Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Introduction à l'Objective C

ainsi qu'à son environement de dévelopement

Projet de semestre

Filière informatique

v1.0

Superviseur : Jacques Bapst
Auteur : Xavier Butty

Printemps 2011

Table des matières

Chapitre 1

Introduction à l'Objective C

1.1 But du document

Ce document a été réalisé dans le cadre de mon projet de semestre au printemps 2011. Il a pour but de faire une brève introduction de l'objective C pour des développeurs habitué à travailler avec Java ou encore C. Seules les concepts de base de ce langage sont présentés au travers d'exemples et de comparaisons avec d'autres langages.

1.2 Objective C, iOs et Cocoa Touch

Cette première partie d'analyse concerne les technologies utilisées pour le développement sur la tablette tactile d'*Apple* tels que l'Objective C et l'API Cocoa.

1.2.1 Introduction

Le début des années septante a vu naître un langage de programmation qui devint rapidement populaire : le C. Ce langage, faisant toujours partie des plus utilisés, servit à d'autres innovations informatiques majeures de cette décennie. Le système UNIX écrit entièrement en C en est un exemple.

Cependant, l'arrivée de la programmation orientée objet révéla une des principales limites de C. En effet, ce langage procédural se prêtait parfaitement à la programmation structurée majoritairement utilisé dans les années 70-80. Mais dès la fin des années septante, pour palier à cette limitation, des extensions de C offrant la programmation orientée objet furent développés. Bjarne Stroustrup développa par exemple le C++ en 1979. C'est quelques années plus tard, au début des années 80, que Brad Cox et Tom Love créent l'Objective C. Les deux développeurs de la société Stepstone se basèrent sur Smalltalk-80, un langage orientée objet du début des années septante, pour ajouter leur propre couche orientée objet au C.

Comme pour Java et surtout Smalltalk-80 dont il s'inspire, le code compilé d'Objective C s'exécute au sein d'un runtime léger lui-même écrit en C. Le langage de Cox et Love est donc bien une couche du C. Ce runtime a un rôle important ; En effet, comme pour java, ce moteur d'exécution se charge de la création des classes et objets, de l'évaluation des types de variables, etc. Elle distribue aussi les différents messages entres les objets. Ce dernier point est expliqué plus en détails ci-dessous, dans le sous-chapitre du même nom.

En 1988, NeXT, jeune société fondée par Steve Jobs, achète la licence d'Objective C à StepStone et crée son propre compilateur et bibliothèque Objective C. Grâce à cela, NeXT développa son propre système d'exploitation : NeXTSTEP. Même si ce système ne fut pas un succès, il disposait de

nombreux avantages comme son API performante et orienté objet (OpenStep, l'ancêtre de Cocoa), son environnement de développement et sa base UNIX. D'ailleurs de nombreux projets célèbres ont été développés sur ce système : Les jeux Doom et Wolfenstein ou plus sérieusement le Web par Tim Berners-Lee au CERN. En rachetant NeXT en 1996, Apple acquère NeXTSTEP qui va leur servir pour le développement de leur prochain système : Mac OS X. En effet, l'environnement de développement fut repris et deviendra plus tard Xcode alors que Cocoa se basa sur l'API de NeXT. Les différentes particularités d'Objective C ainsi que son organisation de base sont présentées dans

FIGURE 1.1 – Logo de Next

la suite de cette section. A noter que de bonnes bases de C sont très utiles pour la compréhension de ce langage ; Que ce soit au niveau des types ou encore la notion de pointeur.

La version d'Objective C utilisée sur iOS 4 est l'Objective C 2.0 sorti en 2006. Cependant, il ne dispose pas sur ce système de toutes les améliorations apportées. Par exemple, le *garbage collector* n'est pas implémenté.

1.2.2 Dynamisme

Une des forces de l'Objective C est son dynamisme. Le langage de Cox et Love est défini comme fortement dynamique. Cela vient en grande partie du fait que, comme expliqué dans l'introduction, son runtime s'occupe de nombreuses tâches, et cela à l'exécution.

Parlons du typage, par exemple : Celui d'Objective C est définit comme dynamique et faible. En effet, l'évaluation des types de variables est faite à l'exécution et peut être changé en cours de route. Objective C permet aussi de typer explicitement une variable. Dans ce cas là, le compilateur va nous avertir en cas de mauvaise utilisation de notre objet. Cependant, cela ne sera qu'un avertissement et non une erreur. Pour mieux illustrer ces différents typages possibles, observons l'exemple ci-dessous :

```
1 id monNom = [MaClasse maMethode];
```

Ici, le type id déclare que la variable monNom peut se référer à n'importe quelle sorte d'objet. La partie de droite ('appel de méthode') sera expliqué plus loin dans cette section.

```
1 id<monProtocol> monNom = [MaClasse maMethode];
```

La différence pour la ligne ci-dessus est que la variable doit se référer à une instance de classe conforme au protocole "mon protocole" (décrit plus loin dans cette section).

```
1  NSString* monNom = [MaClasse maMethode];
```

Notre variable doit ici pointer sur une instance de NSString. On peut remarquer ici l'ajout de l'astérisque. En effet, chaque variable d'objets Objective C est de type pointeur. Cependant, le type id est prédéfini comme pointeur. Ce type ne nécessite pas d'astérisque.

Les autres points qui font de l'Objective C un langage dynamique fort ne seront pas détaillés. Il est à noté que la réflexion est un point fort de l'Objective C, que ce soit par l'intercession (avec le typage par exemple) ou l'introspection.

1.2.3 Création d'une classe

Comme pour C++, une classe Objective C se compose de deux fichiers : un .h pour la déclaration des méthodes, variables et un .m pour l'implémentation. Ces deux différentes parties sont décrites ci-dessous, prenant pour exemple la création d'une classe "MaClasse". En général pour Objective C, on utilise la notation CamelCase pour les noms de variables, méthodes ou classes avec la première lettre en majuscule pour ces dernières.

Déclaration

Pour cette partie, il faut débuter par importer les fichiers d'en-têtes des classes et bibliothèques dont nous avons besoin. Dans le cas de notre projet, il y a beaucoup de chance que l'on doive souvent importer *Cocoa.h*. A noter qu'au contraire de C et ses *#include*, l'*#import* d'Objective C contrôle et évite d'inclure plusieurs fois les mêmes fichiers.

Par la suite, le mot clé *@interface* permet de déclarer le nom de notre classe, ici "MaClasse", et de débuter nos déclarations. A droite du nom, après un point virgule, on peut ajouter le nom de classe parent. Après cela, les différentes variables d'instances de notre classe peuvent être ajoutées entre accolades.

```
1  #import <Cocoa/Cocoa.h>
2
3  @interface MaClasse: NSObject {
4          id variable1;
5          id variable2;
6  }
7
8  + (TypeDeRetour*) methodeDeClasse: (TypeDuParametre*)parametre;
9  - (void) methodeDunObjet;
10
11 @end
```

Les déclarations de méthodes sont à ajouter après les accolades. Un + signifie que la méthode est une méthode de classe, alors que le - est pour les méthodes d'instances. Dans l'exemple ci-dessus, notre méthode d'instance ne prend aucun paramètre et ne retourne rien.

Finalement, la déclaration de notre classe se termine par le mot clé *@end*.

Implémentation

Notre fichier d'implémentation doit débuter par l'importation de notre fichier d'en-tête via *#import*.

Le mot clé utilisé pour débuter notre partie d'implémentation est ici *@implementation*. Il est suivit du nom de la classe comme pour la partie de déclaration. Par contre, pas besoin de s'occuper des variables, donc pas d'accolades.

```
#import ''MaClasse.h''

@implementation MaClasse

+ (TypeDeRetour*) methodeDeClasse: (TypeDuParametre*)parametre{
        [variable1 autorelease];
        variable1 = [parametre retain];
        return variable2;
}

- (void) methodeDunObjet{
        variable2++;
}

- (id) init{
        if(self = [super int]){
                [self methodDunObjet];
        }
        return self;
}

@end
```

Le corps de nos méthodes peut être implémenté au-dessous du nom de notre classe. Une classe possède normalement une méthode *init* pour définir les valeurs de bases de nos variables ou faire d'autres tâches de configurations. Par défaut, les variables sont définie comme *nil*. A noter que pour ce langage, la référence d'un objet à lui même est *self*, et non *this* comme en Java. Le contenu des méthodes ci-dessous n'est pas à regarder en détail pour le moment. L'utilisation de *autorelease* et *retain* sera expliqué dans la sous-section *Gestion de la mémoire*.

1.2.4 Messages

Contrairement à C,C++ ou encore Java, Objective C ne fait pas d'appel de méthode. Il envoie des messages à des objets. C'est un des points qui fait que ce langage est fortement dynamique. En effet, l'envoi de message n'est pas un "ordre" comme l'appel de méthode, mais plutôt une requête.

Prenons, par exemple, notre classe implémentée ci-dessus. Avec du java, pour appeler les méthodes de notre classe, on ferait :

```
variable = MaClasse.methodeDeClasse(parametre);
instanceDeMaClasse.methodeDunObjet();
```

Avec Objective C, les envois de messages se feront comme ci-dessous :

```
variable = [MaClasse methodeDeClasse:parametre];
[instanceDeMaClasse methodeDunObjet];
```

Messages avec plusieurs paramètres

L'utilisation de messages avec plusieurs paramètres nécessite l'ajout de label à nos paramètres. Prenons par exemple la méthode suivante :

```
- (void) setVariables:(Type1*)var1 variable2:(Type2*)var2;
```

Pour cette méthode, nos labels sont ici *setVariables* pour *var1* et *variable2* pour *var2*. Le premier paramètre aura toujours le nom de la méthode comme label. Les labels sont important pour définir le nom de notre méthode, si l'on souhaite utiliser des *selectors* (traités plus loin dans cette section).

Messages imbriqués

Pour l'envoi de messages imbriqués, le code ressemble à cela :

```
variable = [MaClasse methodeDeClasse:[objet methode]];
```

Accesseurs

Par défaut, toutes les variables des objets sont protégées. Pour offrir l'accès à ces variables, il faut créer un *setter* et *getter* pour chaque variable qui sont nos *accesseurs*. Il est définit qu'en Objective C, pour plus de clarté, le nom du *getter* sera simplement le nom de la variable (sans *get*) alors que le setter prendra comme préfixe *set*. Voici un exemple avec une variable "name".

```
- (Type*) name;
- (void) setname: (Type*)name;
```

Ces *accesseurs* peuvent être générés automatiquement via les *properties* qui sont aussi abordées plus tard dans cette section.

1.2.5 Properties

En créant des accesseurs, le code pour les *getter* et *setter* est en quelque sorte toujours répété. Pour éviter cela, Objective C possède une fonctionnalité permettant de les définir et de les créer automatiquement : Les *properties*.

Pour ajouter cette fonctionnalité, il faut modifier la partie implémentation et déclaration. Le mot clé *@property* est à ajouter avant chaque nom de variable auxquelles nous souhaitons ajouter un *getter* / *setter*. Voici un exemple pour la variable *name*.

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

@interface MaClasse: NSObject {
        id name;
}

@property (options) name;

@end
```

Entre parenthèses, après *@property*, on peut définir divers paramètres : Donner un autre nom que le nom de notre variable comme *getter* ou encore définir si il renvoie une copie de l'objet ou une simple référence.

Notre fichier MaClasse.m a alors la forme suivante ;

```
#import ''MaClasse.h''

@implementation MaClasse

@synthesize name;

@end
```

Le mot clé *@synthesize* permet de générer automatiquement les accesseurs. Les propriétés peuvent nous faire gagner beaucoup de temps et rendre notre code plus simple et plus clair.

Selectors

Objective C offre une autre possibilité intéressante au sujet des méthodes, enfin des "messages". On peut enregistré leurs noms en tant que variables et de ce fait les passer entre différents objets. Ces variables sont alors de type *SEL*.

Il existe deux façons différentes de créer ces *selectors*. Soit grâce à la fonction *NSSelectorFromString(NSStringDeLaMethode)* ou via *@selector(nomDeLaMethode)*. Dans la sous-section concernant les messages, il est mentionné que les labels sont importants pour le nom des méthodes. Prenons pour exemple la méthode *setVariables* ci-dessous :

```
- (void) setVariables:(Type1*)var1 variable2:(Type2*)var2;
```

Son nom de méthode est alors :

```
 setVariables:variable2
```

Et voici les deux manières pour obtenir son sélecteur :

Son nom de méthode est alors :

```
SEL selecteur = @selector(setVariables:variable2);

ou

NSString* nomDeMethod =  @''setVariables:variable2''
```

Le nom de la méthode ne prend deux points à la fin que si elle a un ou des paramètres.

Pour retrouver le nom d'une méthode d'après un sélecteur, il suffit d'utiliser la fonction *NSStringFromSelector(nomDuSelecteur)*.

Ces sélecteurs sont très utilisés pour envoyer des messages et surtout pour les transmettre en différents objets. Par exemple, lorsqu'un objet reçoit un message qui ne lui est pas destiné (il ne possède pas la méthode appelée), il peut transférer ce message à un autre objet. Cela offre une grande flexibilité et permet de simplifier la conception de certain composant.

1.2.6 Categories

Contrairement à C++, Objective C ne permet pas l'héritage multiple. Par contre, le langage de Cox & Love permet d'ajouter des nouvelles méthodes à des classes existantes. Ces méthodes seront alors ajoutées à chacunes des instances de la classe concernée par la catégorie, ainsi qu'à ses sous-classes.

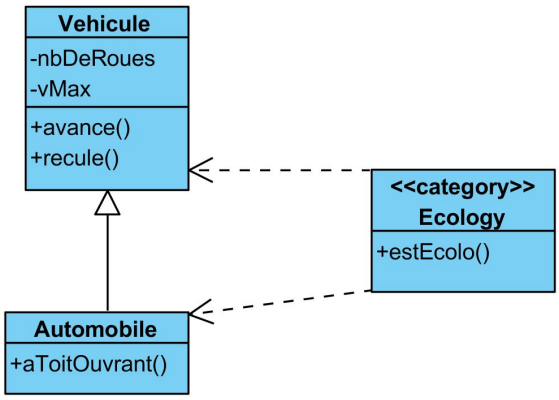

FIGURE 1.2 – Les catégories sont aussi héritées aux enfants de la classes concernée

Une des limitations des catégories par rapport aux sous-classes et le fait qu'aucune variables d'instance ne peut être ajoutée.

Généralement, on inclut le nom de la classe augmentée par la catégorie au nom des fichiers de celle-ci. Par exemple, pour le diagramme ci-dessus, les fichiers sont *VehiculeEcology.h* et *VehiculeEcology.m*.

Partie déclaration de notre catégorie :

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

@interface Vehicule (Ecology)

- (BOOL)estEcolo;
```

```
6
7   @end
```

Code du fichier *VehiculeEcolo.m* :

```
1   #import ‘‘VehiculeEcolo.h’’
2
3   @implementation Vehicule (Ecology)
4
5   - (BOOL)estEcolo
6   {
7           // Actions...
8
9           return VRAI ou FAUX
10  }
11  @end
```

La méthode *estEcolo* est disponible pour chaque instance de *Vehicule* ou une de ses sous-classes.

1.2.7 Protocoles

Le pendant Objective C des interfaces de Java ou encore des classes abstraites de C++ sont les protocoles. Ils permettent de créer et de déclarer des méthodes requises ou optionnelles à implémenter par les classes se conformant à ces protocoles. Les protocole offrent ainsi, comme pour les interfaces, la possibilité d'utiliser conjointement différents types d'objets n'ayant pas les même parents. Grâce à un protocole et à des interfaces offertes par celui-ci on peut aussi cacher les classes utilisées.

Un bon exemple d'utilisation de protocole pourrait être l'ajout d'une méthode de sauvegarde ou d'impression pour différents types de fichiers. Le programme n'aura pas besoin ne connaître la classe de l'instance à sauver, juste que celle-ci soit conforme au protocole de sauvegarde.

Un protocole se déclare dans fichier *.h* comme ci-dessous :

```
1   @protocol MonProtocol<ProtocolParent>
2
3   @required
4   -(void)methodeRequise
5
6   @optional
7   -(void)methodeOptionnel
8
9   @end
```

A noter que si une classe se conformant à un protocole n'implémente pas une méthode requise, Xcode ne générera qu'un avertissement à la compilation et non une erreur.

Si on veut qu'une classe suive un certain protocole, il suffit de déclarer celle-ci de cette façon :

```
1   @interface MaClasse<MonProtocol>
```

1.2.8 Délégués

La délégation est fortement liée aux protocoles. En effet, un objet se conformant à un protocole peut se voir déléguer certaines tâches par un autre objet. Les délégués peuvent être vus comme des contrôleurs.

Le pattern *Observer* illustre assez bien ce principe. Dans ce cas, les objets *observateurs* sont des délégués d'un objet *observé*. A chaque actions, celui-ci envoie alors des messages à tous ses délégués, qui implémentent ou non les messages reçus.

Dans la pratique, la notion de délégué est utilisée pour les différents éléments de l'interface graphique. Un champ texte peut, par exemple, avoir des délégués qui sont avertis à chaque actions, comme à la fin d'édition d'un texte.

Si l'on souhaite créer un délégué pour ce champ texte, il faut définir que notre classe se conforme au protocole *UITextViewDelegate*.

```
1 @interface Exemple_Delegue:UIViewController<UITextViewDelegate>
```

Dans ce cas, la classe *Exemple_ Delegue* hérite de la classe *UIViewController* qui est la classe de base pour tout contrôleur de l'interface graphique.

L'implémentation va dépendre des méthodes du protocole que l'on souhaite implémenter. Ci-dessous, seule la méthode appelée à la fin d'édition de texte est implémentée.

```
1 @implementation Exemple_Delegue
2
3 - (void)textViewDidBeginEditing:(UITextView*)textView
4 {
5         // Action
6 }
7
8 @end
```

1.2.9 Gestion de la mémoire

Contrairement à Max OS X, l'iOS n'a pas de *garbage collection* implémenté. Ce qui signifie que la gestion de la mémoire est nécessaire lorsqu'on programme sur iPhone ou iPad. Comme en C++, il faut veiller aux allocations et destructions d'objets.

Selon Apple [1], la règle de base pour la gestion de mémoire avec Objective C est qu'on ne relâche (via release ou autorelease) que les objets que l'on possède. On "possède" un objet si on a incrémenté son compteur : soit fait *init* ou encore *retain*. Une seconde règle de bonne pratique s'ajoute à celle-ci : On relâche tous les objets qu'on ne va plus utiliser.

Pour simplifier la gestion de la mémoire, Objective C implémente un système de compteur de référence sur chaque objet. A chaque création ou copie, ce compteur est initialisé à 1. Par la suite, on peut incrémenter ce compteur via la fonction *retain* si on ajoute une référence sur cet objet. Pour décrémenter le compteur, il existe la fonction *release*. Lorsque ce compteur tombe à 0, *dealloc* (similaire à *delete* en C++) est automatiquement appelé pour détruire notre objet.

1. http ://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html

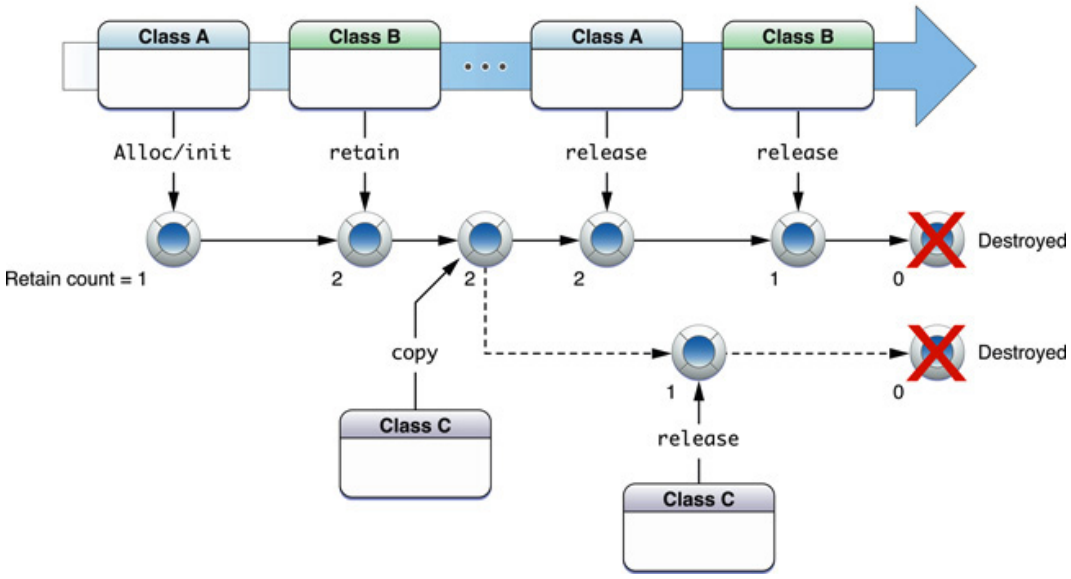

FIGURE 1.3 – Cycle de vie d'un objet avec son compteur de référence

Pour obtenir la valeur courante du compteur, il suffit d'appeler la fonction *retainCount*.
Observons le cycle de vie d'un objet au travers d'un simple exemple avec une chaîne de caractère :

```
1  // Création de l'objet
2  NSString* monObjet = [[NSString alloc] init]; // compteur = 1
3
4  // Incrémentation du compteur
5  NSString* monObjet = [NSString retain]; // compteur = 2
6
7  // Retourne la valeur courante du compteur
8  [NSString retain]; // compteur = 2
9
10 // Décrémentation du compteur
11 NSString* monObjet = [NSString retainCout]; // compteur = 1
12
13 // Décrémentation du compteur et destruction
14 NSString* monObjet = [NSString retain]; // compteur = 0, objet détruit
```

Objective C offre cependant une manière de détruire nos objets à la manière d'un pseudo garbage-collection : la fonction *autorelease*. Lorsqu'on utilise cette fonction sur un objet, on place notre objet sur l'*autorelease pool* qui est une collection d'objets devant recevoir un message de *release*. Le compteur de ces objets est alors décrémenté une fois à la fin de la partie courante du code ayant envoyé le message d'*autorelease*. Cette partie de code peut être un *thread* ou une *méthode*.

Cette fonction peut être très utile dans le cas où, dans une méthode, on crée un objet dont on va retourner la référence. Avec un message de *release*, on décrémente le compteur de l'objet trop vite, comme on doit le faire avant le *return*. Notre objet est alors détruit. Par contre, avec *autorelease*, on s'assure que le compteur soit décrémenté après que notre méthode ait retourné la référence de l'objet.

```
// Méthode avec release (non fonctionnelle)
- (NSString*)methodeExemple
{
        NSString* monObjet = [[NSString alloc] init];
        [monObjet release];
        return monObjet; // Objet peut être détruit!
}

// Méthode avec autorelease (fonctionnelle)
- (NSString*)methodeExemple
{
        NSString* monObjet = [[NSString alloc] init];
        [monObjet autorelease];
        return monObjet;
}
```

1.2.10 Compatibilité avec C, C++

Objective C est, comme expliqué dans l'introduction, une surcouche de C. Nous avons alors accès aux différents types C ainsi qu'aux bibliothèques de ce langage. D'ailleurs, Cocoa possède des classes entièrement codées en C.

Pour C++, cela est plus compliqué. En effet, ce langage reste lui aussi une sur-couche de C. Cependant, C++ possède de nombreuses bibliothèques célèbres comme OpenCV. Pour permettre de les utiliser en Objective C, une nouvelle version du langage chère à Apple a été développée : Objective C++. Mais cela reste une solution de dernier recours car cette version se place comme une couche supplémentaire au-dessus de C++. Cela devient lourd et peut vite amener des problèmes du fait que les interactions entre les classes C++ et Objective C sont impossibles.

1.2.11 iOs

L'architecture du système mobile d'Apple, iOs, est proche de celle de Mac OS X. Comme pour le système d'exploitation de bureau, elle a pour base un kernel *Mach* recouvert par 4 différentes couches :

- **Core OS :** C'est la couche la plus basse. Elle offre des interfaces pour accéder aux systèmes fichiers, au réseau, et tout autre service de bas niveau. La plupart des ces interfaces ont été programmées en C.
- **Core Services :** Comme son nom l'indique, cette couche offre différents services implémentés sur les bases de la couche précédente. Cela concerne les différents services concernant l'accès aux réseaux ou encore l'accès aux fichiers. A noter que le *Fundation framework* dont nous allons parlé dans la sous-section suivante se situe dans cette couche.
- **Media :** La gestion des différents éléments multimédia est effectuée au sein de cette couche. Cela concerne l'audio, la vidéo ou encore le graphisme 2D et 3D.
- **Cocoa Touch :** La sous-section suivante analyse cette couche en détails. Pour résumé, elle se charge de tout ce qui touche directement l'utilisateur, comme les contrôles (multi-touch, accéléromètre, ...) et les interfaces graphiques.

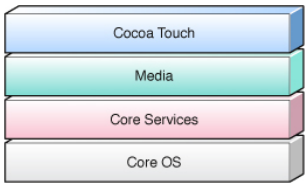

FIGURE 1.4 – Les différentes couches de l'iOS

1.2.12 Cocoa

Quand on parle de programmation en Objective C, le terme Cocoa revient souvent. Néanmoins, les définitions que l'on trouve à ce sujet sont souvent confuses. Certaines personnes décrivent Cocoa comme un écosystème pour la création d'applications pour Mac OS X, iPhone ou iPad. D'autres résument simplement Cocoa à un grand framework en en-capsulant d'autres. Rien de tout cela n'est faux, bien que cela ne soit qu'une brève description.

Origine

Pour commencer, il faut revenir à l'origine de cet API. Dans l'introduction de ce chapitre, l'API OpenSTEP pour le système d'exploitation NeXTSTEP a été brièvement cité. C'est justement cette API qui fut la base de Cocoa lorsque qu'Apple racheta NeXT. NeXTSTEP étant à la base de Mac OS X, il est logique que son API réputée performante soit aussi reprise.
Mais pourquoi cette API alors que NeXTSTEP et par conséquent Mac OS X avaient une base Unix, plus précisément BSD ? De plus, comme vu plus tôt, Objective C permet l'utilisation de différents types C. La réponse est simple : NeXT développa Cocoa afin de fournir une interface de programmation entièrement orientée objet. Cette API fournit aussi des types couramment utilisés, comme les chaînes de caractères ou les listes. D'ailleurs, ils gardent la marque de NeXTSTEP. En effet, les deux premières de leur nom sont toujours *NS*, comme *NSString*. Ces différents types sont souvent appelés value classes, ou encore Foundation value classes, du nom d'un sous-framework de Cocoa.

Composant de l'API

Cocoa ne doit pas être confondue avec la couche *Cocoa touch* de l'iOs. Bien que cette couche ne contienne que des composants de Cocoa, l'API a aussi des composants au sein des autres couches. Voici les deux composants principaux de Cocoa et leurs utilité :

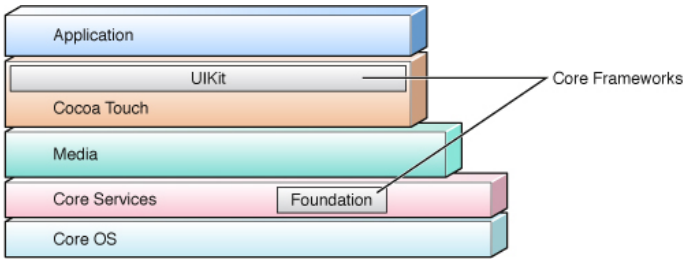

FIGURE 1.5 – Cocoa au sein de l'architecture de l'iOs.

– **Foundation** : Framework fournissant les classes pour les types de bases comme les chaînes de caractères, nombres, tableaux,...

– **UIKit (iOs) / AppKit (Mac OS X)** : Tous les éléments liés à l'interface graphique sont contenus dans ce framework.

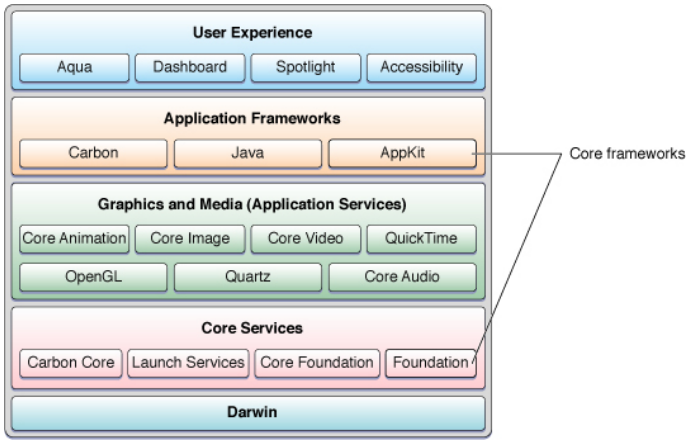

FIGURE 1.6 – Cocoa au sein de l'architecture de Mac OS X.

Un autre framework présent sur iOs et Mac OS X fait partie de Cocoa. Il s'agit du *Core Data* qui s'occupe de la persistance des données, que ce soit avec SQLite, XML ou autre.

1.2.13 Environnement de développement

Apple ayant limité la plateforme de développement à Mac OS X, c'est sur ce système que va être développée notre application. L'environnement de développement est divisé en deux outils : Xcode et Interface Builder.

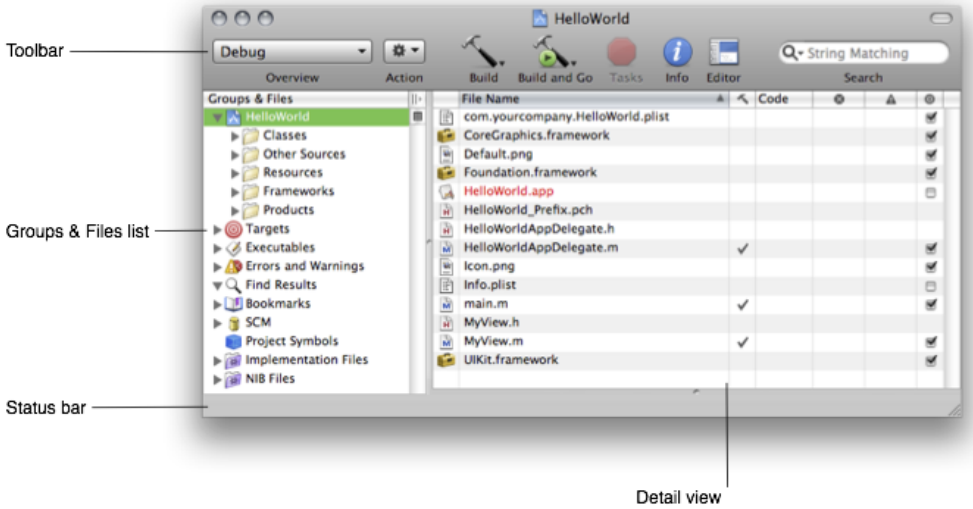

FIGURE 1.7 – Interface de Xcode

Le premier est plus basé création de code (création de projets, classes et implémentation) alors que le deuxième est, comme son nom l'indique, destiné à la création des interfaces graphiques. Les

deux outils sont complémentaires et communiquent bien entre eux. Par exemple, Xcode démarre automatiquement Interface Builder une fois que l'on souhaite modifier un fichier .xib et prend en compte directement les changements effectués dans cet outil. Par ailleurs, Apple a fusionné les deux outils pour la version 4 de leur environnement de développement.

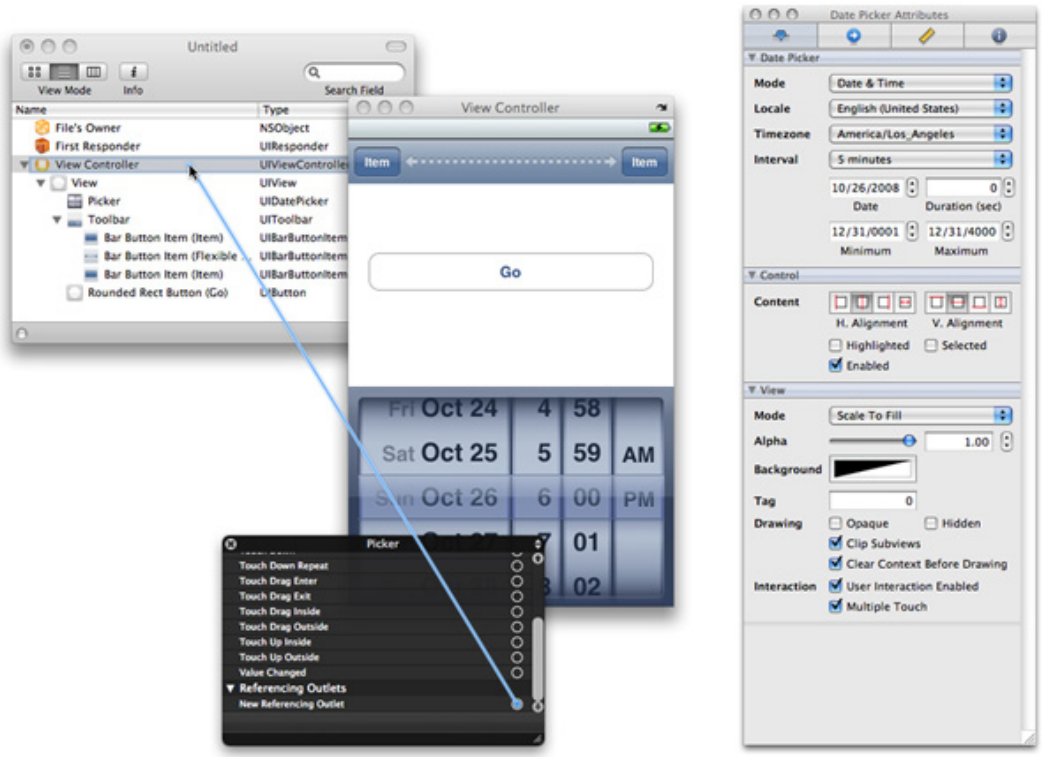

FIGURE 1.8 – Vue d'Interface Builder avec l'émulateur d'iPhone

L'environnement de développement fournit aussi d'autres outils, comme un émulateur pour iPhone et iPad, une documentation, un outil de test unitaire, ... Apple a créés un IDE plus que complet pour développer sur ses systèmes même si on pourrait lui tenir quelques reproches, notamment au niveau des faibles possibilités offertes pour le refactoring de code.

Chapitre 2

Références

1 .....
Maher Ali.
Advanced iOS 4 Programming.
Wiley, 2010.

2 .....
Apple.
ios dev center.
http ://developer.apple.com/devcenter/ios/index.action, 5 2011.

3 .....
James Bucanek.
Lean Objective-C for Java Developers.
Apress, 2009.

4 .....
Damien Vionnet Jean-Frédéric Wagen, Stéphane Pierroz.
Advanced Mobile Solutions.
Ecole d'ingénieurs et d'architecte de Fribourg, 2011.

5 .....
Scott Stevenson.
Cocoa and Objective C : up and Running, volume 1.
O'Reillly, 2010.

Table des figures