Développer pour iPhone : comment démarrer, utiliser les outils et mettre en place un modèle MVC

Antony Gardez
Élève-ingénieur à l'INSA de Rouen
antony.gardez@insa-rouen.fr

21 juin 2009

Résumé

Ce document a pour but d'aider les personnes n'ayant jamais développé pour les plateformes Apple à se familiariser avec l'IDE Xcode, l'outil Interface Builder, le langage Objective-C et le framework Cocoa Touch. Il est rédigé dans le cadre d'une option prise dans la formation ASI de l'INSA de Rouen. L'accent sera mis sur le développement pour iPhone bien que développer pour Mac OS X soit très similaire.

1 Introduction

Apple a sorti l'iPhone 3G le 17 Juillet 2008 en France, et l'opérateur Orange en a vendu plus d'un million à l'heure actuelle. En plus d'apporter la 3G, cette nouvelle version a permis aux développeurs le souhaitant de fournir des applications, développées à l'aide de l'iPhone SDK, pour cette plateforme, par le biais de l'AppStore.

L'iPhone SDK permet d'expérimenter un nouvel environnement, autant en termes d'Interface Homme Machine, de langage de programmation que de modèle de distribution des applications.

Dans cet article, mon but est d'aider un programmeur connaissant la Programmation Orientée Objet (POO) à se lancer dans le développement pour iPhone ou Mac. Cet article ne couvre que quelques notions et ne saurait en aucun cas remplacer les tutoriels officiels d'Apple, qui sont d'ailleurs très complets, disponible sur l'iPhone Dev Center (`http://developer.apple.com`). Pour développer avec le SDK de l'iPhone, il est obligatoire d'avoir un ordinateur Apple équipé de Mac OS X.

Dans un premier temps nous verrons les outils mis à disposition par Apple pour développer pour iPhone, puis nous verrons comment fonctionne le langage objective-C combiné au framework Cocoa, pour enfin comprendre la mise en place le Design Pattern MVC et illustrer toutes ces notions avec un exemple de mise en place d'une vue permettant l'insertion par l'utilisateur de noms de joueurs.

2 Xcode

Xcode est l'IDE développé par Apple, qu'ils utilisent eux-mêmes pour développer leurs applications. Il s'adapte à l'objective-C mais également au C, au Java et autres langages. Comme la plupart des IDE, il contient un outil de debug utilisant *gdb* et compile ses sources à l'aide de *gcc*, qui lui permet dans le cas de la programmation pour Mac de construire des exécutables pour processeurs Intel et PowerPC. Il est livré avec Mac OS X et contient également des outils d'analyse des performances ainsi qu'Interface Builder, l'outil de conception d'interface que nous utiliserons dans ce document et qui est détaillé dans la section 5.

Le développement d'une application iPhone à l'aide de l'outil XCode peut se dérouler de la manière suivante :

- Création du projet. Il est possible de choisir un template qui défini un type d'application bien défini parmi les suivants :
  - Navigation-based application,
  - OpenGL,
  - Tab bar,
  - View based,
  - Utility (widget),
  - Window-based (base de tous, contient le *main* ainsi que les fichiers nécessaires au démarrage d'un projet).
- Code dans l'IDE après avoir fait de la modélisation. L'IDE permet l'auto-complétion, le refactoring, mais il est également possible d'afficher une petite fenêtre qui reste au premier plan et donne la documentation du symbole situé à l'emplacement du curseur d'édition. En terme de gestion de projet, il est possible d'utiliser directement des gestionnaires de version tels que subversion, cvs ou perforce. Une autre fonctionnalité est appelée Snapshot et permet d'enregistrer l'état actuel des sources d'un projet rapidement afin de pouvoir y revenir plus tard en cas de mauvaise manipulation,
- Définition des vues et des contrôleurs. Les vues peuvent être définie dans des classes ou utiliser l'outil Interface Builder, afin de placer les éléments de la vue de manière graphique,
- Test sur le simulateur iPhone directement sur l'écran de l'ordinateur,
- Test sur un iPhone. Si ce dernier est relié à l'ordinateur, il est possible d'utiliser des outils de débug ainsi que visualiser les performances de l'application par le biais des outils *Shark* et *Instruments*,
- Distribution de l'application. Lorsque votre application est prête, il existe deux modes de diffusion : via l'appStore où en mode ad hoc. Le mode ad hoc permet de diffuser une application directement à des personnes, avec un maximum de 100 iPhones.

Les deux dernières étapes de ce cycle nécessitent une adhésion au programme de développeur iPhone d'Apple, qui coûte 80€ par an et donne accès à la diffusion d'application sur l'appStore, soit en choisissant de la distribuer gratuitement ou avec un prix défini. Les pays dans lesquels l'application sera disponible peuvent être définis lors de la mise en ligne de celle-ci. Par défaut, c'est la diffusion dans tous les appStore du monde qui est sélectionnée.

3 L'objective-C

L'objective-C est un langage de programmation orienté objet et réflexif. Apparu quelques années après le C++, au début des années 80, il est également basé sur le C ANSI mais diffère du C++ par sa vision de l'envoi dynamique de messages empreinté au langage SmallTalk. Utilisé à l'origine dans le système d'exploitation NeXTSTEP de NeXT, entreprise fondée par Steve Jobs à son départ d'Apple, il a été intégré à Mac OS lors de son retour dans la société à la pomme. Bien que principalement utilisé par Apple, ce langage existe également dans le monde linux, où les bibliothèques Cocoa/NeXTSTEP sont par exemple remplacées par GNUstep.

Ce langage est une surcouche très stricte au C. Un compilateur pour objective-C compile correctement tout code C, et les instructions spécifique à l'objective-C sont clairement différenciées lors de la programmation : elles commencent par exemple par une arobase, et les envois de messages aux objets sont explicités entre crochets.

En objective-C, tout est objet. Le typage est faible, ce qui permet notamment de manipuler toute sorte d'objets par le type général *id*. L'introspection est ainsi aisée, et il est possible d'envoyer des messages à des objets qui n'implémentent pas la méthode invoquée au moment de la compilation. En effet, des propriétés telles que le *forwarding* permettent à un objet de récupérer le message qui lui est envoyé, de l'analyser puis de l'envoyer à un autre objet. Ces aspects et beaucoup d'autres sont détaillés dans la documentation Apple

mais également sur la page anglaise de wikipedia correspondante à ce langage.

Voyons quelques aspects basiques de la programmation orientée objet en objective-C.

3.1 Le hello world

Le hello world est quasiment le même qu'en C. Le code est le suivant :

```
1  #import <stdio.h>
2
3  int main( int argc, const char *argv[] ) {
4      printf( "hello world\n" );
5      return 0;
6  }
```

hello.m

On observe qu'*include* est remplacé par *import*, et que l'extension par défaut des fichiers est *.m* .

Pour l'affichage, il est plus courant d'utiliser la fonction *NSLog*, qui prend en argument un objet *NSString*, qui est la chaîne de caractère de l'objective-C. ces chaînes sont notées entre guillemets mais avec une arobase avant le premier guillemet. L'appel est donc `NSLog(@"hello world");`.

3.2 Les classes et l'envoi de messages

Les classes sont découpées en deux fichiers. Le .h contient ce qui est appelé l'interface de la classe, et le .m contient son implémentation. Voici un exemple simple.

```
1  #import <Foundation/Foundation.h>
2
3  @interface Dimensions : NSObject {
4      int largeur, longueur;
5  }
6
7  @property int largeur, longueur;
8
9  - (id) initAvecLargeur: (int) largeurDim etLongueur: (int) longueurDim;
10 @end
```

Dimensions.h

Dans le fichier de déclaration de la classe, on observe plusieurs aspects importants du langage. Tout d'abord, la bibliothèque générale à importer pour utiliser tous les types définis par Cocoa est le fichier *Foundation/Foundation.h*. Celui-ci contient toutes les références des objets *NS** (initiales de NeXTSTEP) tels que *NSObject* ou *NSString* que nous verrons par la suite.

Une classe est définie dans son fichier *.h* par son interface, portant le nom du fichier, qui hérite d'une classe, ici *NSObject*. L'héritage est symbolisé par les deux points (:). Vient ensuite la déclaration des membres de la classe, qui sont dans cet exemple des type simples du C. Les membres sont par défaut privés. Il est possible de les définir explicitement publics, privés ou protégés de la manière suivante :

```
1  #import <Foundation/NSObject.h>
2
3  @interface ExempleAcces: NSObject {
4  @public
```

```
5      int variablePublique;
6  @private
7      int varPrivee;
8  @protected
9      int varProtected;
10 }
11 @end
```

Exemple d'utilisation des niveaux d'accès au membres d'une classe

À la suite de cette déclaration, il est possible de définir des *property*, qui spécifient la manière de générer les accesseurs publics pour les membres de la classe. Il est possible de spécifier des paramètres, tels que *readonly* si on souhaite que la propriété ne possède qu'un *getter* et ne puisse donc pas être modifiée, mais il est également possible de définir son comportement vis-à-vis des accès concurrents : *atomic* et *nonatomic* permettent de spécifier si l'objet doit gérer un accès simultané par plusieurs *threads*.

Les constructeurs par défaut en Objective-C s'appellent *init*. Pour créer un constructeur personnalisé, il convient de l'appeler par un nom commençant par *init*. La syntaxe des méthodes d'une classe est celle du SmallTalk : les arguments apparaissent après des deux points (:) et sont tous nommés. Le nom de notre méthode de la ligne 9 est `initAvecLargeur:etLongueur:`. Elle retourne un *id* qui est, comme défini précédemment, un type générique permettant de manipuler l'intégralité des types objective-C. Enfin, le moins qui débute la ligne permet de dire que cette méthode n'est pas une méthode de classe mais d'instance. Pour les méthodes de classes, définies par exemple par *static* en java, il faut utiliser un +.

Nous avons donc les prototypes des méthodes d'instance définis de la manière suivante :
`- (typeRetour) nomDeLaMethodeAvecArgument1 : (typeArg1) nomArg1 etAutresArguments : (typeArg2) nomArg2 ;`
Pour faire une méthode privée, il suffit de ne pas la faire apparaître dans l'interface.

Il est important de remarquer que la notion d'interface est différente de celle définie dans des langages comme Java. En Java, une interface définit un comportement précis en imposant la présence de certaines méthodes dans les classes l'implémentant. Cette notion en objective-C est présente sous le terme *Protocol*. Par exemple, une classe qui est capable de recevoir et gérer les informations retournées par les accéléromètres de l'appareil doit suivre le *Protocol* **UIAccelerometerDelegate**.

En terme de code, voici la méthode pour utiliser ce protocole au sein d'une classe :

```
1 #import <Foundation/NSObject.h>
2
3 @interface MouvementViewController : UIViewController <UIAccelerometerDelegate> {
4     ...
5 }
6 @end
```

Utilisation du protocole UIAccelerometerDelegate

Contrairement aux interfaces Java, il n'est pas nécessaire d'implémenter toutes les méthodes d'un protocole objective-C lorsqu'une classe le suit.

Une interface en objective-C est donc la déclaration d'une classe au sein d'un fichier *.h*, ce qui se rapproche du langage C puisque ces fichiers sont dédiés à la déclaration de prototypes de fonctions et sont appelés fichiers d'en-tête.

Voyons maintenant l'implémentation de la classe *Dimension* définie précédemment.

```
1  #import "Dimensions.h"
2
3  @implementation Dimensions
4
5  @synthesize longueur, largeur;
6
7  - (id) initAvecLargeur: (int) largeurDim etLongueur: (int) longueurDim {
8      self = [super init];
9
10     if (self) {
11         self.largeur=largeurDim;
12         self.longueur=longueurDim;
13     }
14     return self;
15 }
16
17 @end
```

Dimensions.m

Dans ce fichier, comme en C, il est nécessaire d'importer le fichier *.h* correspondant, qui est ici la déclaration de la classe. Il faut ensuite démarrer l'implémentation. À la ligne 5, nous voyons qu'il est demandé au compilateur de synthétiser les accesseurs pour les *property* longueur et largeur selon les modalités définies dans le *.h*.

Ensuite, les méthodes sont définies. À la 8ème ligne, on observe l'appel au constructeur de la classe mère, placé entre crochets. Les envois de message en Objective-C se font entre crochets. Il ne s'agit pas d'appel de méthodes mais bien d'envoi de message. En effet, comme expliqué précédemment, le destinataire du message peut ne pas avoir déclaré cette méthode, et il est également possible d'envoyer un message à *nil*, qui est justement l'équivalent de *NULL* mais pour les objets objective-C. Nous voyons ensuite des appels aux accesseurs de l'instance, par les commandes *self.largeur* et *self.longueur*. En effet l'appel est implicite dans cette notation, mais si nous avions ajouté le paramètre *readonly* à nos *property* (par exemple en tapant `@property (readonly) int largeur, longueur;` dans l'interface) nous aurions eu l'erreur suivante à la compilation :

```
error: object cannot be set - either readonly property or no setter found
```

4 La gestion des objets et de la mémoire

Pour illustrer la gestion des objets en objective-C, nous allons utiliser le type Dimension défini précédemment. Voici un exemple d'utilisation de ce type dans une classe différente :

```
1  ...
2  - (Dimensions *) obtenirDimensionsIdealesAvec4Joueurs {
3      Dimensions *dim = [[Dimensions alloc] initAvecLargeur:2 etLongueur:2];
4      return dim;
5  }
6  ...
```

Utilisation d'un objet Dimension dans une méthode d'une autre classe

Dans cette méthode dénuée d'intérêt, on constate que l'utilisation d'un objet se fait toujours par son pointeur en objective-C, aussi bien en tant que type de retour que pour son instanciation.

Il existe deux méthodes de gestion de la mémoire : manuelle ou automatique. Dans cet exemple, la mémoire est gérée manuellement : de l'espace mémoire est alloué lors de l'envoi du message *alloc* à la classe

Dimensions, puis le constructeur est appelé sur l'objet dont la mémoire vient d'être alloué. Quand la gestion est manuelle, il est nécessaire d'explicitement envoyer le message *release* à l'objet lorsque celui-ci n'est plus utilisé dans la classe ou fonction en question.

Lorsqu'un objet est créé ou placé dans une variable, un message *retain* lui est envoyé, et il sait alors combien de fois il est encore référencé. En envoyant *release*, ce compteur est décrémenté, et lorsque l'objet atteint 0 référence il est supprimé.

Il est possible d'indiquer que la gestion de la mémoire doit être automatique à l'instancition d'un objet par le biais du message *autorelease* envoyé par exemple à l'objet *dim*.

Il est cependant conseillé de gérer manuellement la mémoire dans un soucis de performance optimale, notamment sur un appareil comme l'iPhone qui n'a pas les capacités techniques d'un ordinateur personnel.

Lors de la conception de classes, il est donc important d'écrire la méthode appelée lors de la suppression d'un objet de cette classe, afin de définir le comportement à tenir vis-à-vis des membres instanciés de cette classe. Voici un exemple de cette méthode :

```
1 - (void) dealloc {
2     [nombreJoueurs release];
3     [ niveauDifficulte  release ];
4 ...
5     [super dealloc ];
6 }
```

Exemple de méthode de désallocation

En général, il suffit de bien envoyer le message *dealloc* aux objets qui ont été instanciés dans le constructeur ou durant le cycle de vie de l'objet, tout en terminant par `[super dealloc]`.

4.1 Autres aspects du langage

Nous n'entrerons pas plus dans le détail ici puisque d'autres aspects du langage seront expliqués lorsque ceux-ci apparaîtrons au fil du document, comme par exemple la gestion des inclusions multiples dans la section 5.2 ou la mise en place du MVC dans la partie 6. Pour approfondir, le site `http://www.otierney.net/objective-c.html` est intéressant car il présente de manière concise et très complète beaucoup d'aspects du langage.

5 Interface Builder

5.1 Description générale et utilisation

Apple Interface Builder est l'outil de création d'interfaces fourni avec XCode. Il est utilisé pour les IHM des applications visant Mac OS X mais également iPhone OS.

Les interfaces d'un projet XCode créées par le biais d'Interface Builder sont des fichiers d'extension *xib* qui sont des fichiers *XML* très complexes. Pour utiliser un de ces fichiers, il faut par exemple créer une instance de *UIViewController* ou d'une classe fille de cette dernière en passant en paramètre d'initialisation le nom de ce fichier *xib*.

L'ordre de la démarche est simple. Dans un premier temps, il faut créer un nouveau fichier depuis l'interface d'XCode, en choisissant la section de templates *User Interface* et sélectionner par exemple *View XIB*.

Cette partie est représenté sur la figure 1.

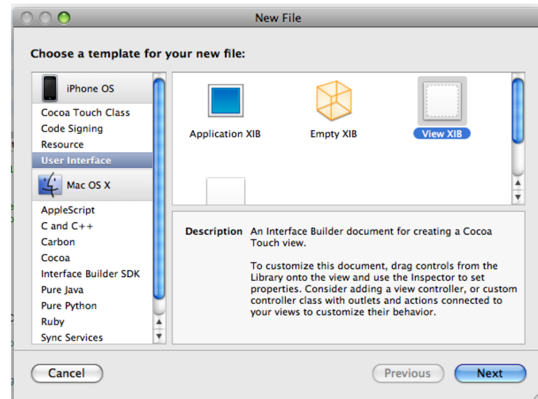

Fig. 1 – Choix du template pour la création d'une nouvelle vue

Lorsque ce fichier est créé, il est intéressant de créer un contrôleur dédié pour cette vue, de lequel vous pourrez mettre des objets tels que des champs de texte ou des labels, mais également des méthodes qui seront appelées par la vue en cas de clic ou autre interaction de l'utilisateur. Pour cela, la démarche est la même que précédemment, mais il faut choisir la section de templates *Cocoa Touch Class* puis le template *UIViewController subclass*. Il est intéressant de noter qu'il est possible de générer avec ce fichier un fichier .xib directement configuré pour y être lié. Cependant, nous allons voir comme faire le lien entre un contrôleur et une vue par le biais d'Interface Builder.

Lorsqu'une classe de contrôleur et une vue sont créés, il faut les lier. Pour ce faire, un double clic sur la vue dans XCode ouvre Interface Builder. Une fois celui-ci chargé, deux configurations sont nécessaires. Dans le fenêtre représentée à gauche sur la figure 2, on retrouve une entité nommée *File Owner*. En cliquant sur cet élément, l'inspecteur affiche les informations le concernant. Dans notre cas, la vue que nous avons créé doit appartenir à notre contrôleur : il faut donc sélectionner la classe correspondante dans la partie *Class Identity* dans la partie *Identity* de l'inspecteur.

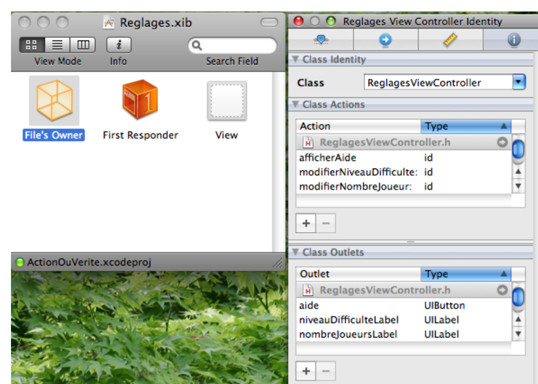

Fig. 2 – Fenêtres permettant le lien entre une vue et son contrôleur dans Interface Builder

5.2 Création d'un contrôleur et problème d'inclusion multiple en objective-C

Maintenant que le contrôleur est assigné à la vue, il faut que le contrôleur sache lui aussi qui est sa vue. Pour cela, un clic droit sur ce dernier permet d'afficher les éléments qu'il possède pouvant être liés à une partie de la vue et les méthodes pouvant être appelées en cas d'événement survenu sur la vue. Il suffit alors de lui indiquer que sa vue est celle que nous éditons, c'est à dire celle placée dans la fenêtre de gauche représentée par la figure 2. La figure 3 présente un exemple de File Owner ayant des attributs liés à la vue et des méthodes liées à des événements.

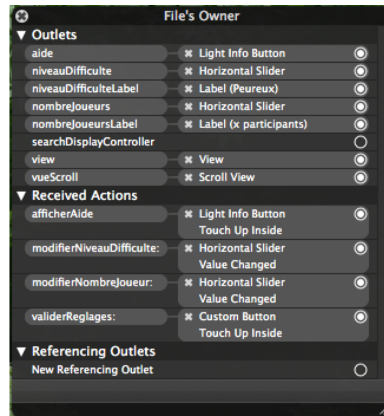

FIG. 3 – Exemple de menu contextuel pour un *File Owner*

Voici l'interface correspondante à ce contrôleur :

```
1  #import <UIKit/UIKit.h>
2
3  @class ControleurDeVue;
4  @class Enonces;
5
6  @interface ReglagesViewController : UIViewController <UITextFieldDelegate> {
7      NSMutableArray *nomsJoueurs, *sexesJoueurs;
8      UISlider *nombreJoueurs, *niveauDifficulte;
9      UITextField *textfieldActif;
10     UILabel *niveauDifficulteLabel, *nombreJoueursLabel;
11     BOOL clavierVisible;
12     UIScrollView *vueScroll;
13     ControleurDeVue *controleurPrincipal;
14     UIButton *aide;
15 }
16
17 @property (nonatomic, retain) IBOutlet UISlider *nombreJoueurs, *niveauDifficulte;
18 @property (nonatomic, retain) IBOutlet UILabel *nombreJoueursLabel, *niveauDifficulteLabel;
19 @property (nonatomic, retain) IBOutlet UIScrollView *vueScroll;
20 @property (nonatomic, retain) ControleurDeVue *controleurPrincipal;
21 @property (nonatomic, copy) NSMutableArray *nomsJoueurs, *sexesJoueurs;
22 @property (nonatomic, retain) UITextField *textfieldActif;
23 @property (nonatomic, retain) IBOutlet UIButton *aide;
24 @property BOOL clavierVisible;
25
26 - (void)  sEnregistrerEnTantQuObserver;
27 - (IBAction)modifierNiveauDifficulte:(id)sender;
28 - (IBAction)modifierNombreJoueur:(id)sender;
29 - (void) mettreAJourNombreJoueurs: (NSNotification *) notif;
30 - (void) mettreAJourNiveauDifficulte: (NSNotification *) notif;
31 - (IBAction)validerReglages:(id)sender;
```

```
32 - (IBAction) afficherAide;
33
34 @end
```

ReglagesViewController.h

Dans ce fichier, nous avons plusieurs aspects intéressants. Tout d'abord nous constatons l'utilisation d'un protocole nommé *UITextFieldDelegate*, qui définit les méthodes permettant à la classe de recevoir les informations relatives à l'utilisation d'un champ de texte.

Ensuite, les lignes 3 et 4 présentent une nouvelle instruction : la commande @class. Cette instruction permet de préciser que l'on utilise une classe qui sera par exemple définie ultérieurement. Ainsi, il n'est pas nécessaire d'importer la classe *ControleurDeVue* pour définir un membre de ce type. Il suffit alors d'importer cette classe au moment de l'implantation de notre interface, dans le fichier *.m*.

Cette technique permet de contourner le problème des inclusions multiples du langage C qui nous contraint normalement à utiliser des constantes pour définir si un fichier d'en-têtes a déjà été chargé ou non et ainsi ne pas l'appeler de nouveau. Ici, les fichiers d'interfaces ne contiennent pas l'appel aux autres fichiers d'en-têtes définissant des classes. L'inclusion multiple n'a donc jamais lieu.

Enfin, certaines lignes de définition de *property* contiennent un mot-clef *IBOutlet*. Ce dernier permet à Interface Builder de savoir à quels attributs il est possible de lier les éléments que nous plaçons sur la vue. De même, *IBAction* est un type de retour identique à *void* mais qui indique à Interface Builder que cette méthode peut être appelée depuis la vue. Si on compare cette définition de classe à la figure 3, on retrouve effectivement les éléments voulu dans l'outil d'édition de vue et il est alors possible de les lier à des éléments tels que des boutons, des sliders et autres.

Aux lignes 29 et 30, on remarque un type appelé *NSNotification*. Celui-ci est utilisé lors de la mise en place du *Design Pattern Observer* et nous allons en expliquer le fonctionnement dans la section suivante.

Pour conclure sur l'outil Inteface Builder, celui-ci est un outil de création d'interface extrêmement puissant et pratique. Il est très bien intégré à XCode, mais il faut garder en tête que des interfaces définies par le biais de tels outils sont statiques, et que pour définir une vue qui évolue par exemple en affichant des objets initialement absents, il est nécessaire de savoir développer des interfaces dans le code, ce que nous présenterons lors de notre exemple plus loin dans ce document.

6 Le MVC sur iPhone

6.1 L'organisation d'un projet en MVC

Dans un projet Xcode, l'organisation des classes est libre. Il est donc possible de créer des dossiers dans lesquels placer le modèle, les contrôleurs et les vues séparément. Le framework Cocoa Touch est très orienté MVC. En effet, comme expliqué précédemment, les classes héritant de *UIViewController* sont les contrôleurs, et les fichiers *.xib* sont les vues. Pour le modèle, il suffit donc de créer soit même des classes héritant de la classe *NSObject*. *Core Data* est un Framework permettant la gestion du modèle, par exemple en aidant à l'utilisation de commandes telles que *undo* ou en permettant la persistance des données du modèle. Cependant, nous ne détaillerons pas son utilisation dans ce document.

Dans le Design Pattern MVC, la vue se met à jour lorsque le modèle est modifié. Le Design Pattern Observer a pour but de permettre au modèle, qui est dit observable, d'informer directement la vue, qui est dite *observer*. En Cocoa, ce Design Pattern est implanté par le biais des notification *NSNotification*.

Le principe est simple : des objets peuvent envoyer des notifications à un centre de notification, en leur donnant un nom spécifique, par exemple *NombreJoueurModifieNotification*. La vue qui souhaite recevoir cette notification s'enregistre auprès du centre de notification en précisant quelle méthode doit être appelée pour cette notification. Il est possible de joindre à une notification un objet de type *NSDictionary* contenant des objets liés à cette notification. Voici un exemple d'utilisation pour simplifier la mise à jour de la vue :

```
1  - (void) modifierNombreJoueursActif:(int) nouveauNombre {
2      if (nouveauNombre <= [lesJoueurs.joueurs count]) {
3          lesJoueurs.nombreActifs = nouveauNombre;
4          [[NSNotificationCenter defaultCenter] postNotificationName:@"NombreJoueursModifieNotification" object:self
       userInfo: [NSDictionary dictionaryWithObjectsAndKeys: [NSNumber numberWithInt:lesJoueurs.nombreActifs], @"new"
       , nil]];
5      }
6  }
```

Modèle : envoi d'une notification

Dans cet exemple d'envoi de notification, on observe que dans un premier temps le centre de notification par défaut est récupéré, et qu'il est envoyé à celui ci le message *postNotificationName :object :userInfo :*. On peut ainsi préciser qui est l'expéditeur de la notification, et on voit que *userInfo* correspond à l'information que l'on souhaite associer, sous la forme d'un dictionnaire. Ici, le dictionnaire contient une seule entrée, ayant pour clé la chaîne *new* et on y associe le nombre d'actifs. Il est important de noter que le nombre doit être un objet, et que nous ne pouvons donc pas joindre directement un *int*. On observe ainsi, comme pour la création du dictionnaire, un envoi de message à une classe, ici *NSNumber* pour créer directement un objet à partir d'un *int*. L'intérêt de générer l'objet de cette manière est qu'il n'y a pas à se soucier de la gestion de la mémoire évoqué précédemment dans la section 4.

En utilisant des *UIViewController* et les vue *.xib*, c'est au sein du contrôleur que la notification est récupérée. Cependant, il est possible de créer des classes héritant de la classe *UIView* et récupérer alors la notification directement dans la vue. Mais l'utilisation de *UIViewController* et de *xib* réduit la limite entre les vues et les contrôleurs et la gestion dynamique de la vue se fait ainsi dans le contrôleur.

Au niveau du contrôleur (ou de la vue), il faut dans un premier temps s'enregistrer auprès du centre de notification, puis définir la méthode associée à la notification désirée.

```
1   ...
2   [[NSNotificationCenter defaultCenter] addObserver:self
3                                            selector:@selector(mettreAJourNombreJoueurs:)
4                                                name:@"NombreJoueursModifieNotification" object:nil];
5   ...
6   - (void) mettreAJourNombreJoueurs: (NSNotification *) notif {
7       NSDictionary* info = [notif userInfo];
8       // modification du nombre dans le label
9       nombreJoueursLabel.text = [NSString stringWithFormat:@"%d participants", [[info objectForKey:@"new"] intValue]];
10  ...
11  }
```

Contrôleur ou Vue : réception de la notification

Nous voyons ici le message à envoyer au centre de notification. Il est explicitement dit que l'on s'enregistre en temps qu'*Observer*, on associe une méthode puis on précise le nom de la notification en question. Le paramètre *object* permet de préciser si l'on souhaite recevoir cette notification uniquement d'un objet précis, dans le cas ou plusieurs objets peuvent envoyer des notifications. Pour définir la méthode appelée, on constate qu'il est possible de donner un objet correspondant à une méthode à partir de son nom avec l'instruction *@selector()*.

La méthode correspondante doit accepter comme argument un objet de type *NSNotification* qui contiendra l'éventuel dictionnaire envoyé avec la notification. Les lignes suivantes précisent la démarche à adopter pour récupérer une entrée de ce dictionnaire.

Enfin, il existe des centre de notifications adaptés à l'utilisation de plusieurs processus dans une application, et il également possible de différer l'envoi de la notification.

7 Application : générer une liste dynamique de champs de texte

7.1 Notions et aspects présents dans l'exemple

Dans cet exemple, le MVC sera mis en place, avec en plus le Design Pattern Observer. Il sera également montré comment utiliser des *UIScrollPane*, notamment dans le cas ou un clavier doit apparaître pour permettre de rentrer du texte.

Nous allons donc créer une vue, dans laquelle nous mettrons un *UIScrollPane* pour pouvoir y déposer des *UITextField* en nombre variable (nombre changé par le biais d'un *UISlider*).

7.2 Mise en place des fichiers nécessaires

Pour mettre en place cela, il nous faut deux fichiers de classe et une vue *xib*.

Il est possible de créer 3 dossiers pour y classer séparément nos modèle, contrôleur et vue, par le bais du panneau *Groups & Files* de XCode. Créons un fichier appelé *ReglagesViewController*, contenant une classe héritant de *UIViewController*, puis une classe appelée Joueurs, définie comme suit :

```
1  #import <Foundation/Foundation.h>
2
3  @interface Joueurs : NSObject {
4      NSMutableArray *joueurs;
5      int nombreActifs;
6  }
7
8  @property (nonatomic, retain) NSMutableArray *joueurs;
9  @property int nombreActifs;
10
11 - (id) initAvecNombreMaxJoueurs:(int)nbMaxJoueurs;
12 - (void) definirNomJoueurAvecIndex:(int) indexJoueur etNom: (NSString *) nouveauNom;
13 - (void) modifierNombreActifsAvecNombre: (int) nbActifs;
14 @end
```

Joueurs.h

Cette classe est minimaliste, elle représente une classe du modèle d'un jeu quelconque. L'intérêt ici est de lier cette classe à la vue pour permettre à cette dernière d'être à jour. Le tableau de joueurs est rempli de chaînes de caractères correspondants aux noms des joueurs, avec une capacité totale de *nbMaxJoueurs*, mais seuls *nombreActifs* joueurs actifs et donc affichés dans la vue.

Voici le contenu du fichier d'extension *.m* :

```
1  #import "Joueurs.h"
2
3  @implementation Joueurs
4
```

```
5     @synthesize joueurs, nombreActifs;
6
7     -(id) initAvecNombreMaxJoueurs:(int)nbMaxJoueurs {
8         self = [super init];
9
10        if (self) {
11            String *joueur;
12            NSMutableArray *t = [[NSMutableArray alloc] initWithCapacity:nbMaxJoueurs];
13            for (int i = 0; i < nbMaxJoueurs; i++) {
14                joueur = [NSString stringWithFormat:@"Joueur %d", i+1];
15                [t addObject:joueur];
16            }
17            self.joueurs = t;
18            [t release];
19        }
20        return self;
21    }
22
23    // methode permettant de mettre a jour le modele et d'en informer la vue
24    - (void) definirNomJoueurAvecIndex:(int) indexJoueur etNom: (NSString *) nouveauNom {
25        if (indexJoueur < [joueurs count]) {
26            if ([nouveauNom compare:@""] == NSOrderedSame)
27                [[joueurs setValue: [NSString stringWithFormat:@"Joueur %d", indexJoueur+1] forKey: indexJoueur];
28            else
29                [[joueurs setValue: nouveauNom forKey: indexJoueur];
30            // On informe la vue en envoyant une notification que recevront les observeurs
31            [[NSNotificationCenter defaultCenter] postNotificationName:@"NomJoueurModifieNotification" object:self
      userInfo:
32             [NSDictionary dictionaryWithObjectsAndKeys: [NSNumber numberWithInt:indexJoueur], @"index", [[joueurs
      objectAtIndex:indexJoueur], @"new", nil]];
33        }
34    }
35
36    -(void) modifierNombreActifsAvecNombre: (int) nbActifs {
37        if (nbActifs <= [joueurs count]) {
38            self.nombreActifs = nbActifs;
39            [[NSNotificationCenter defaultCenter] postNotificationName:@"NombreJoueursModifieNotification" object:self
      userInfo:
40             [NSDictionary dictionaryWithObjectsAndKeys: [NSNumber numberWithInt:nombreActifs], @"new", nil]];
41        }
42    }
43
44    -(void) dealloc {
45        [joueurs release];
46        [super dealloc];
47    }
48
49    @end
```

Joueurs.m

Dans ce fichier, il est intéressant de regarder la gestion de la mémoire. Au sein du constructeur nous initialisons un *NSMutableArray* (tableau dont les éléments peuvent être modifiés, par opposition à *NSArray* sa classe mère) que nous plaçons par la suite dans la variable *joueurs* de notre classe. En appelant `self.joueurs`, nous faisons appel au *setter* défini dans le fichier d'en-têtes avec les propriétés *nonatomic* et *retain*. *Retain* correspond à la méthode utilisée pour stocker la variable lors de l'appel au *setter* : cet attribut peut être *retain*, *copy* ou *assign*. Dans notre cas, *retain* informe que lors de la réception de l'argument dans le *setter* on souhaite lui envoyer un message *retain*. Le compteur est ainsi incrémenté pour cette instance, et ceci explique pourquoi nous pouvons appeler `[t release];` sur notre tableau : il lui reste encore une référence.

Par défaut, nous avons remplis les noms des utilisateurs à "Joueur x", x étant le numéro de celui-ci. La

variable *nombreActifs* indique ceux qui sont vraiment utilisés. Cette technique permet de conserver les noms définis pour des joueurs même lorsque l'utilisateur choisit de réduire le nombre d'actifs puis de l'augmenter à nouveau.

En ce qui concerne les méthodes envoyant les notifications, il est important de remarquer également que la seconde est une sorte de surcharge du *setter* et devrait être faite proprement en surchargeant correctement le setter, mais nous simplifions ici le code pour qu'il soit plus simplement appréhendable. La documentation officielle Apple précise comment surcharger les *getter* et *setter*.

Voyons ensuite le contrôleur qui est associé à notre vue.

```
1  #import <UIKit/UIKit.h>
2
3  @class Joueurs;
4
5  @interface ReglagesViewController : UIViewController <UITextFieldDelegate> {
6      NSMutableArray *nomsJoueurs;
7      UISlider *nombreJoueurs;
8      UITextField *textfieldActif;
9      UILabel *nombreJoueursLabel;
10     BOOL clavierVisible;
11     UIScrollView *vueScroll;
12     Joueurs *joueurs;
13 }
14
15 @property (nonatomic, retain) IBOutlet UISlider *nombreJoueurs;
16 @property (nonatomic, retain) IBOutlet UILabel *nombreJoueursLabel;
17 @property (nonatomic, retain) IBOutlet UIScrollView *vueScroll;
18 @property (nonatomic, retain) NSMutableArray *nomsJoueurs;
19 @property (nonatomic, retain) UITextField *textfieldActif;
20 @property (nonatomic, retain) Joueurs *joueurs;
21 @property BOOL clavierVisible;
22
23 - (IBAction)modifierNombreJoueur:(id)sender;
24 - (IBAction)validerReglages:(id)sender;
25
26 @end
```

ReglagesViewController.h

Tous les éléments de la vue sont déclarés ici. En réalité, si certains labels ou autres éléments de la vue ne sont pas utilisés dynamiquement, c'est à dire modifiés ou supprimés, il n'est pas nécessaire de les faire apparaître dans le contrôleur. Comme énoncé précédemment, il est important de savoir qu'il n'est pas obligatoire de gérer le contenu de la vue depuis le contrôleur mais qu'il est possible de faire des classes de vue. Dans notre exemple, puisque nous utilisons Interface Builder, il est plus simple de tout contrôler depuis le contrôleur.

Nous avons ici un nouvel exemple d'utilisation de l'instruction *@class*, qui nous permet d'avoir un attribut de type Joueurs sans avoir à importer le fichier *Joueurs.h*.

Voyons maintenant l'implémentation de cette classe.

```
1      #import "ReglagesViewController.h"
2      #import "Joueurs.h"
3
4      @implementation ReglagesViewController
5
```

```
6     @synthesize nombreJoueurs, nombreJoueursLabel, nomsJoueurs, vueScroll, textfieldActif, clavierVisible;
7
8     // reagit au slider pour mettre a jour le label correspondant et ajouter les champs de texte
9     // necessaires pour rentrer les noms des joueurs
10    - (IBAction)modifierNombreJoueur:(id)sender {
11        // Selon la position du slider, on interprete differement le choix de l'utilisateur
12        if (nombreJoueurs.value < [controleurPrincipal.laPartie obtenirNombreJoueursActifs] - 0.5 ||
13            nombreJoueurs.value > [controleurPrincipal.laPartie obtenirNombreJoueursActifs] + 0.5) {
14                // mise a jour du modele en fonction cette modification de la vue
15                [joueurs modifierNombreActifsAvecNombre: floor(nombreJoueurs.value + 0.5)];
16            }
17    }
18
19    - (void) mettreAJourNombreJoueurs: (NSNotification *) notif {
20        // La notification nous permet de recuperer la nouvelle valeur du nombre de joueurs dans le modele
21        NSDictionary* info = [notif userInfo];
22        int nbJoueursRecu = [[info objectForKey:@"new"] intValue];
23        // modification du nombre affiche
24        nombreJoueursLabel.text = [NSString stringWithFormat:@"%d participants", nbJoueursRecu];
25
26        // modification du nombre de textfields affiches
27        UITextField *tf;
28
29        // ajout des textfields manquants
30        for (int i=0; i< nbJoueursRecu; i++) {
31            if (![ tf = [self.nomsJoueurs objectAtIndex:i] superview]) {
32                // si l'objet a l'index i n'est pas dans la vue principale, i.e. s'il n'a pas de superview,
33                // on l'ajoute a la vue
34                [self.vueScroll addSubview: tf];
35            }
36        }
37
38        // retrait des textfields et switchs en trop
39        bool termine = FALSE;
40        int i = nbJoueursRecu;
41        while (!termine && i < [joueurs.joueurs count]) {
42            if ([tf = [self.nomsJoueurs objectAtIndex:i] superview]) {
43                [tf removeFromSuperview];
44            } else {
45                termine = TRUE;
46            }
47            i++;
48        }
49
50        // agrandissement de la fenetre scroll
51        [self.vueScroll setContentSize:CGSizeMake(320, 300 + nbJoueursRecu*40)];
52    }
53
54    - (IBAction)validerReglages:(id)sender {
55        // Passage a la vue suivante, par exemple en appelant une methode sur un controleur principal
56        // possedant ce controleur et dont on aurait ici une reference
57    }
58
59    // methode appelee grace au protocol UITextFieldDelegate
60    - (void)textFieldDidBeginEditing:(UITextField *)theTextField {
61        textfieldActif = theTextField;
62    }
63
64    // methode appelee grace au protocol UITextFieldDelegate
65    - (void)textFieldDidEndEditing:(UITextField *)theTextField {
66        int i = 0;
67        bool trouve = FALSE;
68        while (i< joueurs.nombreActifs && !trouve) {
69            if ([nomsJoueurs objectAtIndex:i] == theTextField)
70                trouve = true;
```

```
71            i++;
72        }
73        if (trouve) {
74            [ self definirNomJoueurAvecIndex:i−1 etNom:theTextField.text];
75        }
76
77    }
78
79    − (void) mettreAJourNomJoueur: (NSNotification ∗) notif {
80        NSDictionary∗ info = [notif userInfo ];
81        int index = [[info objectForKey:@"index"] intValue];
82        UITextField ∗t = [nomsJoueurs objectAtIndex: index];
83        NSString ∗nouveauNom = [info objectForKey:@"new"];
84        if ([nouveauNom compare:[NSString stringWithFormat:@"Joueur %d", index+1]] != NSOrderedSame)
85            t.text = nouveauNom;
86    }
87
88    − (void) sEnregistrerEnTantQuObserver {
89        // notification utile pour gerer l'arrivee du clavier sur la vue
90        [[ NSNotificationCenter defaultCenter] addObserver:self
91                                                selector :@selector(keyboardWasShown:)
92                                                    name:UIKeyboardDidShowNotification object:nil];
93
94        // notification utile pour gerer l'arrivee du clavier sur la vue
95        [[ NSNotificationCenter defaultCenter] addObserver:self
96                                                selector :@selector(keyboardWasHidden:)
97                                                    name:UIKeyboardDidHideNotification object:nil];
98
99        [[ NSNotificationCenter defaultCenter] addObserver:self
100                                               selector :@selector(mettreAJourNombreJoueurs:)
101                                                   name:@"NombreJoueursModifieNotification" object:nil];
102
103       [[ NSNotificationCenter defaultCenter] addObserver:self
104                                               selector :@selector(mettreAJourNomJoueur:)
105                                                   name:@"NomJoueurModifieNotification" object:nil];
106   }
107
108   − (void) keyboardWasShown:(NSNotification ∗) notif {
109       if (! self . clavierVisible ) {
110           NSDictionary∗ info = [notif userInfo ];
111           // Recuperons la taille du clavier
112           NSValue∗ aValue = [info objectForKey:UIKeyboardBoundsUserInfoKey];
113           CGSize keyboardSize = [aValue CGRectValue].size;
114
115           // Redimensionnement de la vue scroll
116           CGRect viewFrame = [vueScroll frame];
117           viewFrame.size.height −= keyboardSize.height;
118           vueScroll.frame = viewFrame;
119       }
120
121       // On remonte le textfield actif dans une partie visible de l'ecran.
122       CGRect textFieldRect = [textfieldActif frame];
123       textFieldRect.origin.y += 40;
124       [vueScroll scrollRectToVisible:textFieldRect animated:YES];
125
126       self . clavierVisible = YES;
127   }
128
129   − (void)keyboardWasHidden:(NSNotification ∗) notif {
130       NSDictionary∗ info = [notif userInfo ];
131
132       // On prend la taille du clavier
133       NSValue∗ aValue = [info objectForKey:UIKeyboardBoundsUserInfoKey];
134       CGSize keyboardSize = [aValue CGRectValue].size;
135
```

```
136         // on remet la taille de la vue scroll comme il faut
137         CGRect viewFrame = [vueScroll frame];
138         viewFrame.size.height += keyboardSize.height;
139
140         // on met une petite animation
141         [UIView beginAnimations:nil context:NULL];
142         [UIView setAnimationBeginsFromCurrentState:YES];
143         [UIView setAnimationDuration:0.3];
144
145         vueScroll.frame = viewFrame;
146
147         [UIView commitAnimations];
148         self . clavierVisible = NO;
149     }
150
151     // The designated initializer . Override to perform setup that is required before the view is loaded.
152     − (id)initWithNibName:(NSString ∗)nibNameOrNil bundle:(NSBundle ∗)nibBundleOrNil {
153         if ( self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
154             // Custom initialization
155             [ self sEnregistrerEnTantQuObserver];
156             self . clavierVisible = FALSE;
157         }
158         return self ;
159     }
160
161     // Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
162     − (void)viewDidLoad {
163         [super viewDidLoad];
164
165         // configuration du scroll de la vue, cf . documentation pour details des differentes valeurs
166         [ self . vueScroll setCanCancelContentTouches:NO];
167         self . vueScroll . clipsToBounds = YES;
168         self . vueScroll . indicatorStyle = UIScrollViewIndicatorStyleWhite;
169         // on prend une taille adequate pour afficher le contenu qui est dependant du nombre de joueurs actifs
170         [ self . vueScroll setContentSize:CGSizeMake(320, 300 + joueurs.nombreActifs∗40)];
171         [ self . vueScroll setScrollEnabled:YES];
172
173         nombreJoueurs.minimumValue = 2;
174         // reglage du slider en fonction de la constante du nombre max de joueur dans le jeu
175         // on considere que la variable joueurs de notre controleur a ete remplie depuis la classe qui l'a
176         // instancie . Il faudrait creer un constructeur pour ce controleur qui prend en argument cet objet joueurs.
177         if (joueurs != nil)
178             nombreJoueurs.maximumValue = [joueurs.joueurs count];
179         else
180             [NSException raise:@"Variable non definie" format:@"la variable joueurs n'est pas definie" ];
181
182         // affichage de 4 textfields , et ecriture de "4 participants" dans le label adequat.
183         nombreJoueursLabel.text = [NSString stringWithFormat:@"%d participants", 4];
184         nombreJoueurs.value = [controleurPrincipal.laPartie obtenirNombreJoueursActifs];
185
186         // mise en place de l' affichage dynamique des champs texte
187         NSMutableArray ∗tableauTextfields = [[NSMutableArray alloc] init];
188         UITextField ∗t;
189         NSString ∗chaine;
190
191         for (int i=0; i < [joueurs.joueurs count]; i++) {
192             t = [[UITextField alloc] initWithFrame: CGRectMake(40.0, 280.0 + i∗40, 150.0, 31.0)];
193             chaine = [[NSString alloc] initWithFormat:@"Joueur %d", i+1];
194
195             t.textAlignment = UITextAlignmentCenter;
196             t.borderStyle = UITextBorderStyleRoundedRect;
197             t.autocapitalizationType = UITextAutocapitalizationTypeWords;
198             t.keyboardType = UIKeyboardTypeDefault;
199             t.autocorrectionType = UITextAutocorrectionTypeNo;
200             t.returnKeyType = UIReturnKeyDone;
```

```
201            t.delegate = self;
202            // mise du texte gris par defaut dans le fond du champ texte
203            [t setPlaceholder:chaine];
204
205            [tableauTextfields addObject:t];
206            [chaine release];
207            [t release];
208        }
209
210        self.nomsJoueurs = tableauTextfields;
211
212        [tableauTextfields release];
213
214        // affichage des textfields du depart
215        for (int i=0; i< 4; i++) {
216            [self.vueScroll addSubview:[self.nomsJoueurs objectAtIndex:i]];
217        }
218
219    }
220
221    // Par defaut, le clavier ne se retire pas quand on valide par le bouton en bas a droite
222    // il faut donc lui dire que faire.
223    - (BOOL)textFieldShouldReturn:(UITextField *)theTextField {
224        [theTextField resignFirstResponder];
225        return YES;
226    }
227
228    - (void)didReceiveMemoryWarning {
229        [super didReceiveMemoryWarning]; // Releases the view if it doesn't have a superview
230        // Release anything that's not essential, such as cached data
231    }
232
233    - (void)dealloc {
234        [nombreJoueurs release];
235        [nombreJoueursLabel release];
236        [nomsJoueurs release];
237        [vueScroll release];
238        [textfieldActif release];
239        [super dealloc];
240    }
241    @end
```

ReglagesViewController.m

Cette classe est assez conséquente, c'est pourquoi j'ai choisi de commenter le code directement pour divers aspects de la programmation.

Pour résumer les enjeux, il faut comprendre deux choses. Dans un premier temps, nous avons défini une vue avec Interface Builder dans laquelle nous avons placé une *UIScrollView* que nous avons lié à sa variable correspondante dans le *File Owner* qui est de classe *ReglagesViewController*. Les labels sont eux aussi à placer dans la vue, manuellement. De plus, il faut comprendre que tous les éléments d'une vue sont eux-même des vues : la vue scroll est une vue au même titre que celle qui la contient, et les champs de textes en sont également. Tous ces objets se manipulent donc tous de la même manière, et il est ainsi possible de savoir si un champ texte est affiché en lui demandant sa *superview*.

Ensuite, les difficultés liées à la mise en place d'une telle vue sont multiples :

Gérer l'affichage dynamique des champs de texte La mise en place de ceux-ci est déclenchée par la réception d'une notification. En effet, dans le constructeur de la classe nous nous sommes enregistrés en tant qu'observer de deux notifications, qui concernent les modifications du modèle qui nous intéresse. Lorsque l'on reçoit la notification qui indique un changement du nombre de joueurs actifs, on sait que

l'on doit modifier l'affichage pour adapter celui-ci au modèle. Pour ce faire, on les positionne dans la vue scroll en leur fixant une taille et un emplacement spécifiques en pixels. Ainsi, dans une boucle, il est possible d'en générer un nombre variable. Parallèlement à ça, il faut ajuster la taille définie pour le contenu dans la vue scroll, et ceci de la même manière que pour les *textfield*,

Gérer l'affichage du clavier et le scroll jusqu'au champ choisi Lorsque le clavier s'affiche, par défaut il se place au dessus de la vue scroll, et ce sans rien faire d'autre. Or, le problème est dans un premier temps que le champ sélectionné est probablement caché par le clavier, mais également que le bas du contenu de la vue scroll ne sera jamais accessible puisque caché. La solution est de redimensionner la vue scroll pour n'occuper que la partie non cachée de l'écran quand le clavier est visible, et de scroller automatiquement dans cette vue pour que le champ de texte désiré se retrouve au dessus du clavier. Ces deux techniques sont mises en place dans la méthode `keyboardWasShown:`, et le retour à la normal est géré par `keyboardWasHidden:`. Ces deux méthodes se trouvent respectivement aux lignes 108 et 129.

Pour que ceci se déroule bien, il faut évidemment identifier le champ qui est actuellement édité. En écrivant la méthode `textFieldDidBeginEditing:`, à la ligne 60, puisqu'on suit le protocole *UITextFieldDelegate*, on peut savoir quand un champ texte est sélectionner et ainsi le stocker dans une variable pour pouvoir l'identifier lors de l'apparition du clavier. Lorsque l'édition du champ texte est terminée, on met à jour le modèle avec le nouveau nom tapé.

Cette méthode n'est probablement pas parfaite, mais après de longue recherches, elle s'avère être une des plus efficaces. Je l'ai mise au point à partir de diverses techniques proposées par d'autres développeurs notamment sur les forum d'Apple.

Après un travail de présentation, et en élaborant un peu plus le contenu (possibilité de choisir un niveau de difficulté, de définir les sexes des joueurs), la vue devient comme présentée sur la figure 4.

Fig. 4 – Exemple de rendu de la méthode proposée, avant et après le choix d'un champ texte

Enfin, il reste quelques aspects intéressants de la programmation en objective-C pour iPhone. la fonction `validerReglages:` de la ligne 54 est appelée depuis la vue. Son type de retour est *IBAction*, ce qui permet comme expliqué précédemment à Interface Builder de l'identifier comme disponible pour la vue. L'argument reçu est de type *id*. Il s'agit de l'émetteur de l'action. Ainsi, si cette méthode peut être appelée depuis deux boutons dans la vue, il est possible de savoir duquel des deux il s'agit. Ensuite, en terme de gestion de la mémoire, il est également possible de parametrer un comportement spécifique en cas de problème. La

méthode *didReceiveMemoryWarning* permet de définir une conduite à suivre pour alléger la charge et éviter une fermeture forcée de l'application.

8 Conclusion

Pour un développeur connaissant le C et la programmation orientée objet, la transition à ce langage est relativement aisée. La plus grosse différence vient plutôt de la plateforme elle-même. En effet, les contraintes de performance et de sauvegarde de la mémoire permettent au développeur de gagner en concision. C'est également un avant goût intéressant de ce que peut être la programmation pour systèmes mobiles et embarqués, et c'est à mon avis une branche intéressante de l'informatique.

Bien que ce document n'ait pas la prétention de remplacer les documentations officielles fournies par Apple, je pense qu'il récapitule une grande partie des notions primordiales au développement pour iPhone et Mac OS X. J'espère enfin que ce document pourra aider certains à débuter, comprendre un peu mieux certaines notions du langage ou encore résoudre certains problèmes de gestion du clavier de l'iPhone.