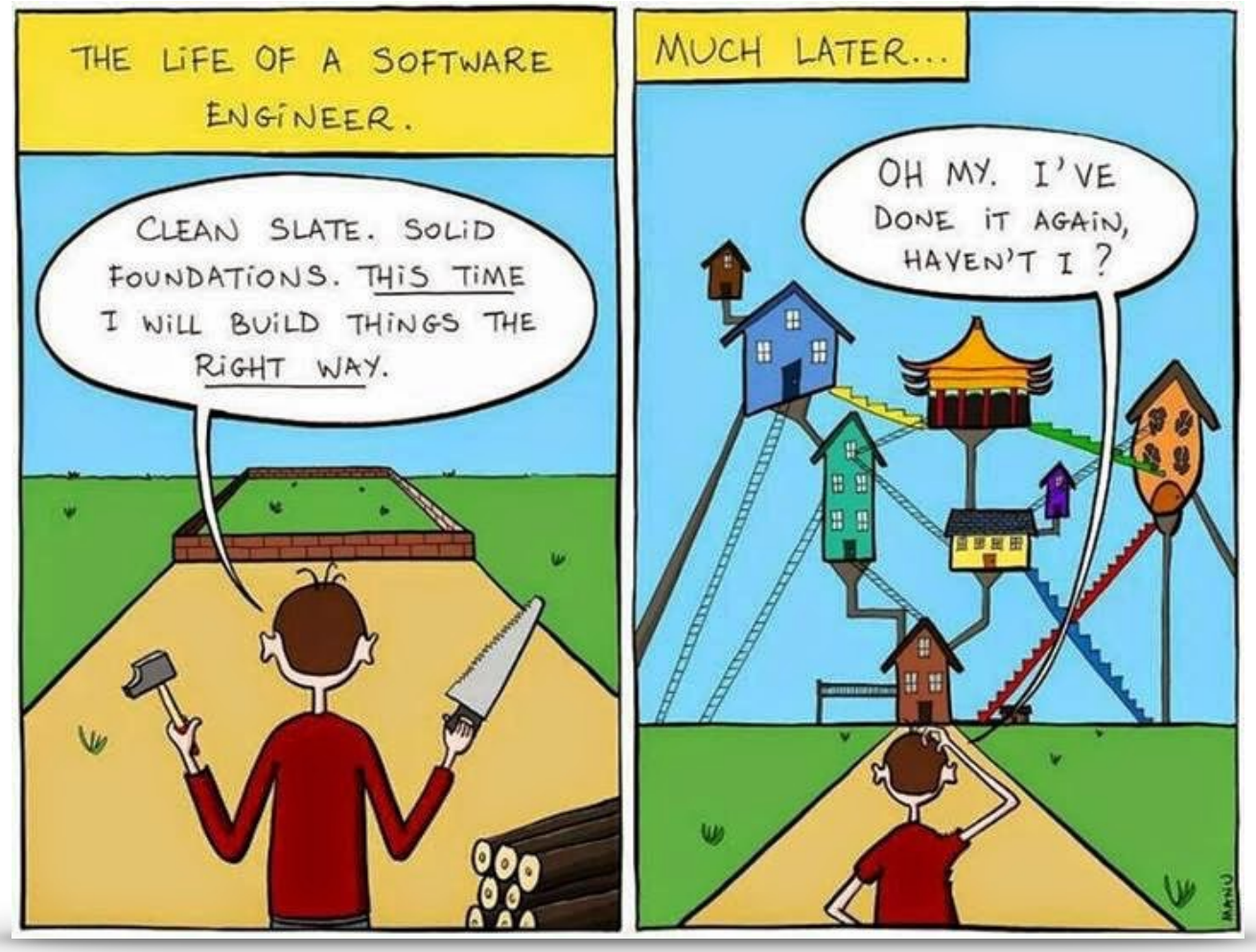
Fundamentos - HackerBooks

Objetivos de la práctica

El principal objetivo de esta práctica es acertar en la arquitectura de la App. Es decir, aplicar correctamente los principios del MVC y los mecanismos de comunicación que debemos usar entre los objetos, dependiendo del papel que juega cada uno.

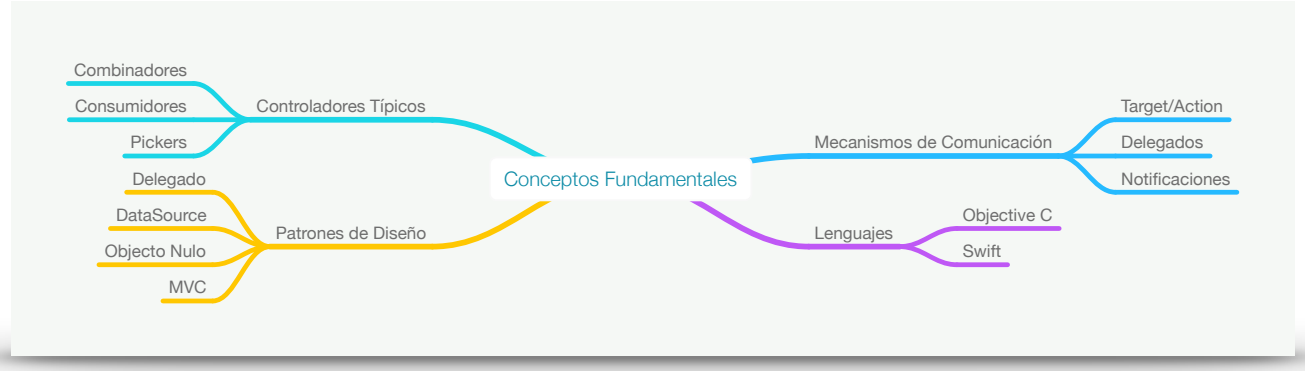
No se trata de hacer grandes viguerías, sino de algo muchísimo más importante: no hacer chapuzas. Tenemos que crear unos cimientos sólidos sobre los que construir funcionalidades más complejas, sin que por por ello nuestro código se vuelva imposible de mantener.



En resumidas cuentas, el objetivo es NO perpetrar atrocidades. ;-)

Esto se consigue siguiendo a rajatabla el MVC y los mandamientos de la comunicación entre los objetos.

Los Conceptos Fundamentales



Para sacar adelante la práctica tenemos que tener claros algunos de los conceptos fundamentales para desarrollar para iOS, muchos de los cuales se han visto en el curso (otros son parte del Avanzado o del curso de Swift).

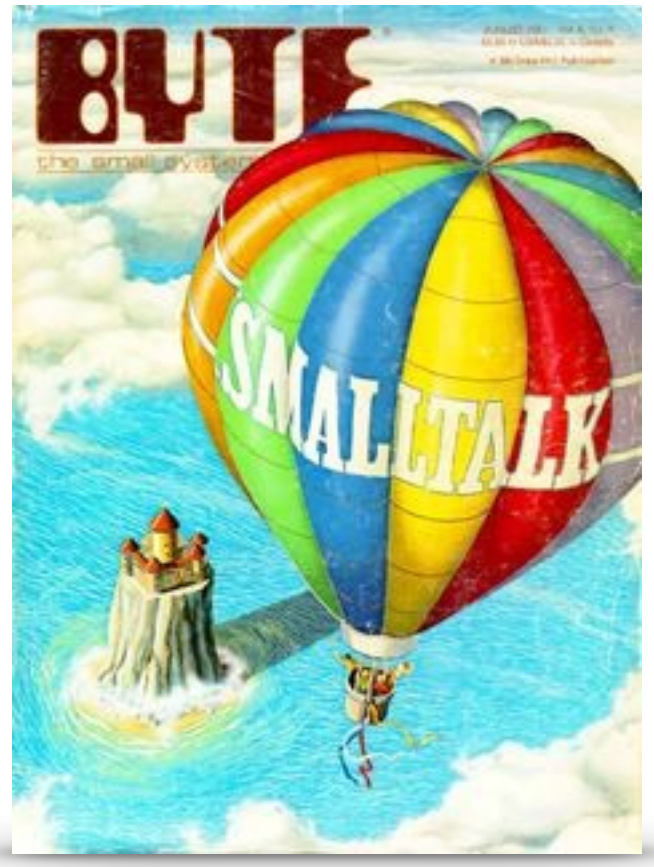
En concreto, pondremos en práctica

- los *Combinadores* (controladores que combinan a otros, como el UINavigationController o el UISplitViewController),
- los *Consumidores* (controladores que reciben los datos que han de mostrar de forma controlada a medida que los necesitan, como por ejemplo los UITableViewController.

Sin embargo, de lejos *lo más importante es el MVC y los mecanismos de comunicación*. Aplicar eso bien, implica tener luego un código ordenado, fácil de mantener, ampliar y testar.

Las Sagradas Escrituras del MVC

Antes de seguir, recordemos un poco el MVC, clave de toda esta práctica. Como casi todo de interés en informática, el MVC fue creado por programadores de Smalltalk.

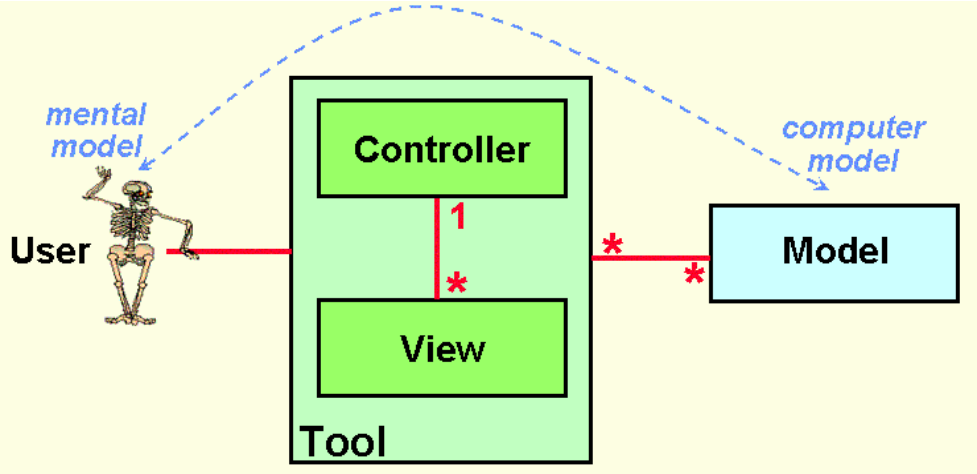


En concreto, fue ideado por un programador de Smalltalk llamado Trygve Reenskaug a finales de los 70 y consiste en *unir el modelo mental que tiene el usuario de una aplicación con el modelo que habita en el ordenador*.

Es un puente entre el usuario y el ordenador.

Un Ejemplo de un programa de Bolsa

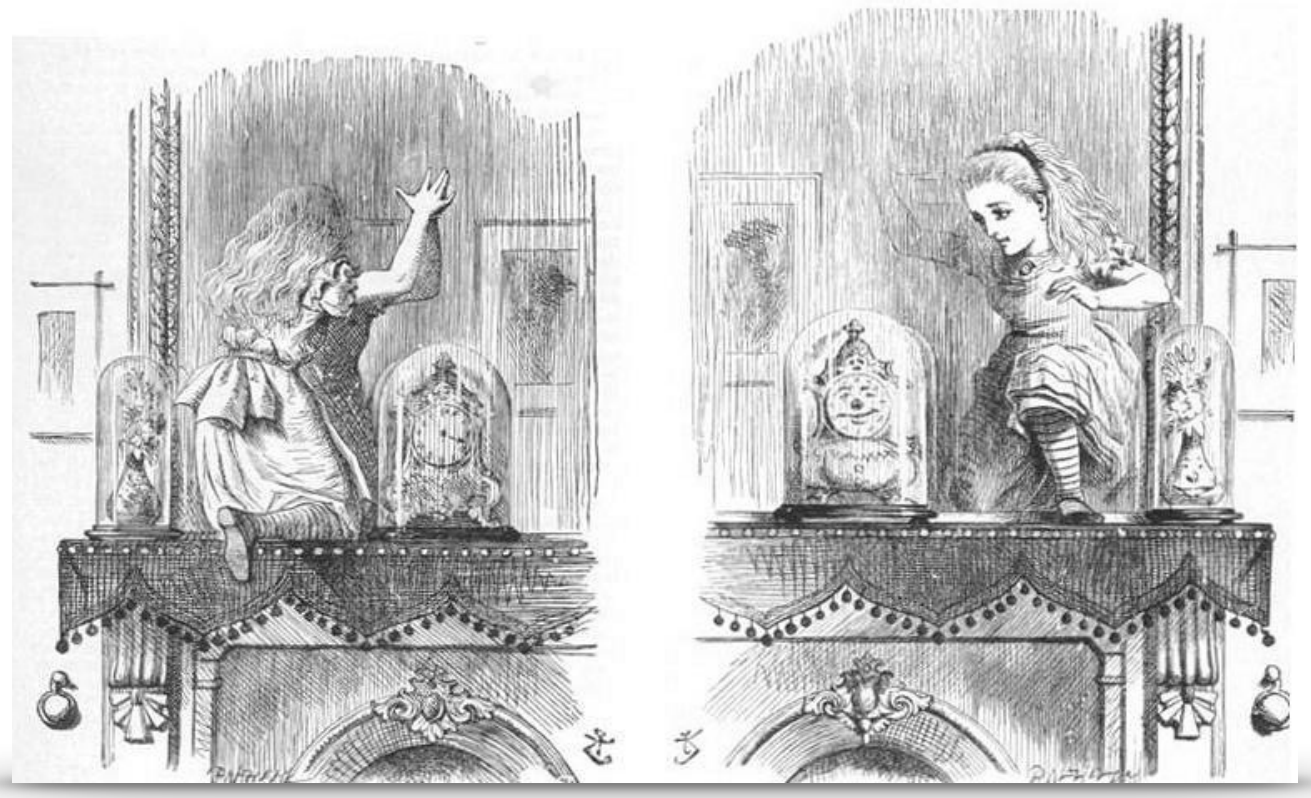
El usuario tiene un modelo mental de cómo funciona el mercado y en una aplicación bien diseñada, debería de dar la impresión de estar manipulando directamente los componentes de dicho modelo: acciones, derivados, órdenes de compra, órdenes de venta, volumen de contratación, etc…



El MVC es lo que permite mantener esa ilusión.

Por lo tanto, el MVC es una parte vital de la experiencia de usuario a ambos lados del espejo:

- Facilita al usuario de la aplicación la utilización de la misma.
- Facilita al usuario del código la gestión y creación del mismo.



Modelo

I tried to picture clusters of information as they moved through the computer. What did they look like? Ships? motorcycles? Were the circuits like freeways? I kept dreaming of a world I thought I'd never see. And then, one day…I got in.
Kevin Flynn - TRON Legacy

El modelo es lo que estamos simulando dentro del ordenador y representa conocimiento: un mercado, un sistema contable, un castillo poblado de nazis, el universo de StarWars o una biblioteca. Puede ser un objeto o un sistema complejo de varios objetos relacionados entre sí.

Es a menudo la clave de la aplicación: un sistema con un modelo bien diseñado será fácil de mantener, entender y expandir. Por el contrario, si el modelo está mal, todo lo que construyamos sobre él tendrá que estar lleno de remiendos. Son los cimientos y la estabilidad de toda la estructura depende de ellos.

La clave está en asegurarse de que todo cambio que sufra el modelo (no importando quién lo cause) sea notificado al resto del sistema mediante notificaciones. Aunque se puede usar el delegado, el modelo suele tener varios objetos pendientes de él (normalmente controladores) y la única forma de tenerlos a todos informados es mediante notificaciones.

Vistas

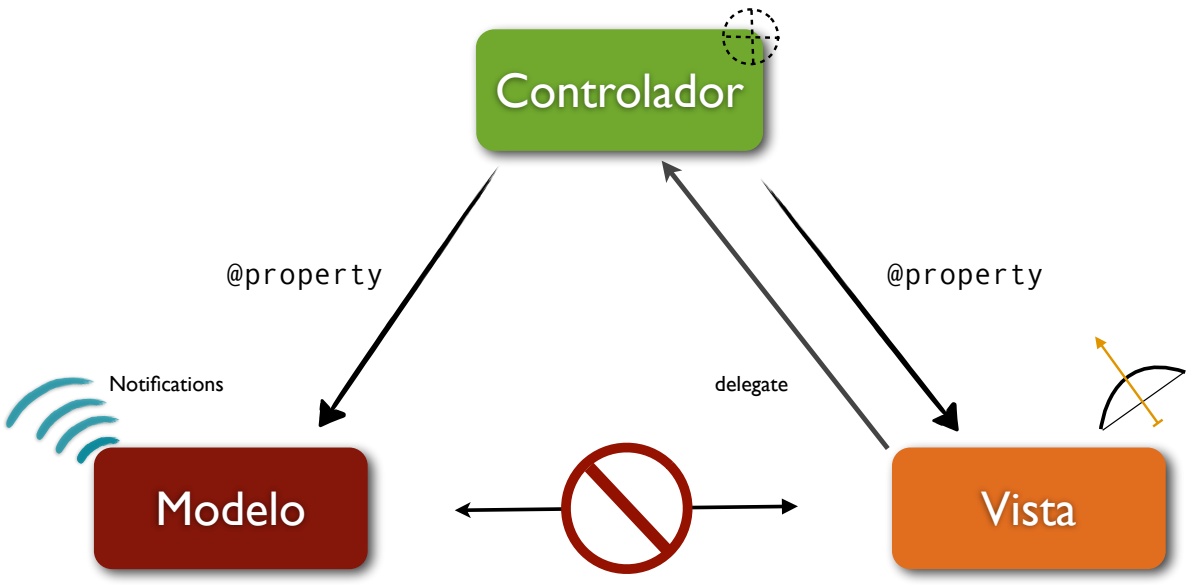
Son una representación visual del modelo y el mecanismo que permite al usuario manipular el modelo.

No se trata de una representación fidedigna del modelo, a veces pueden mostrar tan solo algunos aspectos del modelo o modificados de alguna manera. Es decir, existe un filtro entre la vista y el modelo.

La forma que tienen las vistas de comunicar las acciones del usuario es mediante target/action (para eventos sencillos) y delegado (para secuencias de eventos en serie, de tipo should->will->did).

Controlador

Es el filtro entre la vista y el modelo. El controlador decide qué parte del modelo se expone a las vistas y cómo el “input” del usuario llega al modelo.



El controlador:

- avisa al modelo de los cambios que el usuario ha pedido (usando las vistas).
- recibe (via notificaciones) el aviso de los cambios que el modelo ha decido hacer en respuesta a esas peticiones.

Un error muy común

Es muy común que en un método del controlador en el que se modifica algo del modelo, se haga también la actualización de la vista.

NO LO HAGAS. Es incorrecto y causará que tu código sea más complejo y más limitado.

Pongamos un ejemplo concreto: una orden de compra en un programa de bolsa. ¿Quién te asegura que se podrá llevar a cabo? ¿Cómo sabes cuando se hará efectiva?

Moraleja: el controlador envía la orden de compra al modelo y espera a recibir una notificación sobre el resultado. Ya contestará el modelo…o no, es asunto suyo. Cada componente tiene sus responsabilidades y es vital respetar las de cada uno.

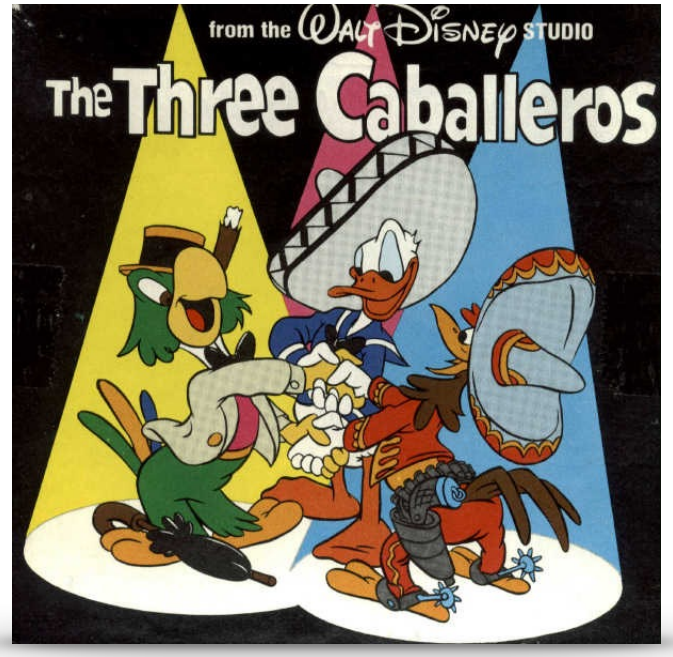
Resolución de la práctica

Para la correcta resolución de la práctica, es vital plantear bien el modelo. Una vez hecho esto, todo lo demás es coser y cantar.

Las especificaciones de la práctica tienen 3 partes problemáticas, que parecen difíciles de diseñar:

- **La Latencia**: los datos están en remoto y el obtenidos tardará una cantidad de tiempo perceptible por el usuario.
- **Los Tags**: son cadenas de texto, pero sin embargo se tienen que ordenar de una manera peculiar. Los favoritos deben de aparecer primero y los demás en orden alfabético.
- **La Relación entre Libros y Tags**: Un libro puede tener varios tags y un tag puede tener varios libros asociados. Con un `NSDictionary` podemos relacionar un objeto con otro (un `NSDictionary` no es más que un conjunto de parejas de objetos); sin embargo tenemos que relacionar varios objetos con varios objetos. No está claro qué estructura de datos debemos usar.

En cuanto tengamos dominados a estos tres caballeros, la práctica está resuelta y bien resuelta.



La Latencia

Aunque es de vital importancia, en este caso no nos debe de preocupar mucho, ya que aun no hemos visto cómo ejecutar código en segundo plano. Sin embargo hay varias manera de resolverlo con lo que ya sabemos. Hay clases que ocultan el hecho de que algo se está haciendo en segundo plano y se limitan a avisar de cuando la tarea ha terminado usando el patrón del delegado. Eso sí que podemos hacer.

Hay varias opciones:

- `NSURLConnection`. Está un poco desfasada y tiene muchos bugs. Apple la está sustituyendo por `NSURLSession`.
- `NSURLSession`. Es la versión mejorada de `NSURLConnection` aunque algo más compleja de manejar. Se ve con todo detalle en el curso avanzado.

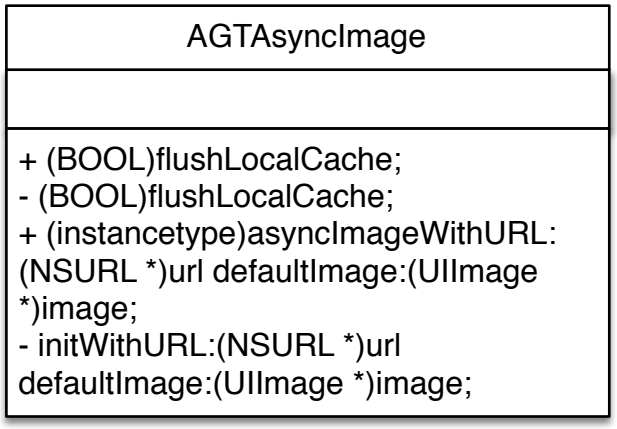
Aunque no es vital manejar la latencia en esta práctica, sería interesante que las imágenes de los libros apareciesen primero como una imagen genérica y pasado un tiempo (cuando se haya descargado la real), se actualice automágicamente.

Para ello podríamos haber diseñado una clase como `AGTAsyncImage`.

AGTAsyncImage

El comportamiento que queremos es algo que empieza con una imagen por defecto y cuando tiene la definitiva, se actualiza. Posteriormente, no quiero que esa imagen se vuelva a descargar, así que se tendrá que guardar en algún caché local.

`AGTAsyncImage` hace precisamente eso:



Se inicializa con una imagen por defecto y si dicha imagen no está en disco, se la descarga en segundo plano, la guarda en disco, actualiza su propiedad image y avisa del cambio mediante el protocolo de delegado.

También permite eliminar el caché local creado por una instancia o todas las imágenes guardadas por objetos de la clase. Para ello tenemos métodos de instancia y de clase, respectivamente.

Notifica de la llegada de la imagen definitiva mediante un protocolo de delegado:

```
@protocol AGTAsyncImageDelegate <NSObject>

/** The image property has a new value after downloading
 the remote one.
 */
-(void) asyncImageDidChange:(AGTAsyncImage*) aImage ;


@end
```

En este repositorio tenéis un ejemplo de uso de AGTAsyncImage.

Pórtala a Swift.

Con esto resolvemos la parte más compleja de la latencia: el descargar imágenes sin hacer que el scroll de la tabla se vea afectado. Para la descarga del `JSON` o del PDF, podemos usar `NSURLSession`, `NSURLConnection` o simplemente ignorar eso en esta práctica (no es vital al ser cosa del curso avanzado).

Los Tags

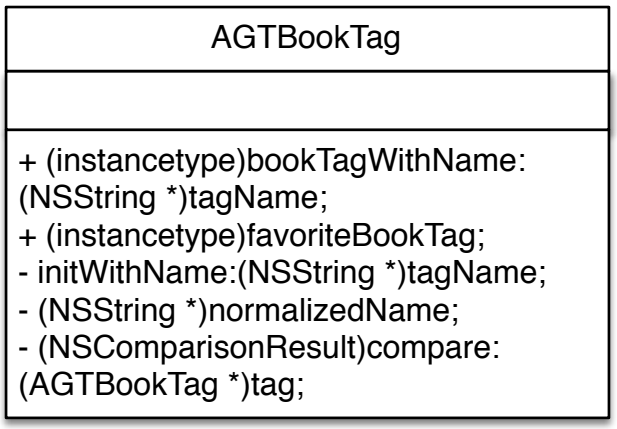
All problems in computer science can be solved by another level of indirection.
Butler Lampson

Los tags representan un problema por dos razones: su ordenación es rara y luego están los favoritos, que se comportan como una especie de de tag que a veces un libro tiene y a veces no.

El problema de los favoritos es fácil de resolver: “favorite” pasa a ser un tag más, que un libro podrá tener o no. Al igual que los demás tags, éste podrá tener varios libros asociados.

Luego tenemos el problema de la forma extraña en que se tienen que ordenar los tags. La forma más sencilla de resolver esto es la de añadir una nueva capa de abstracción (o nivel de indirección, si de verdad quieres hablar raro como el amigo Butler Lampson). Claramente `NSString` no nos atiende bien como modelo de un tag, así que inventaremos una clase nueva llamada `AGTBookTag`.

`AGTBookTag` tiene dos grandes tareas: asegurarse de que los tags son únicos independientemente de su “caja”. Es decir, Algorthims y ALGorithiMs son lo mismo, y debería de tener una representación normalizada. Quiero que todo nombre de tag empiece por mayúscula la primera palabra y el resto sea minúsculas. Para eso tenemos el método `normalizedName`.



Además, y esto es muy importante, debe de saber ordenarse.

En Swift, para que un objeto sepa ordenarse, debe de implementar el protocolo Comparable. Implementa esta clase en Swift. Tienes un ejemplo de comparable en los personajes.

Lo único que tenemos que hacer nosotros es implementar el protocolo Comparable que hace precisamente lo que nos pide la especificación:

- Favorite gana a todos.
- Los demás se ordenan como cadenas, de forma lexicográfica.

Referencias:

- Chapter 17: Another Level of Indirection in “Beautiful Code: Leading Programmers Explain How They Think”

La Relación entre los Libros y los Tags

(…) se acercó a un anciano inofensivo entretenido con su cerveza y le abrió el cráneo de un porrazo: cuando le preguntaron por qué había atacado al viejo, contestó : «¿Qué pasa? Llevaba cuarenta y nueve golpes con esta porra y me faltaba una marca para los cincuenta».
José Luis Borges — Historia de la Infamia

Es la última muesca que nos falta en nuestra porra y con ella podremos hundirle el cráneo a cualquier bug se ponga por delante.

Necesitamos relacionar tags y libros de la siguiente manera:

1. Un tag puede tener varios libros
2. Un libro puede tener varios tags.
3. Si a un libro que ya tiene un tag, le vuelvo a poner el mismo tag, no debería de pasar nada (no quiero estar comprobando si un libro tiene o no cierto tag).
4. Si un tag ya tiene un libro y le vuelvo a a asignar el mismo libro, no debería de pasar nada (no quiero estar comprobando si un tag tiene o no cierto libro).
5. Si a un libro le asigno un tag que no existía (uno nuevo), se tiene que crear una entrada para ese tag con ese libro asociado.
6. Si a un libro le elimino un tag, y ese libro era el último que lo tenia, quiero que ese tag sea eliminado.

Estas últimas dos condiciones son importantes para la gestión del tag “favorite”. Si a un libro se le asigna ese tag y es el primer favorito, la sección Favorites en la tabla debe de aparecer. De la misma manera, si se le quita el tag de favorito al último de los libros favoritos, la sección debe de desaparecer.

Todo esto podríamos hacerlo con un algoritmo complejo y con varios casos especiales que habría tener en cuenta y testar. **No lo haremos**.

"Tiran más dos estructuras de datos que 30 algoritmos"

Siempre que creas necesitar un algoritmo complejo, *busca primero alguna estructura de datos que lo resuelva de forma implícita*. Un excelente ejemplo es la búsqueda binaria. Aunque aparentemente sencillo, es un algoritmo sorprendentemente difícil de implementar bien.

En vez de usar una estructura de datos inadecuada (un array) y luego implementar un algoritmo complejo para buscar datos, es preferible usar una estructura de datos como un árbol binario, que implementa de forma implícita el algoritmo de búsqueda binaria.

La forma de organizar datos en un árbol binario resuelve de forma automática el algoritmo de búsqueda binaria.

Busquemos entonces si hay una estructura de datos que resuelva nuestro problema sin necesidad de algoritmos complejos y proclives a bugs.

Un `Dictionary` crea parejas de objetos (objeto clave y objeto valor). Sin embargo, eso no nos sirve, ya que no tenemos una relación de uno a uno sino de muchos a muchos.

Aunque no nos sirve, pero podemos echar un vistazo a las demás colecciones que tenemos en Cocoa, a ver si hay algo que se pueda aprovechar:

- `Set`: representan conjuntos únicos de objetos y tienen las propiedades de un conjunto en matemáticas. Tienen la característica interesante que un objeto no puede estar dos (o más) veces dentro de un conjunto. Si intentas meterlo una segunda vez, no pasa nada.
- `NSCountSet`: Similar a los anteriores, pero cuando intentas meter un objeto que ya está, le aumenta un contador. Es decir, almacena dos cosas: un objeto y un contador de cuantas veces lo has metido.
- `NSCache`: Similar al diccionario pero no guarda para siempre los objetos que tiene almacenados. Bueno para hacer cachés, como el nombre indica.

UN DICCIONARIO "MÁGICO"

Nada de lo que viene de serie parece resolver nuestro problema, pero fácilmente podemos pensar en un diccionario "mágico" que lo haga.

Tendría las siguientes características:

1. Asocia objetos clave con conjuntos de objetos. Es decir, bajo una misma clave podemos tener varios objetos, metidos en un "cubo".
2. Podemos añadir y sacar objetos del "cubo" que cuelga de una clave.
3. Si añadimos un objeto a una clave que aun no existe, no nos dará un error, sino que hará el favor de crear la clave, su "cubo" correspondiente y poner el objeto ahí dentro.
4. Si sacamos el último objeto de un "cubo", la pareja clave-"cubo vacío" se elimina.

Pues bien, esa estructura de datos existe y se llama un MultiMapa. Aunque no disponemos de ella de forma nativa, podemos implementarla de forma muy sencilla con un `Dictionary` y `Set`.

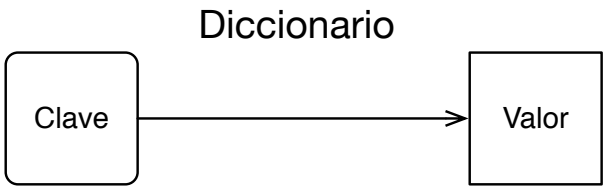
AGTMultiDictionary
+ (instancetype)dictionary; - (void)addObject:object forKey:key; - (NSSet *)objectsForKey:key; - (void)removeObject:object forKey:key; - (NSUInteger)count; - (NSArray *)allKeys;

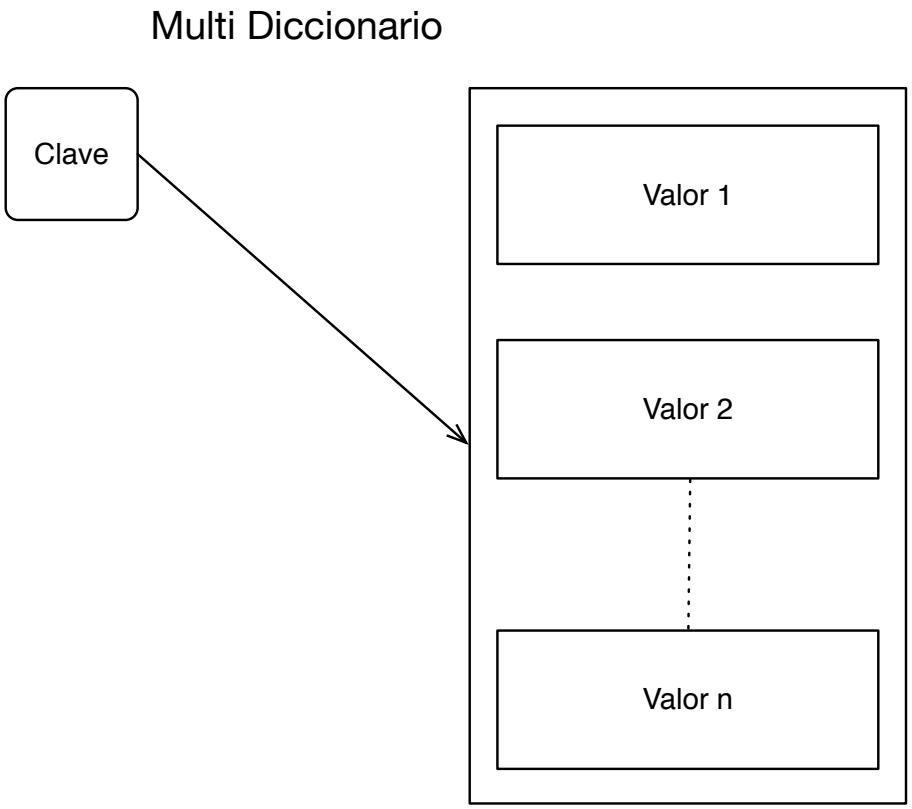
Un `AGTDictionary` asocia claves con `NSSets` de objetos.

Un mismo objeto puede estar bajo varias claves (un libro puede tener varios tags).

La propiedad `count` me devuelve el número de objetos únicos dentro del `AGTDictionary`. Es decir, que si un libro está en 20 tags, solo se le cuenta una vez.

El método `allKeys` me devuelve un `NSArray` con todas las claves.





Aquí tenéis la implementación en Objective C. Pórtala a Swift.

```
//
//  AGTMultiDictionary.h
//  HackerBooks
//
//  Created by Fernando Rodríguez Romero on 11/04/15.
//  Copyright (c) 2015 Agbo. All rights reserved.
//
/**
 Contains a set of objects for each key.
 */
#import <Foundation/Foundation.h>

@interface AGTMultiDictionary : NSObject

+(instancetype)dictionary;

-(void) addObject:(id) object forKey:(id<NSCopying>) key;
-(NSSet *) objectsForKey:(id<NSCopying>) key;

-(void) removeObject:(id)object forKey:(id<NSCopying>)key;

-(NSUInteger) count;
-(NSArray*) allKeys;

@end

//
//  AGTMultiDictionary.m
//  HackerBooks
//
//  Created by Fernando Rodríguez Romero on 11/04/15.
//  Copyright (c) 2015 Agbo. All rights reserved.
//

#import "AGTMultiDictionary.h"

@interface AGTMultiDictionary ()
@property (nonatomic, strong) NSMutableDictionary *dict;
@end

@implementation AGTMultiDictionary

#pragma mark -  Class Methods
+(instancetype)dictionary{
    return [[self alloc] init];
}

#pragma mark - init
-(id) init{
    if (self = [super init]) {
        _dict = [NSMutableDictionary dictionary];
    }
    return self;
}
```

```
#pragma mark -  accessors
-(void) addObject:(id) object
          forKey:(id<NSCopying>) key{

    NSMutableSet *objs = [self.dict objectForKey:key];
    if (!objs) {
        objs = [NSMutableSet set];
    }

    [objs addObject:object];
    [self.dict setObject:objs
                  forKey:key];
}

-(NSSet *) objectsForKey:(id<NSCopying>) key{

    NSMutableSet *objs = [self.dict objectForKey:key];
    if (!objs) {
        objs = [NSMutableSet set];
    }

    return objs;
}

-(void) removeObject:(id)object
              forKey:(id<NSCopying>)key{

    NSMutableSet *objs = [self.dict objectForKey:key];
    if (objs) {
        [objs removeObject:object];
        if ([objs count] > 0) {
            [self.dict setObject:objs
                          forKey:key];
        }else{
            // ELiminamos ese set y su key del diccionario
            [self.dict removeObjectForKey:key];
        }

    }
}

-(NSUInteger) count{
    /**
     The number of unique objects in all the buckets
     */
    NSMutableSet *total = [NSMutableSet set];
    for (NSMutableSet *bucket in [self.dict allValues]) {
        [total unionSet:bucket];
    }
    return total.count;
}

-(NSArray*) allKeys{
    return [self.dict allKeys];
}
```

```
#pragma mark - NSObject
-(NSString*) description{
    return [NSString stringWithFormat:@"<%@:\n %@>", [self class], _dict];
}

@end
```

Cacharrea con ella, entiéndela y pórtala a Swift.

Si antes el modelo de la App se nos hacía complicado, era porque teníamos las herramientas inadecuadas.

Con estas 3 clases ya contamos con las herramientas correctas y podemos abordar el modelo de manera que sea sumamente fácil su implementación.