luego refactorizas tu código.



Test Unitarios

- Escribir pruebas automatizadas a menudo se percibe como un trabajo no real y aburrido en comparación con la construcción de nuevas características.

- Sin embargo, el TDD convierte las pruebas en una actividad de diseño. Utilizamos nuestras pruebas para asegurarnos de que nuestro código esté haciendo lo que queremos que haga.

- También mantiene el código lo más simple posible para que sea más fácil de entender y modificar, especialmente porque los desarrolladores pasan más tiempo leyendo código.

icono TRAINING CONSULTING

logiRAIL

SwiftUI

SwiftUI & UIKit

SwiftUI & UIKit

UIKit vs SwiftUI

- **UIKit:** El marco de desarrollo de interfaz de usuario tradicional para aplicaciones iOS, ampliamente utilizado desde el lanzamiento del iPhone.

- **SwiftUI:** Un marco de desarrollo de interfaz de usuario declarativo e innovador introducido por Apple en iOS 13. Es necesario menos código para lograr el mismo resultado que con UIKit.

SwiftUI & UIKit

¿Cómo utilizamos SwiftUI con UIKit?

- Importar el módulo '***import SwiftUI***' donde sea necesario.
- Crear un objeto ***'UIHostingController'*** para alojar vistas de SwiftUI.
- Configurar y presentar la vista SwiftUI dentro de la interfaz UIKit.

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("¡Hola desde SwiftUI!")
    }
}

let hostingController = UIHostingController(rootView: ContentView())
navigationController.pushViewController(hostingController, animated: true)
```

SwiftUI & UIKit

- También podríamos crear una **subclase** de **UIHostingController** para controlar diferentes aspectos del ciclo de vida de la pantalla de UIKit.

- Nos daría acceso a la propiedad **rootView**, que será del tipo View indicado, teniendo a nuestra disposición todos las variables y funciones que sean accesibles.

¡Cuidado! Hay que tener especial atención porque no todos los métodos del ciclo de vida de UIKit son ejecutados como cuando utilizamos UIKit directamente.

https://developer.apple.com/documentation/swiftui/uihostingcontroller

SwiftUI & UIKit

- **¿Cómo crear una subclase UIHostingController?**

```swift
import UIKit
import SwiftUI

// Vista de SwiftUI
struct SwiftUIView: View {
    let text: String = "Hello, World!"

    var body: some View {
        Text(text)
    }

    func doSomething() {
        print("Do something")
    }
}
```

```swift
// Puedes utilizar esta clase en UIKit
class ContentViewController: UIHostingController<SwiftUIView> {
    required init() {
        super.init(rootView: SwiftUIView())
    }

    // Se utiliza por el Storyboard init
    @objc required dynamic init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder, rootView: SwiftUIView())
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        rootView.doSomething()
        print(rootView.text)
    }
}
```

SwiftUI & UIKit

- **¿Podemos comunicar SwiftUI con UIKit?**
- En la integración de SwiftUI en proyectos UIKit, es fundamental establecer una comunicación efectiva entre las vistas y controladores UIKit y las vistas SwiftUI.
- Para lograrlo, utilizamos propiedades como '@Binding', '@State', y 'ObservableObject' para compartir datos y acciones.

SwiftUI & UIKit

¿Cómo utilizamos UIKit en SwiftUI?

- Utilizar '**UIViewRepresentable**' para crear una vista representable de UIKit en SwiftUI.
- Configurar y personalizar la vista UIKit en el método '**makeUIView**'.
- Podemos establecer comunicación bidireccional entre vistas SwiftUI y vistas UIKit.
- Utilizar '**@Binding**', delegados o notificaciones para intercambiar datos y eventos.

SwiftUI & UIKit

- **¿Cómo utilizamos UIKit en SwiftUI?**

```swift
import SwiftUI
import UIKit

struct MyUIKitView: UIViewRepresentable {
    func makeUIView(context: Context) -> MyCustomView {
        return MyCustomView()
    }

    func updateUIView(_ uiView: MyCustomView, context: Context) {
        // Actualizar la vista UIKit
    }
}
```

Conclusiones

Dudas

Conclusión

- Durante esta formación, hemos explorado cómo utilizar **Patrones de diseños** para desarrollar la arquitectura de una aplicación iOS.

- También hemos aprendido los conceptos clave de **SOLID** y **CLEAN** para comprender cómo podemos aplicarlos de manera efectiva en el desarrollo de software.

- Estos conocimientos son esenciales para lograr un desarrollo robusto, mantenible, evolucionable y consistente en nuestras aplicaciones.

Conclusión

- Hemos profundizado en el importante tema de elegir un **patrón de diseño adecuado** en nuestras aplicaciones iOS.
- **MVVM** es un patrón de diseño extendido y habitual en la mayoría de proyectos, pero no por ello debemos elegirlo sin plantearnos nuestras necesidades concretas.
- Además, hemos entendido los conceptos clave de **MVC** y **VIPER**, que nos permiten desarrollar proyectos de manera eficiente y estructurada.
- No debemos olvidar las **ventajas y limitaciones** de cada uno de ellos antes de comenzar el desarrollo de un proyecto.

Conclusión

- Por último, hemos explorado la integración de **MVVM** con **Inyección de Dependencias** para crear aplicaciones testeables en iOS.

- Además, hemos abordado el tema del **testing en el desarrollo de aplicaciones**, cubriendo aspectos como la cobertura de código y el testing asíncrono aplicado a un proyecto con arquitectura MVVM.

- También hemos visto cómo realizar **migraciones** de **UIKit** a **SwiftUI.** Pudiendo integrar código de **SwiftUI en UIKit** y de **UIKit en SwiftUI** permitiendo desarrollar nuevas aplicaciones reutilizando componentes o migrar aplicaciones existentes poco a poco.

¡Muchas Gracias!

David Jardón
Formador

Espero que hayáis disfrutado de esta formación en iOS y que, juntos, podamos crear una **comunidad de desarrolladores** detallistas y comprometidos con las buenas prácticas de desarrollo, consiguiendo la creación de experiencias tecnológicas que mejoren y cambien el mundo.

¡Enhorabuena un paso más en tu camino como desarrollador mobile!

Gracias por vuestra participación

¡Seguimos en contacto!

www.iconotc.com