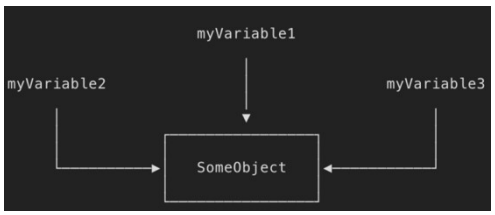
PREGUNTAS SWIFT – NTTDATA

ARC (Automatic Reference Counting)

A diferencia de otros lenguajes como Java o Kotlin, Swift no tiene un recolector de basura. En su lugar, utiliza una técnica mucho más rápida denominada Automatic Reference Counting (ARC).
ARC puede liberar memoria ya que lleva la cuenta de cuántas propiedades, constantes o variables apuntan hacia una instancia.
Cada vez que asignamos una variable(myVariable) a una instancia de una clase (SomeObject), la cuenta de referencias aumenta en +1. Si asignamos la misma instancia a 3 variables diferentes, su cuenta de referencias será 3 como el ejemplo siguiente.

```
var myVariable1: SomeObject = SomeObject()
var myVariable2 = myVariable1
var myVariable3 = myVariable1
```



Sólo cuando esa cuenta de referencias baje hasta cero (haciendo nulo a sus variables), ARC considerará que esa instancia nunca se va a volver a utilizar y se puede eliminar de memoria.
OJO: Cuando intentas acceder a una instancia de una clase que ya ha sido eliminada de memoria, se produce un crash en tu aplicación.

MRC (Manual Reference Counting)

MRC es la gestión manual de enlaces a través del código. Al principio y en tiempos prehistóricos, los propios desarrolladores lograron el recuento de enlaces a través de comandos. Fue, por decirlo suavemente, difícil:

- alloc - crear un objeto (crear un enlace)
- retain - acceso a él (+1 al enlace)
- release - reducir el contador de referencia (-1)
- dealloc - si el contador de referencia es 0 = descarga de la memoria

De hecho, seleccionas un objeto, lo guardas en algún momento y luego envías un problema para cada selección / guardado que envías. El método dealloc se llama para un objeto cuando se elimina de la memoria.

Problemas:

- Tienes que contar constantemente retener, release
- se bloquea al acceder desde descargado de la memoria
- olvidé poner una liberación - fuga de memoriaC

TIPOS DE VALOR Y REFERENCIA

Los tipos en Swift se dividen en una de dos categorías:

1) "Tipos de valor" -> donde cada instancia guarda una copia única de sus datos. Por lo general, se define por

- estructuras (incl. matrices y diccionarios)
- enumeraciones
- tupla
- tipos de datos básicos (booleano, entero, flotante, etc.)

Así que usa un tipo de valor cuando:

- Comparar los datos de la instancia con == tiene sentido
- Quieres que las copias tengan un estado independiente
- Los datos se utilizarán en código a través de múltiples hilos

2) "Tipos de referencia" -> donde las instancias comparten una sola copia de los datos, y el tipo generalmente se define como una clase.

Usa un tipo de referencia (por ejemplo, usa una clase) cuando:

- Comparar la identidad de la instancia con === tiene sentido
- Quieres crear un estado compartido y mutable

¿CUÁL ES LA DIFERENCIA?

Tipo de valor -> La característica distintiva más básica de un tipo de valor es que la copia, el efecto de la asignación, la inicialización y el paso de argumentos, crea una instancia independiente con su propia copia única de sus datos:

```
struct S {
var data: Int = -1
}
var a = S()
var b = a                          // a is copied to b
a.data = 42                                // Changes a, not b
println("\(a.data), \(b.data)")    // prints "42, -1"
```

Tipo de referencia -> Copiar una referencia, por otro lado, crea implícitamente una instancia compartida. Después de una copia, dos variables se refieren a una sola instancia de los datos, por lo que modificar los datos en la segunda variable también afecta al original, por ejemplo:

```
class C {
   var data: Int = -1
}
var x = C()
var y = x          // x is copied to y
x.data = 42                // changes the instance referred to by x (and y)
println("\(x.data), \(y.data)")    // prints "42, 42"
```

Strong

Cada vez que una variable de puntero almacena la dirección de un objeto, ese objeto es propietario y permanecerá vivo. Esto se conoce como una referencia fuerte. Así que significa que quieres "poseer" el objeto. Solo cuando establezcas la propiedad en cero, el objeto será destruido.

Weak

Tiene la funcion de eliminar un ciclo de retencion ya que este hace que la relacion entre ellas se destruya al solo contar con una referencia fuerte. Cumple con valores opcionales, es decir puede ser nulo o no. Se denota con el simbolo “?”

Unowned

Tiene la misma función que Weak, pero además no puede ser nulo. Esto significa que solo puede usar unowned si no accede al espacio de memoria eliminado. Por lo general, No se recomienda Unowned

¿Por qué usamos Strong?

- Crea propiedad entre la propiedad y el valor asignado.
- Este es el valor predeterminado para la propiedad de objeto en ARC.
- Débil: una variable opcionalmente no puede tomar posesión de un objeto al que apunta. Una variable que no se hace cargo de un objeto se conoce como una referencia débil.
- Así que una propiedad débil significa que no quieres tener control sobre el ciclo de vida de los objetos. El objeto solo vive mientras que otros objetos tienen una fuerte referencia a él. Si no hay referencias fuertes al objeto, entonces será destruido. El escenario más común para una propiedad débil es para las subvistas de una vista de controladores de vista. Porque estas subvistas ya están fuertemente mantenidas por la vista de controladores de vista.

¿Por qué usamos Weak?

- Crea no propiedad entre la propiedad y el valor asignado.
- Strong se usa en el objeto padre y débil en el objeto hijo cuando se libera el padre, luego la referencia del objeto hijo también se establece en nil
- Ayuda a evitar ciclos de retención.
- Weak está esencialmente asignado, propiedad no retiene. Una referencia débil es útil para una situación inusual llamada ciclo de retención.

Retain Cycle (ciclo de retención)

Un ciclo de retención ocurre cuando dos o más objetos tienen fuertes referencias entre sí. Estas son malas noticias. Cuando dos objetos son propiedad, nunca serán destruidos por el ARC. Incluso si todos los demás objetos de la aplicación liberan la propiedad de estos objetos, estos objetos (y cualquier objeto que posean) seguirán existiendo en virtud de esas dos fuertes referencias. Por lo tanto, un ciclo de retención es una fuga de memoria que ARC necesita tu ayuda para arreglar. Lo arreglas haciendo que una de las referencias sea débil.

Git

Pull Request

Un pull request es una petición que el propietario de un rama de un repositorio hace al propietario del repositorio original para que este último incorpore los commits que están en sus cambios. En el caso que nos ocupa, el usuario (yo) le enviará la petición a (master) p crea un repositorio nuevo

Crea un directorio nuevo, ábrelo y ejecuta para crear un nuevo repositorio de git.
git init

hacer checkout a un repositorio
Crea una copia local del repositorio ejecutando
git clone /path/to/repository
Si utilizas un servidor remoto, ejecuta
git clone username@host:/path/to/repository

flujo de trabajo
Tu repositorio local esta compuesto por tres "árboles" administrados por git. El primero es tu Directorio de trabajo que contiene los archivos, el segundo es el Index que actua como una zona intermedia, y el último es el HEAD que apunta al último commit realizado.

add & commit
Puedes registrar cambios (añadirlos al Index) usando
git add <filename>
git add .
Este es el primer paso en el flujo de trabajo básico. Para hacer commit a estos cambios usa
git commit -m "Commit message"
Ahora el archivo esta incluído en el HEAD, pero aún no en tu repositorio remoto.

envío de cambios
Tus cambios están ahora en el HEAD de tu copia local. Para enviar estos cambios a tu repositorio remoto ejecuta
git push origin master
Reemplaza master por la rama a la que quieres enviar tus cambios.

Si no has clonado un repositorio ya existente y quieres conectar tu repositorio local a un repositorio remoto, usa
git remote add origin <server>
Ahora podrás subir tus cambios al repositorio remoto seleccionado.

ramas
Las ramas son utilizadas para desarrollar funcionalidades aisladas unas de otras. La rama master es la rama "por defecto" cuando creas un repositorio. Crea nuevas ramas durante el desarrollo y fusiónalas a la rama principal cuando termines.

Crea una nueva rama llamada "feature_x" y cámbiate a ella usando
git checkout -b feature_x
vuelve a la rama principal
git checkout master

y borra la rama
git branch -d feature_x
Una rama nueva no estará disponible para los demás a menos que subas (push) la rama a tu repositorio remoto
git push origin <branch>

actualiza & fusiona
Para actualizar tu repositorio local al commit más nuevo, ejecuta
git pull
en tu directorio de trabajo para bajar y fusionar los cambios remotos.
Para fusionar otra rama a tu rama activa (por ejemplo master), utiliza
git merge <branch>
en ambos casos git intentará fusionar automáticamente los cambios. Desafortunadamente, no siempre será posible y se podrán producir conflictos. Tú eres responsable de fusionar esos conflictos manualmente al editar los archivos mostrados por git. Después de modificarlos, necesitas marcarlos como fusionados con
git add <filename>
Antes de fusionar los cambios, puedes revisarlos usando
git diff <source_branch> <target_branch>

etiquetas
Se recomienda crear etiquetas para cada nueva versión publicada de un software. Este concepto no es nuevo, ya que estaba disponible en SVN. Puedes crear una nueva etiqueta llamada 1.0.0 ejecutando
git tag 1.0.0 1b2e1d63ff
1b2e1d63ff se refiere a los 10 caracteres del commit id al cual quieres referirte con tu etiqueta. Puedes obtener el commit id con
git log
también puedes usar menos caracteres que el commit id, pero debe ser un valor único.

reemplaza cambios locales
En caso de que hagas algo mal (lo que seguramente nunca suceda ;) puedes reemplazar cambios locales usando el comando
git checkout -- <filename>
Este comando reemplaza los cambios en tu directorio de trabajo con el último contenido de HEAD. Los cambios que ya han sido agregados al Index, así como también los nuevos archivos, se mantendrán sin cambio.
Por otro lado, si quieres deshacer todos los cambios locales y commits, puedes traer la última versión del servidor y apuntar a tu copia local principal de esta forma
git fetch origin
git reset --hard origin/master

datos útiles
Interfaz gráfica por defecto
gitk
Colores especiales para la consola
git config color.ui true
Mostrar sólo una línea por cada commit en la traza
git config format.pretty oneline
Agregar archivos de forma interactiva
git add -iara que este último incorpore los commits que tiene en su rama master.

MOCKUP

¿Qué es un Mockup?
El término Mockup es muy utilizado entre los diseñadores gráficos ya que corresponde a un montaje o boceto que se realiza con la intención de mostrar al cliente el resultado final de un proyecto. Así, se puede elaborar una maqueta en la que se enseña cómo quedaría un logo, el packaging de un producto, la estética de una web o la estructura de una app y a su vez la posibilidad de crear una simulación que permite al cliente tener una idea más realista de cómo va a quedar el trabajo solicitado.

¿Para qué sirve un Mockup?
Los clientes podrán realizar modificaciones en el diseño elaborado, incluyendo y eliminando elementos, colores, formas o tipografías de forma sencilla.
Este hecho tendrá como consecuencia directa un ahorro de tiempo y dinero, ya que no es necesario tener la página web o la aplicación creada para realizar cambios.
Además que radica en la detección de problemas o deficiencias que tiene el proyecto, pudiendo solventar estos dilemas rápidamente.

¿Cuándo debemos utilizar este recurso?
Los Mockups se utilizan habitualmente en la fase inicial del desarrollo de un proyecto, como podría ser la creación de una web o una app, por ejemplo. Así, se presenta la apariencia de la plataforma, a la vez que se realiza un control de calidad de la misma.

¿Cómo puedo crear un Mockup?
Entre las plataformas más conocidas destacan y gratuitas estan Gliffy, Cacoo y Mockflow. Por el contrario, si preferimos utilizar soportes de pago con los que realizar Mockups completamente profesionales, podemos optar por CreativeMarket y Envato Elements.

Requisitos Funcionales

Los requisitos que especifican los aspectos funcionales del software se conocen como requisitos funcionales. Los requisitos funcionales cambian de un proyecto a otro. Definen las funcionalidades que brindan los sistemas o componentes.
Ejemplo: Suponga un sistema de gestión hospitalaria. Puede tener varios módulos, como módulo de inicio de sesión, módulo de paciente, módulo de médico, módulo de cita, módulo de informe y módulo de facturación.

Requisitos No Funcionales

Los requisitos que no están relacionados con el aspecto funcional del software se incluyen en la categoría de requisitos no funcionales. Definen las características esperadas de un software. Los usuarios pueden hacer suposiciones sobre ellos. A muchos usuarios les preocupa conseguir los requisitos no funcionales adecuados, especialmente para sistemas grandes.
Ejemplo: Un sistema de gestión del hospital debe tener requisitos no funcionales como rendimiento, seguridad, mantenibilidad, usabilidad, confiabilidad y disponibilidad.

Diferencia entre Req. Funcionales y No Funcionales

Los requisitos funcionales son los requisitos que definen las funciones de un sistema o sus subsistemas. Los requisitos no funcionales son los requisitos que especifican los criterios que se pueden utilizar para juzgar el funcionamiento del sistema.
Uso
Los requisitos funcionales se utilizan para describir las funcionalidades de un sistema.
Los requisitos no funcionales describen las características de calidad del sistema o los atributos de calidad.

Modularizacion

La palabra modularización proviene de módulo, cuya definición textual es: «Elemento con función propia concebido para poder ser agrupado de distintas maneras con otros elementos constituyendo una unidad mayor.»
La modularización es el proceso por el cual seleccionamos y agrupamos instrucciones de programación que cumplen una función específica.

Crear módulos es importante debido a lo siguiente:
Supongamos que tenemos un problema, cuya solución consiste en ejecutar tantas instrucciones de programación como se necesite. Utilizando módulos de programación, podemos construir una solución que se pueda ejecutar empleando una sola instrucción, esta instrucción lo que hará es ir al contenido del módulo, ejecutar cada una de sus instrucciones y al finalizar el programa retornará al punto donde se hizo la llamada al módulo.
La ventaja de esto es que el módulo lo creamos una única vez y luego lo podemos reutilizar cuantas veces sea necesario y desde cualquier parte del código.

Modulo

Respecto a la Funcionalidad: Un módulo debe tener una función específica y no ir más allá de sus responsabilidades. Por ejemplo, si tenemos un módulo que se encarga de calcular el producto de dos números y devolver el resultado, no sería adecuado que además imprima mensajes en pantalla o haga otra cosa que vaya más allá de su propósito.

Respecto a la Identificación: Los módulos tienen un nombre de identificación que nos permite ejecutarlos, es recomendable que elijamos este nombre de modo que sea lo más representativo posible de la función que realiza, de esa forma resulta más intuitivo a la hora de usarlo.

Algoritmo de Busqueda

Los procesos de búsqueda involucran recorrer un arreglo completo con el fin de encontrar algo. Lo más común es buscar el menor o mayor elemento (cuando es puede establecer un orden), o buscar el índice de un elemento determinado.

- **Búsqueda Lineal:** Para buscar el menor o mayor elemento de un arreglo, podemos usar la estrategia, de suponer que el primero o el último es el menor (mayor), para luego ir comparando con cada uno de los elementos, e ir actualizando el menor (mayor).
- **Búsqueda Secuencial:** Consiste en ir comparando el elemento que se busca con cada elemento del arreglo hasta cuando se encuentra.
- **Búsqueda Binaria:** Consiste en comparar si el valor buscado está en la mitad superior o inferior. En la que esté, subdivido nuevamente, y así sucesivamente hasta encontrar el valor.

Algoritmo de Ordenamiento

Un algoritmo de ordenamiento colocará los objetos - números y letras- en orden. Será muy útil para índices por ej. [a, b, c, d] es un arreglo ordenado alfabéticamente. [1, 2, 3, 4, 5] es un arreglo de números enteros ordenados ascendentemente.

- **Merge Sort:** Utiliza un principio que es divide y vencerás, se separa en partes para resolver problemas individuales, y es relativamente rápido. Al final que hace la fusión de los datos en el orden correcto, es más útil para cuando se tiene muchos datos.
- **Insertion Sort:** Se va comparando valores y se va colocando del lado izquierdo, cada vez que se va llegando un valor nuevo. Se utilizar cuando tu set de datos es corto.
- **Buble Sort:** Es un algoritmo basico, va ordenando de pares y dejando al menor primero, es un lento. Es el algoritmo muy basico que existe y funciona cambiando las celdas adyacentes en caso de que no esten ordenadas y va de par en par.

PROGRAMACION ORIENTADA A OBJETOS (POO)

Antes que nada, es importante comprender que POO es un modelo mental. Es una esquema a partir del cual podemos representar un problema en código a partir de unidades simples.
Los datos y los procesamientos coexisten dentro de unidades llamadas objetos. Un objeto no es más que datos, y procesamientos (funciones, también llamadas métodos), sobre esos datos.
Por ejemplo, podemos tener un objeto "jorge", que tiene como datos, un String nombre, con valor "Jorge", y un String apellido, con valor "Gomez", y un método obtenerNombreCompleto() que devuelva el String "Jorge Gomez". Tanto los datos, como las funciones que operan con esos datos conviven en esa unidad llamada objeto.

Clases

Una clase es un molde a partir del cual se crean objetos. Así, si teníamos el objeto "jorge", podemos tener la clase Persona, que define el tipo de datos que va a tener jorge, así como los métodos que contendrá.
Ojo: self es la palabra clave que nos permite referirnos al objeto en el que estamos operando en este momento.

Herencia

La herencia nos dice que si tenemos dos clases, y una es "un tipo de" la otra clase, entonces la primera clase "hereda", o "es hija" de la segunda, que se considera la clase "padre", o "superclase". Por ejemplo, un Estudiante es "un tipo de" Persona, entonces tendría sentido que Estudiante "herede" de Persona.
Como consecuencia de la herencia, la clase que hija obtiene todos los atributos y métodos de la clase padre, en adición a los atributos y métodos de la propia clase hija.

Polimorfismo

El polimorfismo nos dice que si tenemos un objeto de una clase hija, ese objeto es también del tipo de la clase padre.
Una clase hija puede redefinir el comportamiento (los métodos) de su clase padre.
Cuando una función nos pide un objeto de una clase determinada, podemos pasarle un objeto de una clase hija en su lugar. Lo mismo sucede si queremos asignar un objeto de una clase hija a una variable de tipo de la clase padre.
Otra forma más sencilla de definir polimorfismo es que si llamo al método animal.hablar(), no sé si el animal es un Pato, un Perro o un Gato, lo que sí sabemos es que si el animal era un Perro, al hablar dirá "Guau", si el animal es un Gato, dirá "Miau", y si es un Pato dirá "Cuak!".

Encapsulamiento

Se basa en solo permitir el acceso a determinados atributos y métodos para ser llamados desde fuera de la clase que los define.

Para esto usamos los modificadores de acceso de Swift que son:

- **private:** una variable, función o clase privada no puede ser accedida desde fuera de su alcance. Es el modificador de acceso más restrictivo.
- **fileprivate:** una variable o función fileprivate (no aplica a clases), fileprivate no puede ser accedida desde fuera del archivo en el que fue declarada.
- **internal:** una variable, función o clase internal solo puede ser accedida dentro del módulo en el que fue declarada. Es el modificador de acceso por defecto, es decir que de no especificar nada, toda variable, función a clase que declaremos será internal.

- **public:** una variable, función o clase public puede ser accedida desde cualquier otro lugar del código sin restricción.
- **open:** una variable, función o clase public puede ser accedida desde cualquier otro lugar del código sin restricción, y además, si una clase es open, otra clase puede heredar de ella desde otro módulo, y si una función perteneciente a una clase es open, puede ser sobreescrita (override) desde cualquier otro módulo.

Protocolos

Un protocolo es un "contrato o plano" que define requisitos con los que una clase que "implementa" el protocolo debe cumplir. Es lo que en otros lenguajes de programación se llama interface. En la práctica, consiste en una lista de metodos o propiedades.

Enumeraciones

Los Enums definen un tipo común para un grupo de valores relacionados y permite trabajar con estos valores de una manera segura dentro de tu código.
Es del Tipo Valor o Value Type

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

Extensiones

Las extensiones en Swift nos proporcionan la capacidad de ampliar la funcionalidad de una clase, estructura, enumeración o protocolo. No poseen nombre, y nos permiten de forma transparente y mágica, extender, valga la redundancia, funcionalidad de tipos a los que tenemos o NO tenemos acceso.

Siguiendo la propia documentación de swift.org, las extensiones nos permiten:

- Añadir propiedades de instancia calculadas y de tipo computadas.
- Definir funciones de instancia y de tipo.
- Crear nuevos inicializadores.
- Definir subscripts.
- Definir y utilizar nuevos tipos anidados.
- Hacer que un tipo existente se ajuste a un protocolo.

```
let myMeters:Double = 5000

extension Double {
    var km: Double{
        return self / 1000
    }
    var m: Double{
        return self
    }
    var cm: Double{
        return self * 100
    }
}

print(myMeters.km)
```

Estructuras

Las estructuras en Swift se definen como una herramienta encargada de la agrupación de un conjunto de datos relacionados, a los que se conoce como propiedades y métodos, y que, a su vez, permitirán añadir funcionalidades.
Las llamadas Swfits structs se refieren también a tipos de valores que permiten aumentar la gestión del sistema, en relación con la mutación de los datos.

Closures

Los cierres, también conocidos en inglés como closures en Swift, son fragmentos que permiten capturar y almacenar referencias a cualquier variable o constante del contexto en el que se encuentran definidas. En términos de Apple, los closures se consideran bloques que son autónomos de funcionalidad y que, además, es posible "pasar" y usar en nuestro código. Esto quiere decir que se pueden "pasar" e invocar en lugares distintos a su implementación.

En lenguajes de programación como C y Objetive-C, existen unos elementos llamados bloques o blocks que son similares a los cierres en Swift. En Kotlin también es posible encontrar una similitud a los closures, la cual es la denominada lambda, que es una forma de representar una función.

¿Para qué sirven los closures en Swift?

Después de saber qué son closures en Swift, puede surgir la duda de para qué sirven. A grandes rasgos, los denominados cierres en Swift son fragmentos que permiten capturar y almacenar referencias a cualquier variable o constante del contexto en el que se encuentran definidas.

¿Cómo declarar un closure en Swift?

Tras conocer qué son closures en Swift y para qué sirven, resulta oportuno hablar también acerca de cómo se declaran estos bloques de código. De forma resumida, podemos decir que la declaración de la composición de un closure o cierre en Swift se hace de la siguiente forma:

```
{ (parámetrosEntrada) -> TipoDevuelto in
  código
}
```

Esto significa que, para declarar un cierre o closure en Swift, debes establecer unos parámetros de entrada y el tipo de retorno in.

Tuplas

Las tuplas es un tipo de dato compuesto que es una de las grandes novedades que incorporó Swift, y cuyo mayor valor no es el tipo en sí, si no que está muy integrado en el sistema y por lo tanto podemos sacarle un gran rendimiento.

Ejemplo: Imaginemos que queremos declarar varias variables a la vez y darles un valor, como por ejemplo, un alto y un ancho. Normalmente haríamos:

```
var alto = 2
var ancho = 2
```

Con ayuda de las tuplas, podemos reducir el código así:

```
var (alto, ancho) = (2, 2)
print(alto)
print(ancho)
```

Enumeracion:

Podemos enumerar tanto arrays como diccionarios a través de tuplas, pudiendo acceder a cada elemento de manera individual a través del índice.

Son un conjunto de datos de un mismo tipo que agrupa valores que se relacionan entre sí. Normalmente se usan para acotar posibles características de una propiedad, donde uno de los valores de la enumeración es asignado a una variable tipificada en esta.

Por ejemplo, si tenemos una propiedad de la clase vehículo que es puertas, es fácil saber qué valores vamos a poner: 1, 2, 3 o 4. Podemos usar un entero y poner los valores. Pero, ¿y sí queremos que estos sean los únicos posibles valores de nuestro campo? Usamos las enumeraciones.

```
enum puertas {
   case una, dos, tres, cuatro
}
```

Las enumeraciones tienen dos tipos de valores: la descripción y el valor interno o valor hash. Nosotros siempre trabajaremos con la descripción, pero internamente el sistema les asigna un valor y es como reconoce cada correspondencia.

PATRONES DE DISEÑO CREACIONALES

Singleton
Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Factory Method
Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Builder
Builder es un patrón de diseño creacional que 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.

PATRONES DE DISEÑO ESTRUCTURALES

Adapter
Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

Composite
Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Decorator
Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

PATRONES DE COMPORTAMIENTO

Chain of Responsability
Chain of Responsibility es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Iterator
Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

Observer
Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

/**********/

Diferencia Entre Pila y Cola
Stack y Queue son ambas estructuras de datos no primitivas. Las principales diferencias entre la pila y la cola son que la pila utiliza el **método LIFO** (último en entrar, primero en salir) para acceder y agregar elementos de datos, mientras que la cola usa el **método FIFO** (primero en entrar, primero en salir) para acceder y agregar elementos de datos.

La pila solo tiene un extremo abierto para empujar y hacer estallar los elementos de datos en la otra parte. La cola tiene ambos extremos abiertos para encolar y sacar en cola los elementos de datos.

La pila y la cola son las estructuras de datos utilizadas para almacenar elementos de datos. Realizan las siguientes operaciones: la Pila (Push and Pop) y la Cola (Encolar(add) y En Cola(delete))

Por ejemplo, la pila es una pila de CD donde puede sacar y poner un CD en la parte superior de la pila de CD. De manera similar, la cola es una cola para los boletos de teatro donde la persona que se encuentra en primer lugar, es decir, se colocará primero en la parte delantera de la cola y la nueva persona que llegue aparecerá en la parte posterior de la cola (parte posterior de la cola).

Integrar Codigo a una rama

Principalmente se utiliza lo que es el merge, ya que es la fusión mediante que se toman los cambios de una rama (del mismo repositorio o de una bifurcación) y se aplican en otra. Pero en la parte superior donde dice “Branch” tenemos 3 opciones que podemos integrar un codigo a una rama:

- Merge into Current Branch
- Squash and Merge into Current Branch (Fusionar Mediante Combinacion y Combinar)
- Rebase Current Branch (Fusionar mediante Cambio de Base)

Patron Delegado o Delegation Pattern

El patrón de delegación es otro patrón de comportamiento muy usado en el entorno Apple. Consiste en dar una responsabilidad a otro objeto para que complemente un trabajo.

Un ejemplo, en iOS tenemos UICollectionView para representar de forma visual elementos en forma de grid. Esta clase tiene dos delegates para enriquerezer la UICollectionView. Estos delegados son UICollectionViewDelegate y UICollectionViewDatasource. Si implementamos estos delegados se ejecutan algunas funciones que nos pueden resultar útiles. Como por ejemplo, cuando un user selecciona una celda, o cuando se añade un elemento al grid, etc.

Vamos a ver un ejemplo muy simple. Normalmente la vista recibe las acciones del usuario y delega el trabajo a una clase para que realice algún cálculo, hagua una petición HTTP, etc y luego vuelva a notificar a la vista para indicar que todo ha ido bien.

Pruebas de Interfaz

Cuando los usuarios usan una aplicación móvil por primera vez, no es solo el rendimiento lo que atrae la atención, sino también la atractiva interfaz de usuario. Una aplicación amigable con la interfaz de usuario vende más en comparación con una aplicación que está mejor desarrollada pero con una interfaz de usuario desagradable.

Si una aplicación tiene una interfaz de usuario perfecta y espléndida en un dispositivo, pero en el otro dispositivo está completamente retorcida solo porque tiene un tamaño diferente o un sistema operativo diferente, entonces dejará una muy mala impresión. El éxito comercial de la aplicación se verá gravemente afectado.

- Resolucion de la Pantalla
- Tamaño de la Pantalla
- Diferentes elementos en la Interfaz de Usuario
- Estilo, Color y Tema del Dispositivo ,etc

Pruebas Unitarias

Las pruebas unitarias consisten en aislar una parte del código y comprobar que funciona a la perfección. Son pequeños tests que validan el comportamiento de un objeto y la lógica.
El unit testing suele realizarse durante la fase de desarrollo de aplicaciones de software o móviles.

Consisten en verificar el comportamiento de las unidades más pequeñas de su aplicación. El verdadero propósito de las pruebas unitarias es proporcionarle una retroalimentación casi instantánea sobre el diseño y la implementación de su código.

- XCTest
- iOS Unit Testing Bundle
- Red - Green – Refactor

SOLID:

Los 5 principios SOLID de diseño de aplicaciones de software son:

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

- Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
- Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
- Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.

En definitiva, desarrollar un software de calidad.

- **Single Responsability:** Cada clase debe tener una sola responsabilidad
- **OpeN/Close:** mis entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación
- **Liskov Sustition:** objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Es decir si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.
- **Interface Segregation:** muchas interfaces cliente específicas son mejores que una interfaz de propósito general.
- **Dependency Inversion:** a noción de que se debe "depender de abstracciones, no depender de clases concretas" .Es decir Los modulos de alto nivel no deberian depender de modulos de bajo nivel. Ambos deberian depender de abstracciones. La Inyección de Dependencias es uno de los métodos que siguen este principio.

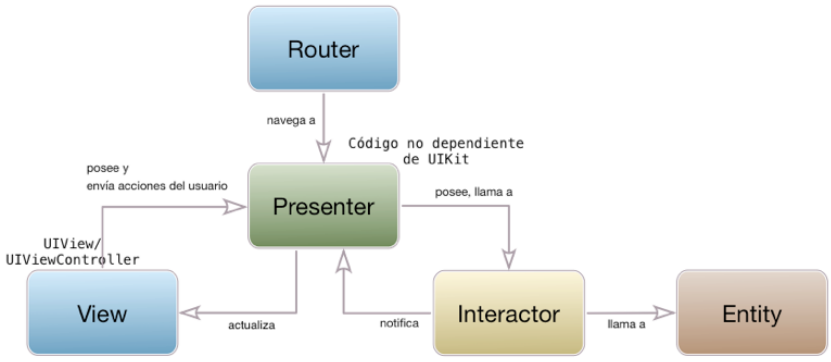
VIPER:

La arquitectura VIPER es un patrón de diseño muy usado para la construcción de aplicaciones móviles, su diseño se basa en la separación por capas de los distintos componentes de nuestro código para una correcta aplicación del principio de responsabilidad única.

De tal forma, existen muchas variantes del patrón, pero en términos generales se trata de separar la capa de la interfaz (View), la capa de negocio (Interactor), la capa de presentación (Presenter), la capa de datos (Entity) y la capa de navegación (Routing).

Como hemos enumerado antes, los componentes son los siguientes:

- View: muestra los datos que le pasa el presenter, y le pasa a este las acciones del usuario. Como vemos es el mismo papel que desempeña en MVP o MVVM.
- Interactor: es la lógica de negocio, los casos de uso de nuestra aplicación. Típicamente cada módulo VIPER implementará un caso de uso distinto.
- Presenter: contiene la lógica de presentación, al igual que en MVP.
- Entity: los modelos del dominio. Se podría decir que contienen la lógica de negocio "genérica" mientras que el interactor contiene la lógica propia de nuestra aplicación.
- Router: contiene la lógica de navegación, para saber qué pantallas mostrar y cómo cambiar entre ellas. En algunos sitios se conoce también como wireframe



Ventajas e inconvenientes
Las ventajas de VIPER son las de cualquier arquitectura bien diseñada:

- Facilita la colaboración en el equipo de desarrollo, si cada desarrollador se ocupa de un componente separado o un conjunto de componentes.
- Facilita el mantenimiento de la aplicación.
- Hace posible el testing.

Como inconveniente principal está la sobrecarga que supone crear un mínimo de 5 componentes por cada módulo. Es una arquitectura que para aplicaciones pequeñas o para aplicaciones implementadas por un solo desarrollador quizá presenta una complicación excesiva.

MVVM:

este modelo de diseño, conocido como patrón Model-View-ViewModel. Tiene como finalidad llevar a cabo la separación del apartado de la interfaz de usuario, también conocida como View, de la parte lógica o Model.

De modo que el objetivo de este patrón de arquitectura es el de hacer completamente visible el aspecto visual del sistema.

Cabe destacar que, para implementar el patrón MVVM, es necesaria la comprensión de la forma de

- factorizar el código de las apps en las clases correctas
- tener conocimientos sobre su interacción con los elementos de diseño.

Así pues, el patrón MVVM en Swift funciona separando la aplicación en tres capas. Estas son la capa lógica del negocio, la lógica de presentación y la interfaz gráfica.