

**Centro Escolar del Lago**

COLEGIO DE CIENCIAS Y HUMANIDADES CEL

TESINA

**“Creación de una interfaz gráfica con SwiftUI en  
Xcode 11.4”**

PRESENTADO POR

ALEJANDRO D. OSORNIO LÓPEZ

MATERIA

TALLER DE LECTURA, REDACCIÓN E INICIACIÓN A LA  
INVESTIGACIÓN DOCUMENTAL II

MESTRA OLGA ANDREA GONZÁLES CAMPOS

13 / MAYO / 2020

Gracias a “Dios” que existe el “Open Source”

# Índice

Introducción .....	4
Capítulo I : Programación orientada a objetos .....	7
1.1 Objeto.....	9
1.2 Clase, instancia y propiedad .....	9
1.4 Sintaxis del punto y método .....	12
1.5 Herencia y polimorfismo.....	14
Capítulo II : SwiftUI .....	17
2.1 ¿Qué es SwiftUI? .....	18
2.2 SwiftUI y Storyboards.....	18
2.3 Declarativo y automático .....	21
2.3.1 Declarativo .....	21
2.3.2 Automatico .....	22
2.4 Composición y consistencia .....	23
2.4.1 Composición .....	23
2.4.2 Consistente .....	26
2.5 Declaración de vistas .....	27
Capítulo III : Development .....	28
3.1 Planeación .....	29
3.2 Programación .....	34
Conclusión.....	36
Referencias .....	37
Matriz de congruencia .....	40

# Introducción

De una forma declarativa y automática, SwiftUI es el nuevo framework de Apple, que brinda a los desarrolladores una alternativa para el diseño de interfaces gráficas de aplicaciones para dispositivos de Apple, como lo son el iPhone, el iPad, la Mac, el Apple Watch y el AppleTV.

Cambiando cómo funcionan y se estructuran las aplicaciones realizadas con Storyboards, para programar una interfaz gráfica con SwiftUI es necesario entender la programación orientada a objetos, ya que el programa, su estructura y función, se enfoca en vistas, es decir, objetos.

Éstos objetos, que actúan como vistas, cuentan con características o atributos, funciones o métodos y al relacionarse entre ellos forman un todo, en éste caso una interfaz y con ello una aplicación. Es necesario una introducción a la programación orientada a objetos y así poder llevar a cabo el desarrollo de una interfaz con SwiftUI.

Se infiere que el lector conoce sobre programación, sabe cómo programar con Swift, conoce frameworks de Apple como UserDefaults y sabe cómo llevar a cabo el desarrollo de aplicaciones en Xcode con interfaces gráficas realizadas con Storyboards. Pero desconoce o conoce poco sobre la programación orientada a objetos con Swift 5 y tiene nula o poca experiencia en el desarrollo de aplicaciones con interfaces gráficas realizadas con SwiftUI.

Todo lo relacionado con programación en el documento está escrito con Swift 5 y también cuenta con diagramas de flujo e imágenes con mapas conceptuales, ejemplificaciones y diseños de interfaces.

Se inicia el escrito con una introducción a la programación orientada a objetos explicando los aspectos más esenciales de la misma. Después, se pasa a la revisión de SwiftUI y se aplican conceptos sobre la programación orientada a objetos para entender cómo funciona SwiftUI. Por último se planifica desde cero el concepto de una interfaz para después desarrollarse en su totalidad.

De forma específica, el objetivo general de éste escrito es el desarrollo de una interfaz gráfica haciendo uso de SwiftUI en Xcode 11.4 para dispositivos iPhone y iPad con iOS 13.4.

Para lograrlo, tomando en cuenta que el lector desconoce sobre la programación orientada a objetos, es necesario entender los conceptos básicos sobre la misma. Durante capítulo 1 se revisa qué es un objeto, cuales son las características de éste, qué es un método, cómo un objeto se puede derivar de otro (herencia) y cómo se puede clasificar a varios objetos como de un tipo (polimorfismo), qué es una clase y cual es su diferencia a un objeto, el significado de instancia y la sintaxis de punto para interactuar con clases.

Durante el capítulo 2, se hace la revisión de qué es SwiftUI como lo define Apple, se revisan y justifican las cualidades que indican en sus definiciones sobre el framework

explicando con esto cómo funciona y que ventajas provee en relación con Storyboards. De igual forma se compara la estructura de un proyecto realizado con SwiftUI contra un proyecto realizado con Storyboards, se explica cómo funcionan los Stacks o colecciones de vistas, qué es @State, su efecto en la interfaz gráfica y cómo se componen vistas (o interfaces gráficas) complejas con el uso de colecciones de sub-vistas que forman un todo y cómo hacer una declaración básica de una interfaz con SwiftUI

Finalmente, durante el capítulo 3 se planifica el concepto de la interfaz a desarrollar, explicando el por qué de cada elemento que será añadido a la vista final, se delimitan las funciones que tendrá la vista y se idealiza la estructura que se seguirá para lograr la misma. Para la realización de la interfaz se usará UserDefaults pero no se explicará su uso para asignarlo a variables, ya que se toma en cuenta que el lector ya conoce sobre éste framework. Una vez completada la planeación y explicadas las funciones de la vista final, se pasa todo lo planificado a código haciendo uso de Xcode 11.4 con Swift 5 y SwiftUI, el código no se incluye en éste trabajo pero se puede consultar en la conclusión junto con el resultado final de la vista.

Para la planificación de la interfaz gráfica se utiliza pseudo código reservando el código con SwiftUI para la revisión individual del lector en un repositorio de GitHub, donde ya conociendo sobre SwiftUI, su estructura y sobre la programación a objetos desglosar el código es una tarea sencilla que al mismo tiempo impulsa la comprensión del mismo al lector. Se puede consultar el repositorio de GitHub y el resultado del desarrollo de la interfaz en la conclusión del documento.

# Capítulo I : Programación orientada a objetos

En la programación existen distintos tipos de paradigmas, por ello es necesario conocer el significado de ésta palabra y entender qué es el paradigma de programación orientada a objetos, para así poder utilizar SwiftUI.

“Paradigma: un modelo o patrón en cualquier disciplina científica.” (*EcuRed*, 2019)

Según Wikipedia Swift es un lenguaje multi-paradigma, lo que quiere decir que puede adaptarse a distintos modelos, entre éstos modelos se encuentra la programación orientada a objetos.

Raúl Duque (2019):

“La Programación Orientada a Objetos (POO u OOP según sus siglas en inglés) es un paradigma de programación en el que los conceptos del mundo real, relevantes para nuestro problema, se modelan a través de clases y objetos, y en el que nuestro programa consiste en una serie de interacciones entre estos objetos.” (p. 41)

Stefan Kaczmarek, brinda una definición alternativa de la programación orientada a objetos:

“Una colección de objetos en un programa. Se realizan acciones en estos objetos” (Kaczmarek, Stefan, 2019:27)

## 1.1 Objeto

SwiftUI enfoca la estructura del código en objetos para lograr la construcción de una interfaz. Éste considera todos los elementos de una interfaz como objetos individuales.

“Un objeto es cualquier cosa sobre la que se pueda actuar. Por ejemplo, un avión, una persona o una pantalla / vista en un iPad pueden ser objetos” (Kaczmarek, Stefan, 2019:27).

## 1.2 Clase, instancia y propiedad

A las características y funciones con las que cuenta un objeto, en programación, se les llaman propiedades y métodos respectivamente como dice Kaczmarek.

Stefan Kaczmarek (2019):

“Los objetos tienen propiedades y métodos. Las propiedades describen ciertas cosas sobre un objeto, como la ubicación, el color o el nombre [...] Los objetos también tienen comandos que el programador puede usar para controlarlos. Los comandos se llaman métodos. Los métodos son la forma en que otros objetos interactúan con un determinado objeto.” (p. 76)

En programación a los objetos en sí se les llama instancia. Éstos se forman a partir de la derivación de una clase. Al hablar de objetos y de clases, se habla de conceptos distintos.

“Declaraciones de objetos, también se podrían definir como abstracciones de objetos. Esto quiere decir que la definición de un objeto es la clase. Cuando se programa un objeto y se define sus características y funcionalidades, en realidad, lo que se está haciendo es programar una clase.” (*DesarrolloWeb*, 2019)

Una clase es la declaración de un objeto, define las características que tiene el objeto. Un objeto se deriva de una clase, lo que lo vuelve una instancia.

“Una clase es como la definición de un objeto, pero no es el objeto en sí, del modo como una idea no es una cosa física [...] necesitaremos convertir esa idea en algo, en un objeto real; a ese objeto lo llamamos instancia.” (*The Fricky*, 2008)

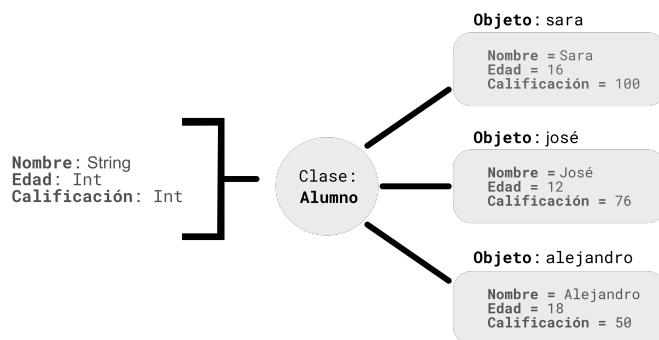


Imagen 1.1 Ejemplo de objetos, productos de la instancia de una clase

En la imagen 1.1 se puede observar la clase **Alumno**, la cual tiene como atributos nombre, edad y calificación, los cuales se aprecian en la llave que abre hacia la izquierda. A partir de ésta clase se crean 3 instancias distintas, cada una con sus propios atributos.

En Swift el código para definir la clase de la imagen 1.1 es el siguiente:

```
class Alumno : NSObject {  
    var nombre : String  
    var edad : Int  
    var calificación : Int  
}
```

Lo que se logra con éstas es solo la definición de una clase que ofrece la posibilidad de la creación de un objeto. La clase contiene el nombre de los atributos y el tipo de dato que es cada atributo.

Es necesario inicializar una clase para que contengan un valor sus variables, si es que éstas no cuentan con un valor inicial. Se amplía la clase con una función init que asignará un valor inicial a todos los atributos de la misma.

```
class Alumno: NSObject {  
    var nombre : String; var edad : Int; var calificación : Int  
    init (nombre: String, edad: Int, calificación: Int) {  
        self.nombre = nombre  
        self.calificación = calificación  
        self.edad = edad  
    }  
}
```

Init recibe los parámetros nombre, edad y calificación y los asigna a la instancia misma.

```
var sara = Alumno(nombre: "Sara", edad: 16, calificación: 100)
```

En las líneas de código anteriores se crea la instancia sara y se pasan los parámetros que serán asignados como atributos.

Otro ejemplo de instancia son las siguientes líneas de código. Se crea la instancia alejandro que tiene como atributo nombre “Alejandro”, el número 18 como edad y el número 50 como calificación.

```
var alejandro = Alumno(nombre: "Alejandro", edad: 18, calificación: 50)
```

La clase es solo un molde que después se puede usar para crear un objeto (instancia).

## 1.4 Sintaxis del punto y método

El siguiente concepto a revisar de la programación orientada a objetos es el método y con ello la sintaxis del punto.

“Los métodos describen el comportamiento de los objetos de una clase. Estos representan las operaciones que se pueden realizar con los objetos de la clase,

La ejecución de un método puede conducir a cambiar el estado del objeto.” (Covantec, 2018)

Se puede hacer uso de métodos para cambiar las propiedades de las instancias.

Ejemplo, se puede cambiar la propiedad nombre del objeto sara de “Sara” a “Pepe”

```
sara.nombre = "Pepe"
```

En el código se escribe la instancia y se agrega un punto, a esto se le llama “dot syntax” o sintaxis del punto.

“En la sintaxis de puntos, escribe el nombre de la propiedad inmediatamente después del nombre de la instancia, separado por un punto (.), sin espacios” (*The Swift Programming Language*, 2020)

Con esto se puede accesar a las propiedades y métodos, ésto es util con vistas declaradas con SwiftUI, dónde se pueden cambiar sus atributos.

Así como se puede utilizar la sintaxis del punto para acceder a propiedades, se puede utilizar para acceder a los métodos que contenga una clase.

```
class Alumno: NSObject {  
    var nombre : String; var edad : Int; var calificación : Int  
    init (nombre: String, edad: Int, calificación: Int) {...}  
    func estudiar () {  
        self.calificación += 10  
    }  
    var alumno = Alumno(nombre: "Toño", edad: 19, calificación: 90)  
    alumno.estudiar()
```

En las líneas de código anteriores se agrega la función estudiar a la clase Alumno que declaramos antes. La función suma + 10 a la propiedad calificación de la instancia. Se declara la instancia alumno y se cambia la propiedad calificación de 90 a 100 ejecutando el método estudiar con ayuda de la sintaxis del punto.

## 1.5 Herencia y polimorfismo

Las clases pueden formarse de otras clases, usar otra clase como parte de sí misma, a esto se le llama herencia.

“La herencia permite que una clase base herede los estados y comportamientos que conforman a una clase padre” (*Kodigo Swift*, 2020)

```
class Tele : NSObject {  
    var tamaño : Int  
    var marca : String  
    var prendida : Bool = false  
  
    func prenderApagar() {  
        self.prendida.toggle()  
    }  
    init(tamaño : Int, marca : String) {  
        self.tamaño = tamaño  
    }  
}
```

```
    self.marca = marca  
}  
}  
  
class TeleCurva : Tele {  
    var curavatura : Bool = true  
}
```

En el ejemplo se describen dos clases distintas, sin embargo, la clase TeleCurva es de tipo Tele, lo que quiere decir que hereda todos los atributos y métodos que describe la clase Tele.

Si se instancia la clase TeleCurva, el objeto resultante tendrá la propiedad curvatura pero al ser derivada de la clase Tele también contará con todos las propiedades y funciones de ésta.

Al heredar las propiedades y métodos, las sub-clases pueden ser tratadas como el mismo tipo de dato. A esto se le conoce como polimorfismo.

“El polimorfismo brinda la posibilidad de usar varios objetos a través de una misma interfaz mientras que cada uno de estos se comportan de forma única.” (*Kodigo Swift*, 2020)

Al existir la posibilidad de polimorfismo que describe Herrera en *Kodigo Swift*, se pueden usar todas las clases heredadas de Tele como datos de tipo Tele.

```

class Tele : NSObject {...} //Clase anteriormente descrita

class TeleCurva : Tele {...} //Clase anteriormente descrita

class Tele3D : Tele { var lentes3D : Bool = true }

class TeleSmart : Tele { var internetDisponible : Bool = true }

var Tele1 : Tele = Tele (tamaño = "10", marca = "LG")

var Tele2 : TeleCurva = TeleCurva (tamaño = "15", marca = "Sony")

var Tele3 : Tele3D = Tele3D (tamaño = "10", marca = "Panasonic")

var Tele4 : TeleSmart = TeleSmart (tamaño = "11", marca = "LG")

```

<b>Nombre variable (instancia)</b>	<b>Variable con dato de tipo</b>	<b>Clase de tipo</b>
Tele1	Tele	NSObject
Tele2	TeleCurva	Tele
Tele3	Tele3D	Tele
Tele4	TeleSmart	Tele

**Tabla 1.4 Ejemplo Polimorfismo**

Se observa que ninguna de las instancias creadas son del mismo tipo de clase. Al ser todas derivadas de una misma clase madre, se puede considerar a todas como un mismo tipo de clase.

Debido a esto se pueden realizar acciones sobre las cuatro, siempre y cuando exista la misma propiedad / método en todas, es decir, que son métodos / propiedades de la clase madre y por ende de todas las derivadas.

## **Capitulo II : SwiftUI**

## 2.1 ¿Qué es SwiftUI?

SwiftUI tiene el eslogan: “Mejores Apps. Menos Código”. La frase tiene sentido porque la realización de interfaces gráficas complejas es mas rápido en comparación con las interfaces realizadas con Storyboard.

“SwiftUI es una nueva forma de crear interfaces de usuario. Cree interfaces de usuario para cualquier dispositivo. Con una sintaxis declarativa de Swift, SwiftUI funciona con las nuevas herramientas de diseño de Xcode para mantener su código y diseño sincronizados. La compatibilidad automática con el tipo dinámico, el modo oscuro, la localización y la accesibilidad significa que su primera línea de código SwiftUI ya es el código de interfaz de usuario más potente que jamás haya escrito.” (*Apple Developer Program, 2020*)

Cuando Apple se refiere a que una interfaz realizada con SwiftUI es la “más potente que jamás haya escrito”, hace una comparación implícita con UIKit, otro framework de Apple que permite la creación de interfaces gráficas con código, y con Storyboards. Afirmando que la automatización y declaración con la que cuenta SwiftUI supera, con el resultado final, cualquier interfaz realizada con UIKit y Storyboards.

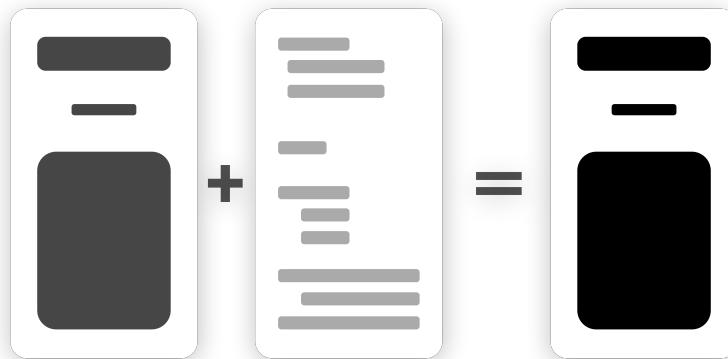
## 2.2 SwiftUI y Storyboards

Para comprobar ésta afirmación de Apple, se necesita comprobar la estructura y función de un proyecto con Storyboards, un proyecto con SwiftUI y analizar las ventajas que tiene SwiftUI respecto con la manera de funcionar de Storyboards.

Jayant Varma (2019):

“El desarrollo tradicional se centraba más en cómo crear elementos y cómo mostrarlos en la pantalla y luego continuar actualizándolos a medida que cambian los datos. Con una interfaz de usuario declarativa, se permite que el desarrollador se centre solo en lo que desea mostrar.” (p. 3)

Ésta cita se puede comprobar analizando la estructura de un proyecto con Storyboards.



**Imagen 1.3** Estructura de un proyecto con Storyboard.

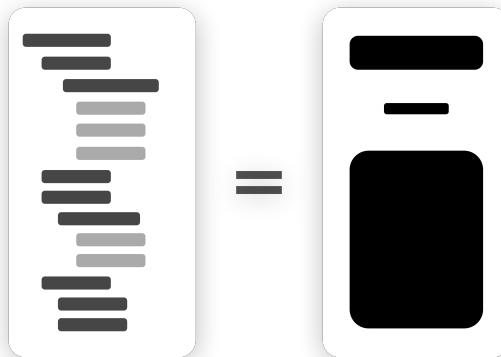
En la imagen 1.3 se observa una representación de la estructura de los archivos que se pueden encontrar en un proyecto con Storyboard.

Muestra todo lo relacionado con interfaz gráfica con un color gris fuerte, todo lo relacionado con programación con un color gris claro y el resultado de éstos dos con un color negro. La suma de éstos dos archivos es una aplicación con interfaz.

En SwiftUI se cambia la estructura y se hace el código para centrarse en cómo se verá la aplicación, haciendo el proceso de escritura algo más orientado a el diseño, siendo la función algo que se da con el paso del tiempo al escribir el código.

Este cambio en la forma de trabajar fuerza a centrarse en el diseño, el *Apple Developer Program* recomienda gastar dos tercios del desarrollo de una aplicación en el diseño de la misma en “Human Interface Guidelines”.

En SwiftUI si una vista no se declara, no existirá, eliminando la posibilidad de llamar a un elemento que no existe en la vista como sucede con Storyboards.



**Imagen 1.4** Estructura de un proyecto con SwiftUI.

Siguiendo el mismo código de colores que en la imagen 1.3, en la imagen 1.4, se muestra la estructura de un proyecto realizado con SwiftUI.

Como se puede observar se describe todo lo que el usuario observará al igual que se define qué funciones y características tienen en el mismo espacio. Incluyendo en esto la actualización de la vista.

## 2.3 Declarativo y automático

Se puede explicar cómo funciona SwiftUI con las características que se describen en su definición: declarativo, automático. Su consistencia y su composición.

### 2.3.1 Declarativo

Con SwiftUI, el diseño de interfaces gráficas se hace por medio de código. A esto se le conoce como declarativo, porque se declaran los elementos que contendrá una vista con código.

```
Text("Hola")  
    .foregroundColor(.green)
```

Con el código anterior se une una vista de texto que contiene la palabra Hola de color verde.

Ésta manera declarativa se relaciona con la programación orientada a objetos, ya que la vista de texto actúa como un objeto, con la propiedad Hola como texto y con un método, al que se accede con la sintaxis del punto, que cambia el del texto a verde.

### 2.3.2 Automatico

Se puede observar que no se especificó el tamaño, posición, opacidad, etcétera de la vista. Lo que quiere decir que SwiftUI realiza éstas tareas automáticamente. Lo que lleva al siguiente punto: Automático

Varma (2019):

"SwiftUI ofrece todas las funciones listas para usar para una funcionalidad gratuita, como Localización; si el código tenía cadenas de idioma, entonces la línea de arriba mostraría la versión localizada, todo sin escribir código adicional, todo automáticamente. Los desarrolladores también pueden aprovechar la funcionalidad de izquierda a derecha, modo oscuro, tipo dinámico y más, todo con la escritura de código mínimo." (p. 3, 4)

SwiftUI posiciona automáticamente una vista dependiendo del tamaño y orientación del dispositivo, asigna el texto, la fuente, el color, asigna el color que tendrá si el dispositivo está en Dark Mode y la proporciona dependiendo de el tamaño del dispositivo y el resto de vistas que existen junto con ella.

Por medio de métodos el programador puede cambiar lo que SwiftUI hace automáticamente, como el cambio de color del ejemplo anterior a verde-

## 2.4 Composición y consistencia

### 2.4.1 Composición

SwiftUI permite cambiar todos los aspectos de una vista, también permite crear nuevas vistas, desde cero, mezclando diferentes elementos que al final dejan una vista nueva, diferente a las preestablecidas, mezclar diferentes vistas lleva al siguiente concepto: Composition

“Este es otro principio interesante en el que SwiftUI se basa simplemente porque una interfaz de usuario no es más que una colección de elementos visuales que juntos proporcionan al usuario una experiencia interactiva.” (Varma, Jayant, 2019:4)

Una interfaz gráfica se compone de muchas vistas individuales, relacionándose así con la programación orientada a objetos.

Ejemplo, en la interfaz gráfica de Instagram se encuentran varios elementos que forman una vista en sí, pero que cada uno por si mismo representa una vista individual.

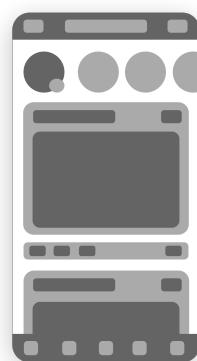


Imagen 1.5 Representación interfaz grafica de Instagram.com

En la imagen 1.5, que es una recreación sobre cómo luce la interfaz gráfica de Instagram, se puede observar éste concepto. Cada vista se ve coloreada con un color distinto, cada color representa un elemento, un botón, un texto, una imagen, etc. Al unir todas éstas vistas se forma un todo.



**Imagen 1.6** Representación de vista de foto y su original de instagram.com

En la imagen 1.6, se puede observar una colección de vistas que actúa como un elemento en la interfaz de la imagen 1.5, específicamente como la vista para una publicación.

La jerarquía de la interfaz que ve el usuario queda de la siguiente forma.

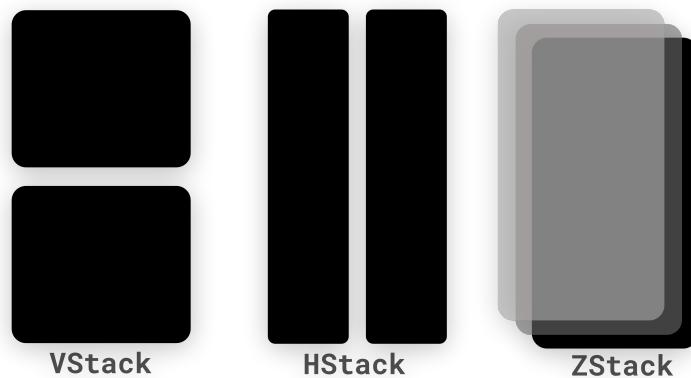
### vista general

- . vista de instagram
  - . - imagen con acción de camara
  - . - texto Instagram
  - . - imagen con acción de corazón
- . vista de historias
  - . - imagen en circulo con acción para publicar historia
  - . - imagen, acción acceso a historia (usuario de imagen)
  - . - Se repite el número de veces que sea necesario
- . vista de publicaciones
  - . - vista publicación
    - . - , vista publicación (foto)
    - . - , . vista imagen con acción llevar a usuario
    - . - , . vista de texto con nombre usuario

- , . vista de imagen con acción reportar
- , . vista de imagen con publicación
- , . vista de comentario / me gusta
- , . vista imagen con acción gustar
- , . vista imagen con acción comentar
- , . vista imagen con acción compartir
- , . Se repite el número de veces que sea necesario
- . vista de navegación
  - imagen con acción ir a la vista inicio
  - imagen con acción ir a la vista buscar
  - imagen con acción ir a la vista publicar
  - imagen con acción ir a la vista de mensajes
  - imagen con acción ir a la vista de perfil

Se aprecia con ésta jerarquía cómo la vista general es una colección de vistas, que a su vez son colecciones de vistas.

Las colecciones de vistas pueden ser de forma vertical, horizontal o encimada como explica la página *Simple Swift Guide* en el artículo “How to use stacks: HStack, VStack and ZStack in SwiftUI”.



**Imagen 1.7** Tipos de colecciones de vistas SwiftUI

## 2.4.2 Consistente

Storyboards presenta errores para pasar la información de una vista a otra o para mantener una vista siempre actualizada. SwiftUI soluciona éste problema.

“Dado que la interfaz de usuario es un reflejo de los datos que representa, siempre debe estar sincronizada para proporcionar una experiencia coherente.” (Varma, Jayant, 2019:4)

SwiftUI proporciona las variables de tipo `@State`. Siempre que cambian de valor actualizan toda la vista.

Paul Hudson (2019):

“Recuerde, SwiftUI es declarativo, lo que significa que le contamos todos los diseños para todos los estados posibles por adelantado, y dejamos que descubra cómo moverse entre ellos cuando cambian las propiedades. Llamamos a este enlace: pedirle a SwiftUI que sincronice los cambios entre un control de UI y una propiedad subyacente.” (*Hacking With Swift*, 2019)

Es importante recordar que `@State` actualiza la vista a la cual pertenece. Para actualizar todas las vistas al mismo tiempo se utiliza algo similar a `@State`: `@EnvironmentObject`.

“Se puede usar para compartir datos entre todas las vistas y para actualizar automáticamente esas vistas cuando los datos cambian.” (*iOSCreator*, 2019.)

Para utilizar `@EnvironmentObject` se necesita crear una clase observable que contenga como propiedades las variables que leerán las vistas que escuchen éste objeto. Éste objeto se instancia directamente en el archivo `SceneDelegate.swift` del proyecto de Xcode como lo menciona Paul Hudson en el artículo “How to use `@EnvironmentObject` to share data between views” en *Hacking With Swift*.

## 2.5 Declaración de vistas

Para declarar dos vistas de texto dentro de una colección de vistas de tipo `HStack` se utiliza el siguiente código:

```
struct ContentView : View {  
    var body: some View {  
        HStack {  
            Text("Texto 1")  
            Text("Texto 2")  
        }  
    }  
}
```

Dónde todos los elementos de la vista se almacenan en la variable “body” de la clase `ContentView` de tipo `View`, es decir vista.

## Capitulo III : Development

### 3.1 Planeación

Para evitar bugs habrá una vista inicial forzosa para el primer inicio de la aplicación que funcionará como inicializador de todos los datos que funcionen con UserDefaults.

Después la inicialización, se pasa a la vista del menú, donde el usuario podrá acceder a distintas vistas por medio del mismo, el menú abarcará una tercera parte del dispositivo donde se utilice, será “flotante”.

En base a éstas primeras especificaciones se tiene que realizar un ZStack. Hasta arriba se encuentra la vista de inicialización, la cual tiene un botón que al presionarse asigna valores a las variables con UserDefaults y cambia su opacidad a 0.



**Imagen 1.8** Estructura Inicial de las vistas de la aplicación

Tal como se puede apreciar en la imagen 1.8, se tiene hasta el frente la vista que servirá como inicializador de los datos de la aplicación, y por debajo de ésta vista irá el nivel 2, que será el menú para cambiar de vistas y por ultimo la vista del nivel 1.

Al iniciar la aplicación por primera vez el usuario primero observará solo la vista de inicio, que se encuentra sobrepuerta a todas las demás. Al presionar el botón con el texto “Start” la vista de inicio desaparecerá, se logrará esto cambiando la opacidad de la misma, al mismo tiempo que desaparece todos los datos correspondientes se inicializaran. Los datos que se inicializaran serán los que indiquen a la aplicación en qué opción del menú se encuentra el usuario, el nombre de la vista del menú en la que se encuentra el usuario, y posibles configuraciones como color del fondo, el tamaño de las letras, etc.

La función se entiende mejor con el diagrama de flujo de la imagen 1.9. La aplicación tiene por defecto la opacidad de la vista de inicio como visible al momento de correrla por primera vez, al presionar Start, se asigna un valor a las variables vacías que contiene la aplicación. Si la ésta ya ha sido ejecutada antes, el programa no inicializa las variables, esta vez las carga de la memoria del dispositivo.

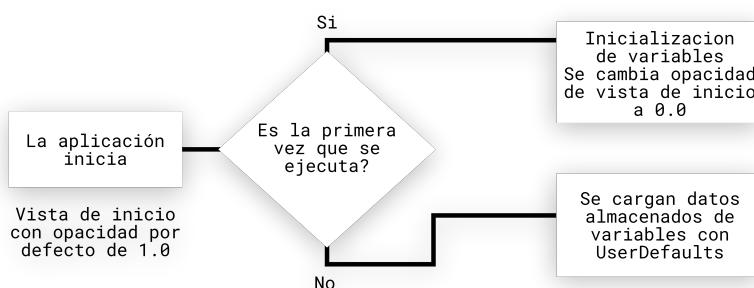
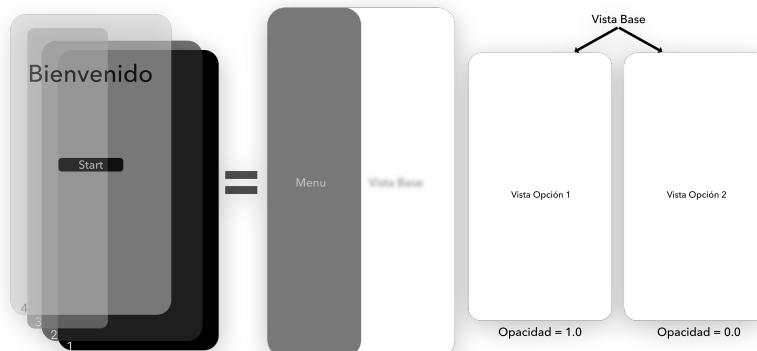


Imagen 1.9 Diagrama de flujo con función de aplicación

El menú contendrá un título menú en la parte superior y contendrá dos botones para que el usuario elija la opción que desee, al activar una opción, la opción que estaba seleccionada con anterioridad cambiará la opacidad que corresponde a su vista a 0.0 y al mismo tiempo la nueva opción cambiará la opacidad correspondiente a su vista a 1.0, creando así el efecto de que el menú muestra una vista u otra dependiendo de lo que seleccione el usuario.

De igual forma, al tener el menú activo, todas las vistas que se encuentren detrás del mismo adoptarán un “blur”, distorsión de desenfoque, para que el usuario se enfoque solo en el menú. De igual forma, cuando el usuario deslice el menú a la izquierda por determinada distancia mínima, lo ocultará.



**Imagen 1.10** Estructura Inicial de las vistas de la aplicación

Con estas nuevas especificaciones, y continuando con lo establecido en la imagen 1.9, en la imagen 1.10 se puede observar cómo se agregó un nuevo nivel, teniendo ahora dos vistas raíz que se encuentran abajo de las vistas de menú y de inicio. Dependiendo de la opción que escoja el usuario se cambiará la opacidad de una u otra para que sea visible o invisible, dependiendo del caso. También se observa

cómo al estar activado el menú se aplica un desenfoque en los elementos debajo del mismo.

Para activar el menú y orientar al usuario se agrega una imagen que funcione como botón para acceder al menú. El cual cambiará su opacidad al estar activo o desactivado para que sea visible o invisible. A un lado de la imagen para acceder al menú se agrega una vista de texto que indique al usuario qué opción está seleccionada. Para realizar esto se agrega una vista más, un nivel entre el menú y las vistas raíz. Quedando de la siguiente forma: Nivel 5: Vista inicio > Nivel 4: Vista menú > Nivel 3: Vista acceso a menú y orientación > Nivel 2: Vista opción 2 > Nivel 1: Vista opción 1

Entrando en detalle con la forma en cómo se programará la vista de inicio, para lograr el aspecto deseado es necesaria la creación de un VStack, es decir una colección de vistas vertical, dentro de ésta se agregaría una vista de texto que tenga como atributo texto “Bienvenido”, ademas de la vista de texto se agregará una vista de botón con su atributo texto como “continuar” como se puede observar en la imagen

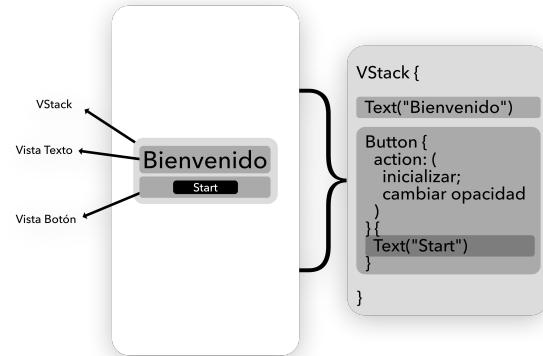


Imagen 1.11 Estructura de Vista Inicio  
1.11

Para la creación del menú se necesita un vertical Stack, dentro de éste VStack se tendría primero una vista de texto con “Menú” como su atributo de texto, después se agrega un espacio y al final se agrega una vista de texto con “Opción 1” como

texto y que al presionarse cambie la opacidad de la Vista “Opción1” a 1.0 y el de todas las posibles vistas activas correspondientes a “Vistas opcionales” a 0.0, se agrega otra vista de texto que realiza algo similar, pero que lleva el texto “Opción 2” como texto y que al presionarse activa la opacidad de la Vista “Opción2”

Debajo de la vista de menú está ubicada la vista para activar el menú e informar al usuario en qué vista se encuentra, puede ser “Opción1” u “Opción2”. Para realizar esta vista se crea un HStack, éste contiene una vista de imagen que al presionarse “activa” el menú cambiando su opacidad, seguido de éste se crea una vista de texto que, de acuerdo a la opción que está almacenada como activa, tendrá como texto “Opción 1” u “Opción 2”. Para lograr esto solo lee la variable que se activa al presionar cualquiera de las opciones en el menú y si es igual a 1 mostrará el texto “Opción 1”, si la variable es igual a 2,. Es decir que se ha elegido la opción 2 del menú, mostrará el texto “Opción 2”. Todo éste proceso se logra gracias a la Variable de Desarrollo o “@EnvironmentObject”, que actúa como un @State, la diferencia es que en vez de actualizar una sola vista, actualiza todas las vistas del programa.

Las siguientes dos vistas, correspondientes a las opciones 1 y 2 de la vista menú solo requieren que se agregue una vista de texto que contenga “Vista opción 1” o “Vista opción 2”

## 3.2 Programación

Para desarrollarlo se codificaron todas las vistas necesarias según la planeación. Se enlazaron todas las vistas con un `@EnvironmentObject` que almacenaba los datos con `Userdefaults` para tener persistencia al salir de la aplicación.

Se programó cada una de las vistas agregando efectos de animación con uso de métodos y cambiando propiedades con condicionales tradicionales como `if`.

Al terminar de programar cada una de las vistas se les unió a todas en una sola para terminar de ensamblarlas juntas, para que así todas sean accesibles en una sola vista. Todas se unen en un mismo `ZStack`, para apilarlas entre sí y que el programa pueda cargarlas todas formando así la vista general.

### **vista general (`ZStack`)**

- .     vista de inicio (`VStack`)
  - vista texto “Bienvenido”
  - vista de botón
    - ,     acción iniciar variables y ocultar
    - ,     texto “Continuar”
- .     vista de menu (`VStack`)
  - vista texto “Menu”
  - vista de botón
    - ,     acción cambiar opacidad del menú a 0.0, cambiar el enfoque de vista de orientación y vista de opción 1 y 2 a 0.0, cambiar opacidad de vista Opción 1 a 1.0 y opacidad vista 2 a 0.0
      - ,     texto “Opción 1”
      - vista de botón
        - ,     acción cambiar opacidad del menú a 0.0, cambiar el enfoque de vista de orientación y vista de opción 1 y 2 a 0.0, cambiar opacidad de vista Opción 2 a 1.0 y opacidad vista 1 a 0.0

```
.      - ,     texto “Opción 2”
.      vista de orientación (HStack)
.      - ,     vista de imagen
.      - ,     imagen de menú
.      - ,     acción cambiar enfoque de vista de opción 1 y 2
así como de la vista orientación a 20.0. Cambiar opacidad de vista de menú a 1.0
.      vista de opción 1
.      - vista texto
.      - ,     texto “Vista Opción 1”
.      vista de opción 2
.      - vista texto
.      - ,     texto “Vista Opción 2”
```

Se unen todas las vistas en un ZStack en una vista general que es la vista general, y se cambian las propiedades de opacidad y desenfoque dependiendo de si el menú esta activado o desactivado y de si la vista 1 o 2 esta activada o desactivada.

Los archivos resultantes, que se pueden consultar en el repositorio de GitHub “Tools-UI” (<https://github.com/AOx0/Tools-UI>), se dividen en 3:

- Están los Delegates de la aplicación (SceneDelegate.swift y AppDelegate.swift).
- Los archivos que describen la interfaz y su funcionamiento (ContentView.swift, MainViewX.swift, VistaInicio.swift y VistaOpciones.swift ) que se pueden compactar en uno solo.
- El archivo encargado de mandar la información a todas las vistas con un @EnvironmentObject (DataConfig.swift)

# Conclusión

El escrito comenzó con una introducción a la programación orientada a objetos con Swift 5 para así entender la redacción de código con SwiftUI, los métodos y atributos de los objetos y dar a conocer la dinámica de “las vistas son objetos” que maneja SwiftUI

Se justificó la definición que brinda Apple de SwiftUI en el *Apple Developer Program* explicando cada una de sus características y comparando el framework con Storyboards a través de la estructura de proyectos y las ventajas que ofrece SwiftUI sobre Storyboards. Se revisó con esto la estructura de un proyecto realizado con SwiftUI y cómo hacer la declaración básica de una vista, así como explicando cómo se almacena toda la vista en la variable “body” de una estructura, que es similar a una clase, de tipo View.

Al final se planificó el concepto de una interfaz gráfica, su estructura y funcionamiento y con apoyo de los conocimientos reunidos previamente se pasó la idea a una aplicación real que se puede consultar en el repositorio de AOx0 para revisar el código o en “Resultado de Tesina” de AOx0 para consultar videos sobre el resultado final. Para la codificación del proyecto se utilizó UserDefaults para la resistencia de datos y un @EnvironmentObject para la distribución de los mismos por las vistas de la aplicación.

# Referencias

## Libros:

1. Varma, J. (2019). "SwiftUI for Absolute Beginners: Program Controls and Views for iPhone, iPad, and Mac Apps". USA: Apress.
2. Kaczmarek, S., Lees, B., Bennett, G., Fisher, M et al. (2018) Objective-C for Absolute Beginners. USA: Apress.
3. Kaczmarek, S., Lees, B., Bennett, G. (2019) "Swift 5 for Absolute Beginners". (5a ed.) USA: Apress.
4. Pressman, R. (1992) "Software Engineering: A practitioners approach". (3ra ed.) USA: McGraw Hill.

## Artículos web:

5. Tyler, J. (2015) "Building Great Software Engineer Teams: Recruiting, Hiring, and Managing Your Team from Startup to Success". USA: Apress.
6. What is Computer Programming? (2018). *CodeAcademy*. Recuperado de <https://news.codecademy.com/what-is-computer-programming/> (14 Abril del 2020).
7. Swift (lenguaje de programación). (s.f.). *Wikipedia*. Recuperado de [https://es.wikipedia.org/wiki/Swift\\_\(lenguaje\\_de\\_programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Swift_(lenguaje_de_programaci%C3%B3n)) (20 de Abril del 2020).
8. The Fricky (2008) "Programación Orientada a Objetos – Clase e Instancia". *The Fricky*. Recuperado de <https://thefricky.wordpress.com/2008/01/15/programacion-orientada-a-objetos-clase-e-instancia/> (12 de Mayo del 2020).

9. Herrera, J. (2020). "Swift – Composición Vs Herencia". *Kodigo Swift*. Recuperado de <https://kodigoswift.com/swift-composicion-vs-herencia/#ftoc-heading-1> (11 Abril del 2020).
10. Álgebra (2012). *Definición.de*. Recuperado de <https://definicion.de/algebra/> (20 de Abril del 2020).
11. Programación orientada a objetos. (2018). Covantec. Recuperado de <https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion9/poo.html> (12 Mayo del 2020).
12. Qué es la programación orientada a objetos (2019). *DesarrolloWeb*. Recuperado de <https://desarrolloweb.com/articulos/499.php> (17 de Abril del 2020).
13. Paradigma (2019). *EcuRed*. Recuperado de <https://www.ecured.cu/Paradigma> (17 de Abril del 2020).
14. Structures and Classes. (2020). *The Swift Programming Language*. Recuperado de <https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html> (5 de Mayo del 2020).
15. Swift (2020.). *Apple Developer Program*. Recuperado de <https://developer.apple.com/swift/> (20 de Abril del 2020).
16. SwiftUI (2019.). *Apple Developer Program*. Recuperado de <https://developer.apple.com/xcode/swiftui/> (20 de Abril del 2020).
17. Human Interface Guidelines (2020). *Apple Developer Program*. Recuperado de <https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/> (25 de Abril del 2020).
18. Hudson, P. (16 de Septiembre del 2019). "SwiftUI vs Interface Builder and storyboards". *Hacking with Swift*. Recuperado de <https://hackingwithswift.com/100/swiftui-vs-interface-builder-and-storyboards>

- www.hackingwithswift.com/quick-start/swiftui/swiftui-vs-interface-builder-and-storyboards (22 de Abril del 2020).
19. Hudson, P. (18 de Septiembre del 2019). “Working with State”. *Hacking with Swift*. Recuperado de <https://www.hackingwithswift.com/quick-start/swiftui/working-with-state> (21 de Abril del 2020).
20. Hudson, P. (19 de Septiembre del 2019). “How to use @EnvironmentObject to share data between views”. Hacking with Swift. Recuperado de <https://www.hackingwithswift.com/quick-start/swiftui/how-to-use-environmentobject-to-share-data-between-views> (30 de Abril del 2020.)
21. SwiftUI EnvironmentObject Tutorial. (2019). *iOSCreator*. Recuperado de <https://www.ioscreator.com/tutorials/swiftui-environment-object-tutorial> (24 de Abril del 2020).
22. How to use stacks: HStack, VStack and ZStack in SwiftUI. (2019). *Swift Simple Guide*. Recuperado de <https://www.simpleswiftguide.com/how-to-use-stacks-hstack-vstack-and-zstack-in-swiftui-equivalent-of-uistackview-in-uikit/> (25 de Abril del 2020).
23. Osornio, A. (s.f.) “Resultado de Tesina”. AOx0. Recuperado de <https://aox0.github.io/Posts/cel/tesina.html> (10 de Mayo del 2020).
24. Tools-UI (s.f.). *GitHub / AOx0*. Recuperado de <https://github.com/AOx0/Tools-UI> (12 de Mayo del 2020).

**Páginas web:**

25. *GitHub*. <https://github.com> (10 de Mayo del 2020).
26. *Instagram*. <https://www.instagram.com> (10 de Mayo del 2020)
27. Swift Documentation. <https://swift.org/documentation> (10 de Mayo del 2020)

# Matriz de congruencia

Preguntas	Objetivo	Hipótesis	Contenido	Instrumentos	Variables
¿Cómo se relaciona la programación orientada a objetos con SwiftUI?	Conocer la teoría detrás de la programación orientada a objetos para comprender el código redactado con SwiftUI	Si se conoce la programación orientada y su funcionamiento, se puede leer y escribir código enfocado a objetos de vista.	Definición de la programación orientada a objetos. Definición de objeto, atributos, métodos. Accesar a propiedades y métodos	Análisis de documentos y artículos.	Modelo que se emplea.
¿Qué es SwiftUI y qué diferencias / novedades incluye respecto con Storyboards?	Conocer las diferencias entre la manera de funcionar de proyectos realizados con distintos frameworks	Si se conoce las diferencias entre Storyboards y SwiftUI se pueden apreciar las novedades del mismo.	Definición de SwiftUI , análisis de características y comparación de funciones y estructuras de programas realizados con distintos frameworks. (Storyboard)	Comparación de archivos de programas de Xcode. Análisis de documentos.	Framework empleado
¿Cómo funcionan las colecciones de vistas en SwiftUI?	Conocer las características de colecciones de vistas en SwiftUI para poder lograr el desarrollo de interfaces complejas.	Si se conocen las colecciones de vistas, es posible realizar interfaces complejas con los conocimientos adquiridos	Definición de colección, definición de tipos de colecciones y su declaración con SwiftUI.	Análisis de documentos y artículos.	Tipo de colección a emplear
¿Cómo es posible el intercambio de datos entre vistas con SwiftUI?	Conocer los @State y @Environment de SwiftUI para la actualización de vistas	Si se sabe cómo usar @proppertyWrappers en vistas con SwiftUI se puede mantener comunicadas y actualizadas las vistas.	Definición de @State, definición de @EnvironmentObject y funcionamiento de @EnvironmentObject en un proyecto con SwiftUI	Análisis de documentos.	Tipo de dato a actualizar. Tipo de situación de actualización.
¿Cómo llevar a cabo el desarrollo de una interfaz con SwiftUI?	Desarrollar una interfaz gráfica con SwiftUI	Si se conoce el funcionamiento de SwiftUI y la programación orientada a objetos se puede llevar a cabo el desarrollo de un interfaz con el mismo	Planificación y estructuración de vistas, idealización del manejo de datos y colecciones necesarias desarrollar la interfaz	Redacción de código. Creación de Documentos	Versión de Xcode Versión de Swift Framework.a utilizar Modelo de programación a utilizar