amsellem.yves@gmail.com

Introduction à la programmation sur iPhone

Partie 1. Langage, interface & contrôles standards

MCours.com

La mobilité à porté de main

Avec l'App Store, première boutique d'applications tierces sur mobile, l'iPhone d'Apple a donné un joli coup de pied dans la fourmilière de la mobilité. Prenons un moment afin d'explorer les possibilités offertes à tout développeur - **Objective-C** et **Cocoa Touch** - via la réalisation d'une machine à sous. Cette application sera composée de 5 disques qu'un bouton "*Insérer une pièce*" fera tourner. Aligner 3 fruits ou plus sur ces disques affichera le prix remporté (ici, une peluche).

Un Mac Intel sous Léopard et un Apple ID (mail/password) sont obligatoires afin d'avoir accès à l'IDE **xCode** et au **SDK 3.x**, sur http://developer.apple.com/iphone. Proposés gratuitement, ces outils permettent le développement, le test sur simulateur et l'optimisation de performances. La distribution - sur son propre iPhone/iPod Touch et sur l'App Store - devant faire l'objet de l'acquisition d'une licence annuelle.

Figure 1. L'écran de l'application que nous nous apprêtons à créer

Objective-C, notions essentielles

En Objective-C, chaque classe se compose de deux fichiers aux noms identiques, un `.h` et un `.m`. Le `.h` déclare les propriétés et méthodes de la classe ainsi que sa classe mère et les différents protocoles qu'elle implémente. Le `.m` implémente les déclarations du `.h`.
Les `.h` jouent deux rôles en Objective-C, interface ou protocole. Une `@interface` est la partie déclarative d'une classe et doit être accompagnée d'une unique implémentation du même nom (comme vu ci-dessus). Un `@protocol` est une liste de méthodes et peut être implémenté par plusieurs classes.

Les protocoles, en plus de permettre l'utilisation du polymorphisme - quand plusieurs classes les implémentent -, sont utilisés abondamment par les composants standards afin de déléguer leur comportement aux classes qui les utilisent. Dans l'exemple qui va suivre, la roue de fruits ne sera pas une classe dérivée de la classe standard idoine - `UIPickerView` - dont les méthodes auraient été surchargées, il s'agira d'un `UIPickerView` classique dont les protocoles - `UIPickerViewDataSource`, `UIPickerViewDelegate` - seront **implémentés par le contrôleur l'utilisant.** La délégation est ainsi préférée à l'héritage.

Les éléments standards de ce type (qui nécessitent l'implémentation de protocoles particuliers afin de fonctionner) déclarent cette nécessité dans leur interface. Sans implémentation dans l'appelant, rien n'arrive.

Création du contrôleur principal

Une fois installé, xCode est accessible dans */Developer/Applications*. Commençons par créer un nouveau projet via le menu **File > New Project...**. Dans la colonne **iPhone OS > Application** sélectionnons **Window-based Application** et nommons-la `SlotMachine`. Bien que des modèles prédéfinis soient proposés, partons de zéro à des fins didactiques.

Le répertoire *Classes* contient par défaut deux éléments : `SlotMachineAppDelegate.h`, une interface, et `SlotMachineAppDelegate.m`, son implémentation. Le raccourcis ⌘⌥↑ permet de basculer de l'un à l'autre.

```
#import <UIKit/UIKit.h>

@interface SlotMachineAppDelegate : NSObject <UIApplicationDelegate> {
      UIWindow *window;
      GameViewController *gameViewController; // seul ajout de notre part, le reste est généré
}

@end
```

La classe créée par défaut par xCode, `SlotMachineAppDelegate`, est le point d'entrée de l'application. Elle hérite de `NSObject` et implémente le protocole `UIApplicationDelegate`. Lorsque la main est donnée à l'application, la méthode `applicationDidFinishLaunching` de ce protocole est appelée. Une instance de `GameViewController`, le contrôleur principal, sera créée lors de cet appel et sa vue, composée de 5 disques de fruits et d'un bouton "*Insérer une pièce*", initialisée automatiquement.

La création des 2 onglets, des 5 disques, du bouton et du label seront fait graphiquement à l'aide d'un autre outil, **InterfaceBuilder**. Pour le reste, l'implémentation de ces 2 classes suffira à faire fonctionner le jeu.

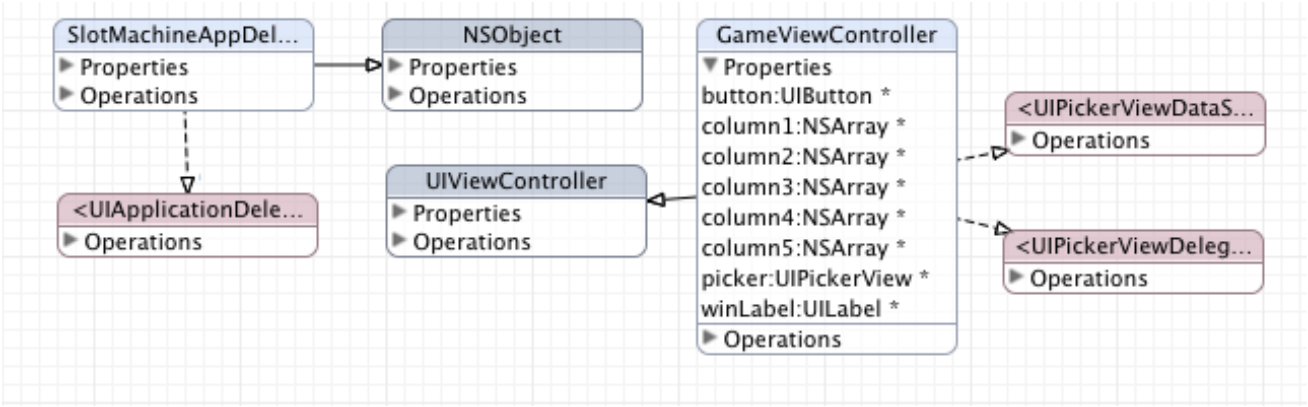

Figure 2. Les deux classes de notre application, `SlotMachineAppDelegate` et `GameViewController` (chacune dispose d'un fichier .xib décrivant sa vue au format xml, `MainWindow.xib` et `GameViewController.xib`).

Passons à l'implémentation de cette interface dans `SlotMachineAppDelegate.m` (la syntaxe du langage est détaillée juste après) :

```
#import "SlotMachineAppDelegate.h"

@implementation SlotMachineAppDelegate

- (void)applicationDidFinishLaunching:(UIApplication *)application {
      gameViewController = [GameViewController alloc];
      [window addSubview: gameViewController.view]; // ajouter la vue de gameViewController à window
      [window makeKeyAndVisible];                   // la fenêtre principale de l'application
}                                                   // suffit à lui passer la main

- (void)dealloc {
      [gameViewController release];
      [window release];
      [super dealloc];
}

@end
```

Ajouter la vue du contrôleur du jeu (`gameViewController`) à la fenêtre de l'application (via `addSubview:`) suffit à lui passer la main. Il en résultera l'affichage de la vue de ce contrôleur : `GameViewController.xib`, dotée de ses 5 disques de fruits.
Il est temps de créer ce contrôleur. Pour ce faire, cliquons droit sur le répertoire *Classes* puis **Add > New File**. Dans la liste **iPhone > Cocoa Touch Class** sélectionnons **UIViewControllerSubclass** et cochons **With XIB for user interface**. Nommons-le `GameViewController`.
Déplaçons le xib, fichier décrivant la vue du contrôleur ainsi créé, dans le dossier *Ressources*.

Point technique : syntaxe des appels de méthodes

En Objective-C, les appels de méthodes se font sous la forme `[object add:(int)value modulo:(int)mod]`. Tous les arguments de la méthode sont **nommés** et **typés**. L'appel à cette méthode s'effectue via la syntaxe `[object add:9 modulo:4]`. L'ordre des arguments est fixe.

Création de la vue du jeu

Le fichier décrivant graphiquement notre application, `GameViewController.xib`, est situé dans le répertoire *Ressources*. Double-cliquer dessus ouvre **InterfaceBuilder**, le pendant graphique à **xCode**. Interface Builder est composé de quatre fenêtres, **la librairie**, liste de tous les éléments d'interface, **l'inspecteur**, liste des propriétés de l'objet sélectionné, **la fenêtre xib** liste de tous les éléments du xib actuel et **la vue courante** affichée à la résolution de l'iPhone, 320 * 480.

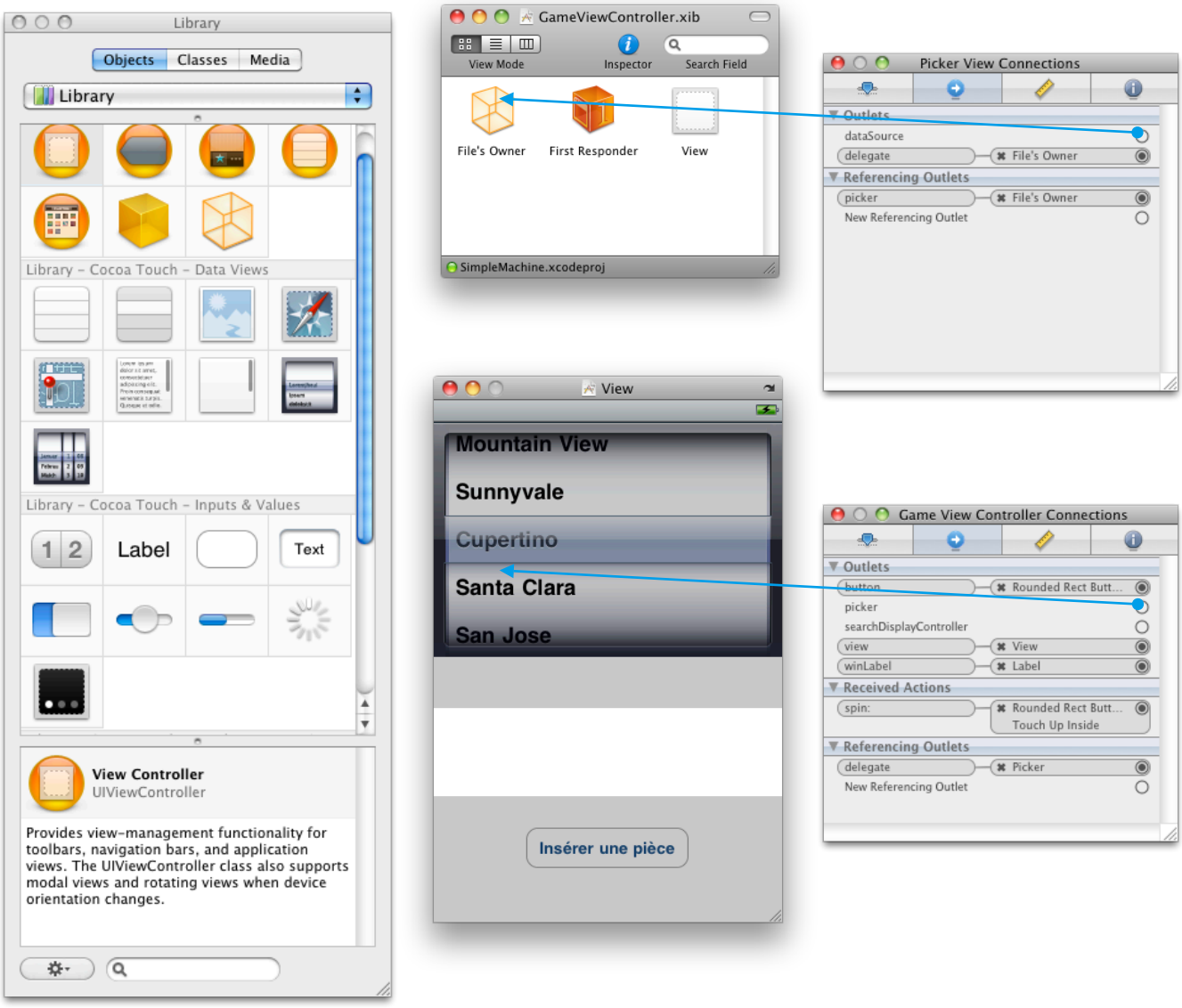

Figure 3. Interface Builder, l'outil permettant de construire l'interface

Nous allons **instancier graphiquement le picker de fruits** utilisé dans notre code, lui indiquer que le contrôleur actuel sera en charge de ses protocoles et le lier au code que nous avons précédemment écrit.

La vue de chaque contrôleur est matérialisée par un xib qu'Interface Builder peut éditer. Chacun des éléments (bouton, label...) ajoutés par ce biais devra être connecté à un objet du même type dans le code afin d'y être manipulé (cela permet, par exemple, de modifier le texte d'un label créé dans InterfaceBuilder).

Point technique : lier les objets de l'interface et du code

Lier les objets du code écrit dans xCode avec ceux positionnés dans Interface Builder est simple : sélectionner le *File's Owner* - classe liée par l'onglet 4 de l'inspecteur au xib édité actuellement - tirer l'anneau en face de ses objets sur les objets déposés dans la vue d'InterfaceBuilder.

Lier les actions marche dans le sens inverse, il faut commencer par sélectionner dans Interface Builder l'objet qui sera responsable de l'appel de la méthode, puis, via l'onglet 2 de l'inspecteur, sélectionner un évènement et relier son anneau à une méthode de *File's Owner*.

Les manipulations suivantes requièrent, au préalable, la création de l'interface `GameViewController.h` :

```
#import <UIKit/UIKit.h>
#define nbFruits 4
#define nbPickers 5

@interface GameViewController : UIViewController <UIPickerViewDataSource, UIPickerViewDelegate> {
      UIPickerView *picker; // l'élément principal du jeu : la roue à 5 disques
      UILabel *winLabel;    // le label indiquant le prix remporté
      UIButton *button;     // le bouton "insérer une pièce"
      NSArray *column1, *column2, *column3, *column4, *column5; // 1 tableau de fruits par disque
}

@property(nonatomic, retain) IBOutlet UIPickerView *picker;
@property(nonatomic, retain) IBOutlet UILabel *winLabel;
@property(nonatomic, retain) IBOutlet UIButton *button;

-(IBAction)spin:(id)sender; // id, le type de sender, est un pointeur vers tous type d'objet
                            // Tous les objets n'héritent pas de NSObject, id est plus large
@end
```

Ajoutons un `UIPickerView` de la librairie en haut de notre vue. Le positionnement est facilité par des pointillés de pose et l'aimant des bords. Posons également un `UILabel` et un `UIButton`. Renommons le bouton "*Insérer une pièce*" et décochons *User Interaction Enabled* du `UIPickerView` via l'inspecteur afin d'empêcher le joueur de sélectionner les fruits individuellement.

En sélectionnant *File's Owner* dans la fenêtre xib, l'onglet 2 de l'inspecteur laisse apparaître nos `picker`, `button` et `winLabel` - déclarés dans le code - suivis d'un anneau. Il suffit de tirer l'anneau - qui tendra alors un fil bleu - à partir du contrôleur et de le relier à l'icône des trois éléments graphiques créés à l'instant. Lions également le bouton à son action, en le sélectionnant puis, via l'onglet 2 de l'inspecteur, en tirant l'anneau de *Touch Up Inside* (qui signifie appuyer puis relâcher) et le déposant sur le *File's Owner* pour faire apparaître la méthode `spin:`.

Point technique : les outlets et les propriétés

Les `IBOutlet` et `IBAction` déclarés dans notre code peuvent être manipulés par InterfaceBuilder.

Indiquer `IBOutlet` sur la propriété d'un `UILabel` dans `GameViewController.h` permet de l'associer à un élément d'interface de `GameViewController.h` dans InterfaceBuilder. Ainsi il pourra être modifié (taille, couleur...) et positionné dans InterfaceBuilder à l'initialisation de la vue et modifié dans le code par la suite.
Indiquer `IBAction` lors de la déclaration d'une méthode dans `GameViewController.h` permet d'indiquer qu'elle pourra être déclenchée à l'appui d'un bouton de `GameViewController.xib`.

Redéfinir une variable avec `@property` permet de la manipuler à l'extérieure de la classe. Cette annotation déclare l'accès possible : `nonatomic`, par exemple, indique non thread safe. Elle fonctionne de paire avec `@synthesize` qui génère getters et/ou setters associés à l'exécution. L'accès aux variables qui se ferrait normalement `[variable setProperty:9]` peut désormais s'effectuer `variable.property = 9`.

Comme c'est couramment le cas, l'`UIPickerView` délègue au `ViewController` sa gestion et son alimentation en données. Après l'avoir sélectionné, nous devons tirer les anneaux *delegate* et *dataSource* de l'onglet 2 de l'inspecteur sur le *File's Owner* (`GameViewController`, le *File's Owner*, doit déclarer, comme c'est le cas ici, les protocoles liés afin que son implémentation des méthodes déléguées soient appelées automatiquement).

Programmation du jeu

Les éléments d'interface sont connectés au code : attaquons nous à la méthode `spin:`, la méthode appelée lors de l'appui sur "*Insérer une pièce*" dans `GameViewController.h` :

```
-(void)showButton {
	button.hidden = NO;
}

-(void)win:(NSNumber *)points {
	switch ([points intValue]) {
		case 3: winLabel.text = @"une peluche !";break;
		case 4: winLabel.text = @"une montre !"; break;
		case 5: winLabel.text = @"une voiture !";break;
		default:winLabel.text = @"perdu";
	}
	[self performSelector:@selector(showButton) withObject:nil afterDelay:.5];
}

-(IBAction)spin:(id)sender {
	int values[nbFruits] = {0}; // Initialisation du tableau
	for (int i = 0; i < nbPickers; i++)
	{
		int value = random() % nbFruits;
		values[value]++;
		[picker selectRow:value inComponent:i animated:YES]; // Déplacement du disque
		[picker reloadComponent:i];
	}
	int bestValue = -1;
	for (int i = 0; i < nbFruits; i++) {
		if (values[i] > bestValue)
			bestValue = values[i];
	}
	button.hidden = YES;
	NSNumber *number = [[NSNumber alloc] initWithInt:bestValue];
	[self performSelector:@selector(win:) withObject:number afterDelay:.5];
	[number release]; // Le selector prend un objet : attention à bien le libérer après
}

-(void)viewDidLoad {

	UIImage *grape = [UIImage imageNamed:@"raisin.png"];
	UIImage *pineapple = [UIImage imageNamed:@"ananas.png"];
	for (int i = 1; i <= nbPickers; i++)
	{
		// Afin d'économiser de la place et faciliter la lecture, seulement 2 des 4
		// UIImageView sont initialisées ici
		UIImageView *grapeView = [[UIImageView alloc] initWithImage:grape];
		UIImageView *pinView = [[UIImageView alloc] initWithImage:pineapple];

		NSArray *imageViewArray = [[NSArray alloc] initWithObjects: grapeView, pinView, nil];

		NSString *fieldName = [[NSString alloc] initWithFormat:@"column%d", i];
		[self setValue:imageViewArray forKey:fieldName];
		[fieldName release];
		[imageViewArray release];

		[grapeView release];
		[pinView release];
	}
	srandom(time(NULL));
}
```

La méthode `spin:` va itérer sur chaque colonne du `UIPickerView` et déterminer un fruit parmi les 4 présents. Via la méthode `random()` on détermine un entier de 0 à 3, on stocke son nombre d'occurrences dans un tableau et l'on met la colonne du `UIPickerView` à jour via l'appel à `reloadComponent:` en précisant que le changement n'est pas abrupte, mais animé. Ensuite, on détermine si une valeur est sortie 3 fois ou plus et on affiche le prix remporté par le joueur.

La méthode `viewDidLoad` est héritée de `UIViewController`. Elle est exécutée dès que le xib de l'interface est chargé. Elle charge les images que nous avons préalablement copiées dans le répertoire *Ressources*, les ajoute dans une `UIImageView` dont elle fait un tableau. Elle initialise finalement les tableaux `column1`, `column2`... avec.

Point technique : les selectors

Nous avons recours à de l'introspection via la méthode `[self performSelector:@selector(showButton) withObject:nil afterDelay:.5]` afin de temporiser l'appel à une méthode. Ainsi, le résultat ne s'affiche qu'après une demie seconde, laissant aux roues du `UIPickerView` le temps de tourner. Attention à la syntaxe, si la méthode à des arguments le caractère " : " lui est ajouté et l'argument précisé via `withObject:`.
L'appel à `[self setValue:imageViewArray forKey:fieldName]` dans la méthode `viewDidLoad` permet, également via de l'introspection, de positionner chacun des 5 `NSArray` (`column1`...) à la valeur passée en paramètre, un tableau d'`UIImageView`. Une de celles-ci est sélectionnée lors de l'appel `[picker selectRow:]`.

Implémenter le *delegate* et la *dataSource* du `UIPickerView`

Ajoutons à `GameViewController.m` l'implémentation des protocoles de `UIPickerView` :

```
#pragma mark Picker Data Source Methods
// #pragma facilite la navigation dans xCode

- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
      return nbPickers;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView numberOfRowsInComponent:(NSInteger)component
{
      return nbFruits;
}

#pragma mark Picker Delegate Methods
- (UIView *)pickerView:(UIPickerView *)pickerView viewForRow:(NSInteger)row
                forComponent:(NSInteger)component reusingView:(UIView *)view
{
      if (component == 0)
               return [self.column1 objectAtIndex:row];
      else if (component == 1)
               return [self.column2 objectAtIndex:row];
      else if (component == 2)
               return [self.column3 objectAtIndex:row];
      else if (component == 3)
               return [self.column4 objectAtIndex:row];
      return [self.column5 objectAtIndex:row];
      // Chaque colonne du UIPickerView à son propre tableau d'UIImageView
}
```

Ces deux protocoles `UIPickerViewDelegate` et `UIPickerViewDataSource` définissent plusieurs méthodes. La *dataSource* détermine les données gérées par chaque disque du `PickerView`. Nous fixons le nombre de composant à `nbPickers` et le nombre de lignes à `nbFruits`. Dans notre exemple, tous les éléments affichent la même chose, mais il arrive couramment que la *dataSource* fasse appel au modèle pour alimenter ses colonnes. L'implémentation du *delegate* indique quelle valeur choisir pour chaque ligne. Ces valeurs sont stockées dans les tableaux `column1`, `column2`...

Point technique : les différents objets du SDK

Nous avons utilisés de nombreuses chaines de caractères - `NSString` - au fil de nos exemples. Celles-ci se déclarent via l'utilisation de `@"texte"` et sont libérées automatiquement à l'exécution.
Il en va différemment des autres objets, `NSNumber`, `NSArray`, `UIImageView`... Contrairement au Mac, l'iPhone ne dispose pas de garbage collection. La gestion de la mémoire y est donc cruciale.
Par convention, toutes les méthodes laissent la charge de la gestion mémoire à l'appelant. Nous verrons dans la partie suivante comment cela se traduit sur le code.
Attention, `NSInteger` et `NSUInteger` (non signé) ne sont pas des objets. Ils permettent simplement de laisser le choix du 32 bits - 64 bits au compilateur. L'instanciation d'un objet sera donc nécessaire pour utiliser leur valeur dans une méthode nécessitant un objet (l'ajout à un `NSMutableArray`, par exemple).

Sauvons, compilons, la première partie du jeu fonctionne. Les roues tournent et indiquent le prix gagné.

Dans la partie suivante nous verrons comment, toujours à l'aide de la délégation et l'utilisation de protocoles, historiser les parties gagnantes dans une `UITableViewController`.