

Programmation iPhone OS 3

Conception, ergonomie,
développement et publication

Thomas Sarlandie



Programmation iPhone OS 3

**Conception, ergonomie,
développement et publication**

Ergonomie web. *Pour des sites web efficaces.*

A. BOUCHER.

N°12479, 2^e édition, 2009, 458 pages.

Bien rédiger pour le Web... *et améliorer son référencement naturel.*

I. CANIVET.

N°12433, 2009, 412 pages.

Conversion web. *Améliorer ses taux de conversion web.*

S. ROUKINE. préface de P. MOSCIUSKO-MORIZET.

N°12499, 2009, 270 pages.

La programmation orientée objet. *Cours et exercices en UML 2 avec Java 5, C#, C++, Python et PHP 5.*

H. BERSINI, I. WELLESZ.

N°12441, 4^e édition, 2009, 602 pages (collection Noire).

Gestion de projet. *Vers les méthodes agiles.*

V. MESSENGER ROTA.

N°12518, 2^e édition 2009, 272 pages (collection Architecte logiciel).

Gestion de projet eXtreme Programming.

J.-L. BÉNARD, L. BOSSAVIT, R. MÉDINA, D. WILLIAMS.

N°11561, 2002, 300 pages (collection Architecte logiciel).

Modélisation XML.

A. LONJON, J.-J. THOMASSON.

N°11521, 2006, 498 pages (collection Architecte logiciel).

Flex 3.

A. VANNIEUWENHUYZE.

N°12387, 2009, 532 pages.

Programmation Python. *Conception et implémentation.*

T. ZIADÉ.

N°12483, 2^e édition 2009, 586 pages.

Best practices PHP 5. *Les meilleures pratiques de développement en PHP.*

G. PONÇON.

N°11676, 2005, 480 pages.

Sécurité informatique. *Principes et méthode à l'usage des DSI, RSSI et administrateurs.*

L. BLOCH, C. WOLFHUGEL.

N°12525, 2009, 292 pages.

BSD, 2e édition (coll. *Cahiers de l'Admin*).

E. DREYFUS.

N°11463, 2004, 300 pages.

Debian Lenny. *Gnu/Linux.*

R. HERTZOG, R. MAS. Dirigé par N. MAKARÉVITCH.

N°12443, 2009, 442 pages avec DVD-Rom.

Asterisk. *Études de cas.* (coll. *Cahiers de l'Admin*).

P. Sultan, dirigé par N. Makarévitch.

N°12434, à paraître.

Nagios 3 pour la supervision et la métrologie. *Déploiement, configuration et optimisation.*

J. GABÈS.

N°12473, 2009, 510 pages

Symfony. *Mieux développer en PHP avec Symfony 1.2 et Doctrine.*

F. POTENCIER et H. HAMON.

N°12494, 2009, 510 pages.

Zend Framework.

G. PONÇON ET J. PAULI.

N°12392, 2008, 460 pages.

Flex 3.

L. JAYR.

N°12409, 2009, 226 pages.

Drupal 6 et 7. *Concevoir et déployer ses sites web.*

Y. BRAULT, préface d'E. PLENEL.

N°12465, 2009, 404 pages.

MediaWiki efficace.

D. BARRETT.

N°12466, 2009, 372 pages.

Réussir son blog professionnel.

T. PARISOT.

N°12514, 2009, 300 pages.

Économie du logiciel libre.

F. ELIE.

N°12463, 2009, 195 pages.

PGP/GPG

Assurer la confidentialité de ses mails et fichiers.

M. LUCAS, ad. par D. GARANCE, contrib. J.-M. THOMAS.

N°12001, 2006, 248 pages.

Programmation iPhone OS 3

**Conception, ergonomie,
développement et publication**

Thomas Sarlandie

EYROLLES



ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Remerciements à Fabrice Bernhard pour sa relecture précieuse.



Le code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 2009, ISBN : 978-2-212-12477-4

Avant-propos

Peu de plates-formes peuvent se vanter d'avoir eu autant de succès et d'avoir tant changé les mentalités que l'iPhone.

Travaillant dans le monde mobile, j'ai eu la chance de voir ce média passer de l'état de curiosité technologique réservé à quelques technophiles à celui de nouveau compagnon du quotidien pour des millions de personnes.

Lorsque nous avons créé il y a trois ans Backelite, une société spécialisée dans la conception et la réalisation de services mobiles, l'« *iPhone* » était une légende persistante sur Internet et personne ne pouvait imaginer l'impact formidable qu'il aurait sur notre industrie.

Avec son design exceptionnel, une ergonomie révolutionnaire et les premiers forfaits de données illimités, l'iPhone est arrivé à un moment charnière de l'histoire du mobile et a ouvert une nouvelle ère.

En 2008, Apple annonçait le SDK iPhone, permettant à tous les développeurs de réaliser eux-mêmes leurs applications. À peine un an plus tard, le succès est au rendez-vous, à tel point que le SDK est devenu un aspect indissociable de l'iPhone.

Pourquoi ce livre ?

Le développement d'applications iPhone est souvent perçu comme un univers sombre et inquiétant : le langage Objective-C est pratiquement inconnu hors du monde des développeurs Mac, les plates-formes mobiles ont la réputation d'être complexes pour les développeurs (ce qui était vrai) et les rumeurs sur le mode de validation des applications par Apple finissent de refroidir les candidats potentiels.

Pourtant, il existe dans le monde mobile un potentiel d'innovation gigantesque encore largement sous-utilisé.

Ce livre doit permettre de dissiper les inquiétudes et aider à cultiver le potentiel qui se cache derrière chaque développeur talentueux.

Il est aussi là pour montrer aux développeurs que le succès ne viendra pas uniquement de la technique, mais aussi de l'effort fait sur la conception de l'interface, sur la réflexion ergonomique et sur les efforts marketing fournis pour accompagner le lancement.

À qui est destiné ce livre ?

Ce livre est destiné à tous ceux qui participent à un projet d'application iPhone.

Les développeurs y trouveront une introduction à l'Objective-C (le langage imposé pour développer sur iPhone), le SDK fourni par Apple, ses outils comme Xcode et Interface Builder et la présentation des principaux concepts utilisés pour développer des applications.

Les experts métier, les consultants marketing et les équipes créatives y trouveront une présentation des principes ergonomiques fondamentaux de la plate-forme, une proposition de méthode pour concevoir une ergonomie mobile et des conseils pour préparer la publication et le lancement de l'application.

Organisation du propos

Ce livre est découpé en cinq parties de difficulté croissante et conçues pour être lues dans l'ordre par un développeur expérimenté découvrant la plate-forme iPhone et le monde Apple. Les parties deux et cinq sont destinées à l'ensemble de l'équipe et ont été écrites dans un langage moins technique.

Première partie : la découverte de l'environnement de développement

Le **chapitre 1** est destiné à toute personne intéressée par le développement iPhone. Il présente les différents programmes développeurs, l'inscription et le téléchargement des outils et documentations. Le développeur y trouvera également des explications pour créer un certificat de développement, indispensable pour tester son application sur un iPhone.

Le **chapitre 2** est une introduction à l'Objective-C. Destiné à des développeurs familiers de la programmation orientée objet, il présente le langage d'une façon très pragmatique et destinée à vous rendre opérationnel rapidement.

Le **chapitre 3** enfin permet au développeur de faire sa première application et de la tester dans le simulateur et sur son iPhone. Les outils indispensables comme Xcode

et Interface Builder sont introduits en suivant quelques exemples très simples (« Hello, World »).

Deuxième partie : la conception et l'ergonomie

La deuxième partie du livre s'adresse à toute l'équipe en charge de l'application, les développeurs, les experts métier, les créatifs (graphistes).

La plupart des équipes sont habituées au développement d'applications web ou client lourd. Le **chapitre 4** présente ainsi le cycle de développement d'un projet iPhone, ses spécificités, les différentes phases et le rôle de chaque intervenant. Il permet de partager un cadre méthodologique qui sera adapté au contexte de l'équipe en charge du projet et à ses habitudes.

Le **chapitre 5** donne les bases de l'ergonomie iPhone que toute l'équipe doit maîtriser. Les conventions iPhone, leur logique et leur utilisation au sein des applications standard sont détaillées car il est essentiel de bien les avoir comprises pour concevoir l'interface d'une nouvelle application.

C'est justement le sujet du **chapitre 6** qui propose une méthode permettant de partir d'une liste de fonctionnalités pour arriver à une interface utilisateur. Nous verrons comment exploiter les éléments standard pour créer une nouvelle application et comment éviter les erreurs classiques.

Troisième partie : le développement de l'interface

La troisième partie est consacrée au développement de l'interface de l'application.

La brique principale de construction d'une application iPhone est le contrôleur de vue qui est expliqué en détail au **chapitre 7**.

L'assemblage des contrôleurs de vue pour construire une application complète, dont les écrans s'enchaînent de manière fluide, est détaillé dans le **chapitre 8**.

Le **chapitre 9** présente les vues, c'est-à-dire tous les éléments d'interface, il donne les clés pour bien comprendre comment assembler un écran à partir des composants graphiques fournis par le SDK et comment les adapter aux couleurs de votre application.

Enfin, au **chapitre 10**, on présentera les tables, qui sont les éléments les plus utilisés dans la plupart des applications, et qui permettent de faire des listes et toutes sortes d'écrans dans lesquels l'utilisateur navigue verticalement avec le doigt.

Quatrième partie : la manipulation des données

Cette partie présente les différentes techniques à utiliser pour lire et enregistrer des données, ainsi que pour permettre à votre application de communiquer avec l'extérieur.

La sérialisation des classes de base (listes et dictionnaires) en XML ou en JSON sera présentée au **chapitre 11**, ainsi que l'utilisation du mécanisme des préférences utilisateur et les techniques de lecture d'un flux XML quelconque.

Le **chapitre 12** montre comment combiner ces techniques avec des appels réseau, et en particulier comment faire en sorte que l'application reste réactive même quand des appels réseau prennent plusieurs secondes.

Le framework CoreData, une nouveauté de l'iPhone OS 3, est présenté dans le **chapitre 13**. C'est un framework complet de mapping objet-relationnel pour l'iPhone et incontestablement le moyen le plus efficace d'enregistrer et de parcourir des volumes importants de données métier.

L'utilisation de contenus multimédias est couverte par le **chapitre 14** : lecture de sons, utilisation de la bibliothèque iPod de l'utilisateur, de la caméra ou encore lecture de vidéos.

Cette partie se termine par un **chapitre 15** sur le mécanisme des notifications qui est également une nouveauté de l'iPhone OS 3 et qui permet de rester en contact avec l'utilisateur, même quand l'application est éteinte. La mise en place des notifications est présentée, de manière détaillée, avec des exemples en PHP pour la partie serveur qui pourront facilement être adaptés dans n'importe quel langage.

Cinquième partie : la publication des applications

Enfin, la dernière partie de ce livre ne contient qu'un **chapitre 16**, consacré à la publication de l'application. Il est destiné à toute l'équipe projet qui y trouvera la liste de tous les éléments à préparer avant de pouvoir soumettre l'application, des conseils pour favoriser les chances que l'application soit validée du premier coup, et quelques outils utiles pour suivre le succès de votre application et préparer la prochaine version.

Remerciements

Derrière un nom se cache le résultat d'un an de travail de toute l'équipe Backelite sur la plate-forme iPhone. C'est à travers d'innombrables conversations sur des sujets techniques, créatifs, fonctionnels et parfois un peu métaphysiques que se sont construits les concepts et les idées présentés dans ce livre. Un grand merci à toute l'équipe !

Backelite n'existerait pas sans la grande confiance dont nous ont honoré nos clients. Chacun d'eux a apporté son lot d'exigences et de challenges qui nous ont permis d'explorer de nouvelles pistes, de découvrir de nouvelles solutions. Mes pensées vont vers eux et en particulier Xavier, Cédric, Hélène, Nicolas, Edouard, Charlotte et Régine qui ont su nous écouter et nous faire confiance durant nos premiers pas sur mobile et iPhone.

Je pense aussi bien sûr à toute l'équipe Apple qui a su mettre tant de passion et de soin du détail dans l'iPhone. Il suffit de se rendre à la conférence annuelle à San Francisco pour constater que, comme nous, les ingénieurs Apple sont des passionnés.

Enfin je tiens à remercier :

- Frédéric Fourdrinier, responsable de la spécialisation SIGL à l'EPITA, ses nombreux conseils continuent de rythmer mes journées ;
- Michel Sutter, responsable du support aux développeurs Apple pour l'Europe qui ne ménage pas ses efforts pour animer la communauté ;
- les frenchies de la liste wwdc@mac4ever, avec qui j'ai passé une semaine inoubliable ;
- Muriel des Éditions Eyrolles, experte en maïeutique des livres et dont l'expérience a été indispensable tout au long de l'écriture de ce livre, ainsi que toute son équipe – Anne, Sophie, Gaël ;
- Fabrice, relecteur *presque* novice dont les retours ont permis de clarifier les passages semblant difficiles ;
- Romain, Raphaël et Victoire, l'équipe créative de Backelite qui a su imaginer des interfaces originales, guider les développeurs et exiger le meilleur ;
- François, Philippe et Maxime de l'équipe technique qui ont relu ce manuscrit ligne par ligne, vérifié chaque exemple de code et m'ont apporté leurs précieux retours ;
- Martin, notre responsable commercial, sponsor de ce projet depuis le premier jour ;
- Yann dont l'expérience et les nombreux conseils ont éclairé les trois dernières années, merci de ta confiance ;
- et bien sûr Sébastien et Vincent, mes associés. Ensemble, nous avons donné au verbe entreprendre tout son sens.

Alors que ce livre va partir à l'impression, mon dernier mot est pour Beth : Oui !

Table des matières

Avant-propos	V
---------------------------	----------

PREMIÈRE PARTIE

Découverte de l'environnement de développement ..	1
--	----------

CHAPITRE 1

Développer pour iPhone	3
-------------------------------------	----------

Équipement matériel requis	3
----------------------------------	---

Un Mac Intel pour développer	3
------------------------------------	---

<i>ALTERNATIVE Développer sans Mac ?</i>	4
--	---

Un iPhone ou un iPod Touch pour tester l'application	4
--	---

<i>CONSEIL Utilisez intensément votre iPhone</i>	4
--	---

Compétences techniques utiles au développeur iPhone	4
--	----------

La programmation orientée objet, au cœur du SDK iPhone	4
--	---

L'Objective-C : un langage comme un autre	5
---	---

<i>VOUS VENEZ D'AUTRES LANGAGES Pour les développeurs Java, PHP et C#</i>	5
---	---

Programmation multithread	5
---------------------------------	---

<i>VOUS VENEZ D'AUTRES LANGAGES Synchronisation de threads</i>	5
--	---

Développement d'un « client lourd »	5
---	---

L'adhésion au programme développeur d'Apple	6
--	----------

Développeur iPhone enregistré : un accès gratuit à l'environnement de développement et à la documentation	6
---	---

Le programme Standard pour tester et publier vos applications	7
---	---

<i>CONSEIL Développer pour un tiers</i>	7
---	---

<i>Les deux modes d'adhésion au programme Standard</i>	7
--	---

<i>Le processus d'adhésion au programme Standard</i>	7
--	---

<i>ATTENTION Ne pas confondre le programme Standard et le programme Entreprise</i>	7
--	---

<i>CONSEIL N'hésitez pas à contacter le support Apple aux développeurs</i>	8
--	---

Le programme Entreprise pour des applications internes	8
--	---

Le programme universitaire pour l'enseignement	8
--	---

Les sites web Apple pour le développeur iPhone	9
--	---

Le centre de développement iPhone	9
Le portail du programme iPhone	9
iTunes Connect, pour la publication des applications	10
Présentation du SDK iPhone	11
REMARQUE <i>Installation du SDK iPhone sous Snow Leopard</i>	12
La documentation Apple, une aide à ne pas négliger	12
Les guides pour le développeur	13
<i>Le guide consacré aux règles à respecter en matière d'ergonomie.</i>	13
<i>La description des API et de la bibliothèque graphique</i>	13
<i>La référence Objective-C.</i>	13
Les exemples de code : des projets Apple exemplaires	13
La documentation de référence exhaustive	13
ASTUCE <i>Avoir toujours la documentation à jour</i>	14
Pré-requis pour la distribution d'une application	14
RAPPEL <i>Signature électronique et certificat</i>	14
Préparation à la diffusion d'une application en test (mode Ad Hoc)	15
ATTENTION <i>Les places d'identifiants sont précieuses</i>	15
<i>Générer un certificat de développeur</i>	15
<i>Faire signer par Apple la demande de certificat</i>	17
<i>Créer un identifiant d'application.</i>	18
ATTENTION <i>Le certificat développeur est lié à la machine du développeur</i>	18
<i>Définir les iPhone autorisés.</i>	19
ASTUCE <i>Obtenir l'UDID d'un iPhone</i>	19
<i>Générer le profil de provisionnement.</i>	20
<i>Installer le profil sur des iPhone</i>	20
<i>Installer l'application sur des iPhone.</i>	21
CONSEIL <i>Vérifications à faire en cas de problème</i>	22
Mode de distribution via l'App Store pour une diffusion large	22
<i>Publication sur l'AppStore.</i>	23
Conclusion	23

CHAPITRE 2

L'essentiel d'Objective-C..... 25

APPROFONDIR <i>Guide de programmation Objective-C.</i>	25
Les origines	25
Principes fondamentaux	26
Tout est objet	26
Envoi de messages	27
La syntaxe Objective-C	27
Envoyer des messages à un objet	27
ATTENTION <i>Signature de méthode en Objective-C.</i>	28

Le type id et la valeur nil	28
REMARQUE <i>id est un pointeur</i>	28
Déclaration d'une classe	29
<i>Les variables d'instance</i>	30
ATTENTION <i>Ne pas confondre variable d'instance et propriété</i>	30
<i>Les différents types de méthodes</i>	30
REMARQUE <i>Méthode de classe et méthode statique</i>	30
<i>Héritage</i>	31
<i>Faire référence à l'objet courant et à son père</i>	31
Initialisation d'une instance d'objet	31
Les protocoles	32
<i>Déclaration d'un protocole</i>	32
<i>Implémenter un protocole</i>	33
Les propriétés	33
<i>Déclarer une propriété</i>	33
<i>Implémenter les accesseurs</i>	34
<i>Attributs des propriétés</i>	34
ATTENTION <i>Les propriétés ne sont que du sucre syntaxique</i>	34
ASTUCE <i>L'attribut nonatomic</i>	35
La notation point	35
Les sélecteurs : des pointeurs sur fonction	35
Synchronisation de threads	36
La bibliothèque standard : le framework Foundation	36
Chaînes de caractères	36
APPROFONDIR <i>Le spécificateur de formatage %@</i>	37
APPROFONDIR <i>Objets mutables</i>	38
Listes	38
<i>Parcourir rapidement une liste</i>	38
Dictionnaire	39
Le mécanisme de comptage de références	39
RAPPEL <i>De l'importance d'une bonne gestion de la mémoire</i>	39
La gestion manuelle de la mémoire : point de ramasse-miettes !	40
RAPPEL <i>Garbage collector ou ramasse-miettes</i>	40
Vie et mort des objets	40
<i>Création d'un objet</i>	40
<i>Libération de la mémoire d'un objet</i>	40
<i>Libération de la mémoire utilisée par vos objets</i>	41
Risques d'une mauvaise gestion de la mémoire	41
<i>Envoyer un message à un objet dont la mémoire a été libérée</i>	41
<i>Libérer plusieurs fois la mémoire d'un objet</i>	41
ASTUCE <i>Toujours mettre à nil les références d'objets après les avoir libérés</i>	42

Comptage de références	42
<i>Incrémenter et décrémenter le compteur de références</i>	42
ATTENTION <i>Utilisation de la méthode retainCount</i>	43
<i>Destruction des objets</i>	43
<i>Libération retardée d'un objet</i>	43
APPROFONDIR <i>Les pools d'autorelease</i>	44
Règles de gestion de la mémoire en Objective-C	45
CONSEIL <i>Relisez trois fois le paragraphe ci-dessus</i>	45
Application de la règle	45
Utilisation des objets à libération retardée	45
Propriétés et gestion de la mémoire	46
ATTENTION <i>Ne confondez pas la propriété et la variable d'instance</i>	47
<i>Propriétés non retenues et références faibles</i>	48
APPROFONDIR <i>Les références faibles</i>	48
Conclusion	48

CHAPITRE 3

Premiers pas avec le SDK iPhone 49

À la découverte de Xcode	49
Les modèles de projet	50
ESSENTIEL <i>Fenêtre et vues</i>	50
Découverte d'un projet iPhone vide	51
<i>Avant toute chose</i>	51
<i>Les groupes de fichiers créés par Xcode</i>	52
ASTUCE <i>Les raccourcis clavier essentiels</i>	52
<i>Fichiers générés lors de la création du projet</i>	53
Comprendre le code généré par Xcode	54
ESSENTIEL <i>Boucle de gestion des événements</i>	54
Rôle du fichier Info.plist	55
<i>Édition du fichier Info.plist</i>	55
ASTUCE <i>Édition des fichiers XML de propriétés</i>	55
<i>Nom de l'application et choix de l'icône</i>	55
<i>Personnalisation de l'écran de démarrage</i>	56
Rôle du délégué de l'application	56
DESIGN PATTERN <i>Délégation de contrôle</i>	56
<i>applicationDidFinishLaunching: – Votre application a fini de se lancer</i>	57
CONSEIL <i>Limitez au maximum les traitements lors du lancement de l'application</i>	57
<i>applicationWillTerminate: – Votre application va se terminer</i>	57
CONSEIL <i>Prévoyez du temps avant que l'application se termine</i>	57
<i>applicationDidReceiveMemoryWarning: – Votre application doit libérer de la mémoire</i>	58

ASTUCE	<i>Utiliser les classes fournies par le SDK.</i>	58
	<i>applicationWillResignActive: – Votre application passe en arrière-plan</i>	58
BONNE PRATIQUE	<i>Enregistrer l'état de l'application</i>	59
	Rôle d'Interface Builder	59
ESSENTIEL	<i>Comprendre Interface Builder</i>	59
	<i>Découverte d'Interface Builder</i>	59
ASTUCE	<i>Utiliser toujours la vue en liste dans Interface Builder</i>	60
CONSEIL	<i>Aux allergiques du glisser-déplacer.</i>	62
	<i>Déclaration du lien entre l'application et son délégué</i>	63
ESSENTIEL	<i>Interface Builder – File's Owner</i>	63
	Hommage à MM. Kernighan et Ritchie	63
	Utilisation de la console	64
CONSEIL	<i>Usez et abusez de la console.</i>	64
	Création d'une nouvelle vue avec Interface Builder	65
	<i>Ajouter une nouvelle vue au projet</i>	65
	<i>Faire appel à la vue</i>	65
ESSENTIEL	<i>La classe UIViewController et le design pattern MVC</i>	66
ERREUR CLASSIQUE	<i>Ne pas oublier de lier la vue</i>	66
	Création d'une nouvelle vue en code	68
	<i>Créer une vue</i>	69
ESSENTIEL	<i>Taille de l'écran</i>	69
	<i>Créer un label</i>	69
	<i>Assembler la vue</i>	69
	<i>Libérer la mémoire</i>	70
	Lancer l'application dans iPhone	70
	Définir les paramètres de signature de l'application	70
	Compiler et lancer l'application sur iPhone	71
	Et maintenant ?	71

DEUXIÈME PARTIE

Conception et ergonomie73

CHAPITRE 4

Méthode de développement d'un projet iPhone 75

Qu'est-ce qu'un projet d'application réussi ? 75

Satisfaire les utilisateurs 75

Maîtriser le projet 76

Les étapes du projet 76

Identifier les fonctionnalités clés 76

CONSEIL *La réflexion fonctionnelle précède celle sur l'interface.* 77

EXEMPLE APPLE *Ne pas adapter sur iPhone toutes les fonctionnalités : Photo versus iPhoto* 77

Définition des principes ergonomiques	77
Story-boarding et spécifications	78
MÉTHODE <i>Mise en œuvre d'une méthode agile</i>	79
Intervention artistique	79
CONSEIL <i>Testez votre design</i>	80
ASTUCE <i>Gagner du temps en réutilisant des bibliothèques de composants</i>	80
Développement de l'interface	80
Développement de l'application	81
MÉTHODE <i>Développement piloté par les tests</i>	81
Tests et optimisation	81
Publication	82
Conclusion	82

CHAPITRE 5

Principes ergonomiques et design patterns d'interface 83

L'ergonomie dans l'univers de l'iPhone	84
Une interface différente basée sur des métaphores	84
APPROFONDIR <i>Le guide ergonomique iPhone</i>	85
Les design patterns d'interface	85
ESSENTIEL <i>Qu'est-ce qu'un design pattern d'interface ?</i>	85
Deux bonnes raisons pour s'aligner sur l'ergonomie de l'iPhone	85
<i>Faciliter la prise en main de l'application</i>	85
<i>Accélérer les développements</i>	86
REMARQUE <i>Les jeux sont une exception</i>	86
Développer des applications originales	86
EN RÉSUMÉ	86
Des applications pour consulter et manipuler des données	86
CONSEIL <i>Ne négligez pas les applications utilitaires</i>	87
Les métaphores de l'interface iPhone	87
Les listes d'éléments	87
<i>Comportements communs aux listes</i>	87
<i>Utilisation de sections dans une liste</i>	88
<i>Ajout d'un index sur une liste</i>	88
<i>Accessoires des éléments et indicateurs de détails</i>	88
<i>Le mode édition</i>	90
Les listes groupées	90
<i>Édition d'une liste groupée</i>	91
Les design patterns d'interface iPhone	91
Navigation dans une application utilitaire	91
Utilisation d'une barre d'outils	91
Navigation dans des listes hiérarchiques	92

<i>La barre de navigation</i>	93
<i>La vue de contenu</i>	94
<i>Les animations</i>	94
<i>Principes de navigation à respecter</i>	94
Navigation par onglet	95
POINT D'ATTENTION <i>De la bonne utilisation des onglets</i>	95
<i>Utilisation en combinaison avec d'autres design patterns d'interfaces</i>	95
REMARQUE <i>Masquer la barre d'onglets</i>	96
<i>Personnalisation de la barre d'onglets</i>	96
Passons à la conception	97

CHAPITRE 6

Conception de l'interface graphique 99

Utilisation d'une application mobile	99
Temps et fréquence d'utilisation	99
BEST PRACTICE <i>L'application doit se lancer rapidement</i>	100
BEST PRACTICE <i>L'application doit gérer les interruptions</i>	100
Concentration et attention disponible	100
CONSEIL <i>L'interface doit rester simple, même pour des fonctionnalités riches</i>	101
Méthode pour concevoir l'interface d'une application	101
Identifier les fonctionnalités	101
Trier les fonctionnalités par ordre d'importance	102
<i>Les trois groupes de fonctionnalités</i>	102
<i>De l'importance des trois groupes</i>	102
CONSEIL <i>La dure sélection des fonctionnalités</i>	103
Concevoir l'interface pour ses fonctionnalités	103
<i>Les fonctionnalités du premier groupe doivent être accessibles en un temps minimum</i>	103
EXEMPLE <i>L'application Facebook</i>	103
<i>Mise en avant des fonctionnalités du deuxième groupe</i>	104
EXEMPLE <i>Application iTunes</i>	104
<i>Fonctionnalités du troisième groupe</i>	104
EXEMPLE <i>Chargement de Omnifocus</i>	104
EXEMPLE <i>Utilisation de l'accéléromètre dans les applications</i>	105
Quelques outils pour concevoir l'interface	105
Un accès unique à chaque fonctionnalité	105
CONSEIL <i>Faites des choix</i>	105
Éviter les menus et autres listes de fonctionnalités	106
Distinguer le premier lancement des suivants	106
EXEMPLE <i>L'assistant de configuration de l'application Mail</i>	106
Adapter l'interface à l'utilisateur	106

<i>Mémoriser le dernier écran utilisé</i>	106
<i>Proposer à l'utilisateur de personnaliser la barre d'onglets</i>	107
EXEMPLE <i>Application Horloge</i>	107
Paramètres de l'application	107
Conclusion	107

TROISIÈME PARTIE

Le développement de l'interface 109

CHAPITRE 7

Contrôler les écrans de l'application 111

Le modèle MVC dans iPhone OS	111
Le modèle pour charger et stocker en mémoire les données de l'application ..	112
La vue pour représenter graphiquement le modèle et fournir l'interface graphique	113
Le contrôleur pour lier le modèle et la vue	113
Le contrôleur de vue standard d'iPhone OS	113
Cycle de vie d'un contrôleur de vue	114
<i>Contrôleur initialisé sans vue</i>	114
<i>Vue chargée, non affichée</i>	115
APPROFONDIR <i>Comment fonctionne le chargement retardé de la vue ?</i>	115
<i>Vue chargée et affichée</i>	115
<i>Avertissement de mémoire</i>	116
ESSENTIEL <i>Prévoir le cas de la destruction automatique des vues</i>	116
Utilisation des contrôleurs de vue	117
Création d'un nouveau contrôleur de vue	117
Instanciation d'un contrôleur de vue	117
<i>Créer un contrôleur de vue sans fichier NIB</i>	117
<i>Créer un contrôleur de vue utilisant un fichier NIB</i>	118
IMPORTANT <i>Bien gérer la mémoire dans les contrôleurs de vue</i>	118
BEST PRACTICE <i>Un fichier XIB pour un contrôleur</i>	119
Réagir au chargement et au déchargement de la vue	120
<i>Utilisation de la méthode viewDidLoad</i>	120
<i>Implémentation de la méthode viewDidUnload</i>	121
<i>Comment savoir si la vue est chargée ?</i>	121
Réagir lorsque la vue est affichée puis masquée	122
<i>Affichage de la vue</i>	122
<i>Masquage de la vue</i>	122
Gérer les événements	123
<i>Créer une méthode pour traiter l'événement</i>	123

<i>Lier un événement à une action</i>	123
Gérer les rotations d'écran	124
APPROFONDIR <i>Comment la vue est-elle adaptée lors d'une rotation ?</i>	125
<i>Événements associés aux rotations d'écran</i>	125
APPROFONDIR <i>Les rotations de contrôleur de vue</i>	126
Conclusion	126

CHAPITRE 8

Assembler les écrans de l'application 127

Généralités sur les contrôleurs-conteneurs	127
CONSEIL <i>Évitez de créer d'autres contrôleurs-conteneurs</i>	128
Le contrôleur de navigation	128
Création d'un contrôleur de navigation	129
Spécifier le contenu de la barre de navigation	129
<i>Titre du contrôleur</i>	130
<i>Boutons supplémentaires</i>	131
BEST PRACTICE <i>Ne pas changer le bouton de gauche</i>	131
<i>Définir la façon dont est représenté le contrôleur quand il n'est plus affiché</i> ...	131
CONSEIL <i>Aidez l'utilisateur à retrouver son chemin</i>	131
<i>Masquer la barre d'outils ou la barre d'onglets</i>	132
Pousser des écrans dans le contrôleur de navigation	132
APPROFONDIR <i>Quand utiliser le paramètre <code>animated</code></i>	132
Personnaliser la barre de navigation	133
Contrôleur d'onglets	133
Création d'un contrôleur d'onglets	134
Personnalisation du titre et de l'icône des onglets	134
Réagir aux événements de la barre d'onglets	135
<i>Suivre les changements de sélection</i>	135
<i>Réagir à la personnalisation de la barre</i>	135
BEST PRACTICE <i>Enregistrement de l'état de la barre d'onglets dans les préférences de l'utilisateur</i>	136
Limiter la personnalisation de la barre d'onglets	136
Combiner les contrôleurs d'onglets avec des contrôleurs de navigation	136
Affichage d'un contrôleur en mode modal	137
Pousser une nouvelle vue modale	137
Faire disparaître une vue modale	137
Définir le mode de transition	138
Conclusion	138

CHAPITRE 9

Développer et animer les vues..... 139

Comprendre les vues	139
---------------------------	-----

Coordonnées des vues	139
<i>Centre et limites d'une vue</i>	140
ATTENTION <i>Origine des coordonnées</i>	140
ATTENTION <i>Limites de la vue</i>	140
<i>Frame d'une vue</i>	141
POINT D'ATTENTION <i>Bonne utilisation de la propriété frame</i>	141
Hierarchie des vues	142
Positionnement des sous-vues et redimensionnement automatique des vues ..	142
Les vues élémentaires de UIKit	143
Les labels pour afficher du texte	143
ASTUCE <i>Couleur et transparence</i>	144
Les vues d'images	144
CONSEIL <i>Utilisez le format PNG</i>	144
Les boutons pour déclencher des actions	145
Les zones de texte	145
Affichage de contenus web dans l'application	146
Animation des vues	147
Conclusion	148

CHAPITRE 10

Listes d'éléments..... 149

Les deux types de listes	149
<i>Les listes simples</i>	149
<i>Les listes groupées</i>	150
Créer une tableView	150
Fournir des données à une tableView	151
Indiquer le nombre de lignes	151
Afficher des données	152
REMARQUE <i>Les styles de cellule</i>	153
Définir les en-têtes et pieds de section	153
Réagir aux actions sur la liste	154
Sélection d'un élément	154
Édition dans une tableView	154
Techniques pour afficher des cellules personnalisées	154
Composition de la cellule	155
ASTUCE <i>Utilisation des identifiants de vue</i>	156
Utilisation d'Interface Builder pour concevoir les cellules	156
Dessiner manuellement le contenu de la cellule	157
POUR APPROFONDIR <i>Dessiner manuellement le contenu de la cellule</i>	157
Un exemple complet	157
Conclusion	160

QUATRIÈME PARTIE

La manipulation des données 161

CHAPITRE 11

Lire et enregistrer des données 163

Les préférences utilisateur	163
Obtenir une instance des préférences utilisateur	164
Enregistrer une valeur dans les préférences	164
Lire les valeurs des préférences	164
Permettre à l'utilisateur de modifier directement les préférences	165
<i>APPROFONDIR</i> <i>Mise en place de pages de préférences utilisateur</i>	166
Les fichiers de propriétés	166
Le format plist	166
Lire un fichier de données plist	168
<i>ASTUCE</i> <i>Obtenir le chemin complet d'un fichier de ressource</i>	168
Écrire un fichier de données plist	168
<i>ATTENTION</i> <i>Savoir où enregistrer vos données.</i>	169
Le format de données JSON	169
Tirer partie de JSON dans vos applications	170
Lire des données JSON	170
Enregistrer des données JSON	170
Manipuler des données XML	171
<i>RAPPEL</i> <i>SAX et DOM.</i>	171
Création d'un parseur XML	171
Gérer les événements XML	172
<i>Début du document</i>	173
<i>Début d'un élément.</i>	173
<i>Récupérer le contenu texte des éléments.</i>	174
<i>Repérer la fermeture d'un élément et enregistrer son contenu</i>	175
Conclusion	175

CHAPITRE 12

Communiquer avec l'extérieur..... 177

<i>RAPPEL</i> <i>Connexions synchrones et asynchrones</i>	177
Premiers appels réseau synchrones	178
Modifier le comportement d'une requête synchrone	179
<i>Authentification</i>	179
<i>Gestion des redirections.</i>	179
<i>Définir le délai d'attente d'une requête</i>	179
Réaliser des traitements en arrière-plan	179

Comprendre le thread principal	180
ATTENTION <i>Ne jamais exécuter de traitements longs sur le thread principal</i>	180
Lancer un traitement en arrière-plan	180
Particularités des traitements en arrière-plan	181
<i>Mise en place d'un nouveau pool d'autorelease.</i>	181
<i>Interactions avec l'interface.</i>	181
ATTENTION <i>Ne jamais modifier l'interface depuis un thread d'arrière-plan.</i>	182
BEST PRACTICE <i>Allumer l'indicateur d'activité réseau.</i>	183
Connexions réseau asynchrones	183
Préparation d'une requête asynchrone	183
REMARQUE <i>Le délégué de NSURLConnection implémente un protocole informel</i>	184
Établissement de la connexion	184
Réception de données	185
Fin de connexion	185
Conclusion	186
CONSEIL <i>Mise en place de caches de données</i>	186

CHAPITRE 13

Persistance d'objets avec CoreData **187**

APPROFONDIR <i>Les frameworks ORM</i>	187
Introduction à l'ORM	188
Du monde objet au monde relationnel	188
Gestion des relations	188
Performances	189
Notion de contexte ou de session	189
Mise en place de l'environnement Core Data	189
RAPPEL <i>Ne pas oublier d'ajouter Core Data à votre projet</i>	190
Chargement de la description du modèle	190
Mise en place de l'entrepôt de stockage des données	191
Création du contexte	192
Description du modèle	193
ASTUCE <i>Séparer le modèle en plusieurs fichiers de description</i>	193
Création d'un nouveau modèle	194
Édition du modèle	194
APPROFONDIR <i>Utilisation de Xcode pour concevoir le modèle</i>	194
Création des classes du modèle	195
Manipulation d'objets gérés par le contexte	196
Création d'une nouvelle instance	196
Enregistrement des objets du contexte	197
Exécution d'une requête pour obtenir des objets	197
<i>Recherche d'une entité dans la base</i>	198

<i>Recherche basée sur un prédicat</i>	198
APPROFONDIR <i>Maîtriser les prédicats</i>	198
<i>Définir l'ordre des objets renvoyés</i>	198
<i>Aller plus loin avec les requêtes</i>	199
Supprimer un objet	199
Conclusion	199

CHAPITRE 14

Manipuler des données multimédias 201

Intégrer le son au cœur de vos applications	201
Les formats audio pris en charge par l'iPhone	202
ATTENTION <i>Les limites du hardware audio de l'iPhone</i>	202
Convertir les fichiers audio pour l'iPhone	202
Lancer la lecture de sons dans votre application	203
REMARQUE <i>Volume de l'iPhone</i>	204
Lecture de vidéos	204
Formats de vidéos pris en charge	205
ASTUCE <i>Déterminer le format d'un fichier vidéo</i>	205
Intégrer le lecteur vidéo dans une application	205
S'abonner aux notifications pour suivre le déroulement de la lecture	205
Personnaliser le lecteur vidéo	206
Aller plus loin avec les vidéos iPhone	207
<i>Proposer des vidéos live</i>	207
<i>Ajouter des éléments par-dessus la vidéo</i>	207
Accéder à la bibliothèque musicale de l'iPhone	207
ATTENTION <i>Seuls les contenus audio sont accessibles</i>	208
ATTENTION <i>La bibliothèque iPod n'est pas accessible dans le simulateur</i>	208
Parcourir la bibliothèque musicale de l'iPhone	208
<i>Demander à l'utilisateur de choisir de la musique</i>	208
POUR APPROFONDIR <i>Les propriétés des médias</i>	209
<i>Interroger directement la bibliothèque iPod</i>	210
Contrôler l'iPod depuis l'application	210
REMARQUE <i>La bibliothèque iPod est étanche</i>	211
Tirer partie des photos et vidéos de l'utilisateur	211
Vérifier ce que peut permettre le matériel	212
Paramétrer l'interface de prise de vue	213
Récupérer le média de l'utilisateur	213
ATTENTION <i>La résolution des photos est bien supérieure à l'écran</i>	214
Conclusion	214

CHAPITRE 15

Utiliser les API de notifications 215

Principe de fonctionnement d'APNS	215
REMARQUE <i>Notification et iPod Touch</i>	216
Qu'est-ce qu'une notification ?	216
Pré-requis pour l'utilisation du service de notification	216
ASTUCE <i>Pour vérifier que tout est prêt.</i>	216
Les notifications en quatre étapes	217
Étape 1 : inscription au service de notification	217
ATTENTION <i>Pas de notification dans le simulateur.</i>	217
BEST PRACTICE <i>Répéter l'inscription à chaque lancement de l'application.</i>	218
Étape 2 : Transmettre le jeton APNS à votre serveur	219
Étape 3 : Envoyer les notifications à votre application	220
Obtenir un certificat SSL pour le service APNS	220
BEST PRACTICE <i>Prendre grand soin de ses certificats.</i>	222
Envoyer une notification depuis le serveur	223
<i>Préparer le message de notification</i>	223
<i>Envoi du message</i>	223
CONSEIL <i>On ne réinvente pas la roue !.</i>	224
Étape 4 : Recevoir les notifications	224
Réception des notifications quand l'application est fermée	224
Réception des notifications lorsque l'application est ouverte	225
Détecter les désinscriptions et les erreurs	225
Conclusion	226

CINQUIÈME PARTIE

La publication des applications 227

CHAPITRE 16

Publier sur l'App Store 229

Préparer les éléments marketing en vue de la publication	229
Nom de société et langue principale	231
Le nom de l'application	231
RAPPEL <i>Choisir le nom qui apparaîtra dans le menu de l'iPhone</i>	231
Description de l'application	231
Terminaux ciblés	232
SKU : référence de l'application	232
Catégorie de l'application	232
Numéro de version de l'application	233
Détenteur des copyrights	233

Mots-clés	233
Informations de contact	233
Informations de démonstration (Demo account)	233
Contrat de licence	233
Niveau de contrôle parental de l'application (Ratings)	234
CONSEIL <i>Soyez plus que rigoureux en répondant à ces questions</i>	234
Pays de distribution	234
CONSEIL <i>Visez large.</i>	234
Éléments graphiques	234
<i> Icône de l'application</i>	234
<i> Captures d'écran de l'application.</i>	235
ASTUCE <i>Prendre des captures d'écran avec l'iPhone</i>	235
RAPPEL <i>La propriété UIPrerenderedIcon.</i>	235
Date de disponibilité de l'application	235
CONSEIL <i>La modification de cette date est possible à tout moment.</i>	235
Prix de l'application	236
RAPPEL <i>Anticiper la mise en place du contrat de vente Apple.</i>	236
Localisation de votre application	236
REMARQUE <i>Localisation de l'application</i>	236
Éléments techniques et dernières vérifications	237
Fournir l'application à Apple	237
Dernières vérifications techniques	237
<i> Mode de compilation et niveau de log.</i>	237
<i> Vérifier le contenu du package applicatif.</i>	237
<i> Respect de la charte graphique Apple.</i>	238
<i> Messages d'erreurs réseau</i>	238
ASTUCE <i>Tests de l'application sans réseau</i>	238
<i> Conserver le fichier de symbole</i>	238
Après la soumission	239
REMARQUE <i>Délais de validation</i>	239
Modification des éléments marketing	239
Modification de l'application	239
En cas de rejet	239
Votre application est publiée	240
Suivre les progrès de votre application	240
<i> Statistiques de téléchargement.</i>	240
<i> Les commentaires</i>	240
<i> Les rapports de crash.</i>	240
Quelques conseils de lancement	241
<i> Utilisez vos canaux existants pour communiquer sur l'application</i>	241
CONSEIL <i>Utilisez les éléments marketing Apple.</i>	241

<i>Communiquez auprès des blogs et des sites spécialisés</i>	242
<i>Utilisez le bouche-à-oreille</i>	242
<i>Utilisez les réseaux sociaux</i>	242
<i>Préparez une vidéo de démonstration</i>	242
<small>ATTENTION</small> <i>La qualité de la vidéo doit être au rendez-vous !</i>	242
<i>N'oubliez pas l'auto-promotion</i>	243
Conclusion	243
Index	245

PREMIÈRE PARTIE

Découverte de l'environnement de développement

Cette première partie constitue une introduction indispensable au développement iPhone. Après un rappel des bases de l'Objective-C, elle donne un aperçu de l'environnement de développement pour créer et tester un premier exemple d'application simple.

Le **chapitre 1** présente les différents programmes développeurs, l'inscription et le téléchargement des outils et documentations. Le développeur y trouvera également des explications pour créer un certificat de développement, indispensable pour tester son application sur un iPhone.

Le **chapitre 2** est une introduction à l'Objective-C. Destiné à des développeurs familiers de la programmation orientée objet, il présente le langage d'une façon très pragmatique visant à vous rendre opérationnel rapidement.

Le **chapitre 3** enfin permet au développeur de faire sa première application et de la tester dans le simulateur et sur son iPhone. Les outils indispensables comme Xcode et Interface Builder sont introduits en suivant quelques exemples très simples.

1

Développer pour iPhone

Le développement d'applications iPhone est à la portée de tous les développeurs. Seuls un Mac, un iPhone et l'inscription au programme développeur Apple sont nécessaires pour développer son application, la tester et la publier.

Ce premier chapitre couvre les pré-requis matériels et les connaissances qui seront utiles au développeur, avant d'accompagner le lecteur dans l'inscription à un des programmes développeurs iPhone et la création d'un certificat pour signer et distribuer des applications. Il est destiné aux développeurs, mais aussi au reste de l'équipe qui y trouvera comment s'inscrire pour accéder à la documentation, comment ajouter un téléphone de test, etc.

Équipement matériel requis

Pour développer une application iPhone, il faut disposer d'un Mac et d'un terminal de test.

Un Mac Intel pour développer

Officiellement, le développement d'applications iPhone avec le SDK Apple ne peut se faire que sur des Mac avec processeur Intel. En pratique, c'est la seule solution pour le développeur qui souhaite publier ses applications sur l'AppStore.

ALTERNATIVE Développer sans Mac ?

Quelques alternatives non officielles existent. Il est possible de développer en utilisant un Mac avec un processeur PowerPC, mais de nombreux dysfonctionnements ont été signalés. L'un d'entre eux est que la signature des applications n'est pas possible, ce qui empêche toute installation sur un iPhone.

Une autre solution est d'utiliser le SDK officieux, qui était disponible avant celui d'Apple et qui peut être installé sur un poste de travail sous Linux. La signature d'application et la distribution sur l'App Store ne sont pas possibles en suivant cette méthode, mais cela peut permettre à des développeurs curieux de commencer à développer sans avoir de Mac.

Enfin, il y a quelques initiatives pour mettre au point un SDK compatible Windows. Aucune n'a dépassé le stade de l'annonce sur un site web au moment de la rédaction de cet ouvrage.

Un iPhone ou un iPod Touch pour tester l'application

Avoir un iPhone ou un iPod Touch à disposition est indispensable. Les règles ergonomiques de la plate-forme et les contraintes liées à la taille de l'écran ne peuvent être comprises sans avoir le terminal entre les mains.

Bien qu'un iPod Touch puisse servir pour tester la plupart des applications, il ne permettra pas de tester votre application dans un contexte EDGE ou 3G (c'est-à-dire avec un débit très différent du Wi-Fi) et vous privera de certaines des applications auxquelles la plupart des utilisateurs sont très habitués, comme le téléphone...

CONSEIL Utilisez intensément votre iPhone

Il est fortement recommandé de posséder un iPhone, de l'utiliser comme téléphone principal et de télécharger fréquemment des applications.

Gardez en permanence un regard curieux et critique sur les nouveautés de l'App Store, c'est votre première source d'inspiration. Il n'est pas envisageable de développer des applications pour iPhone sans être un utilisateur averti.

Compétences techniques utiles au développeur iPhone

Le développeur d'applications iPhone doit maîtriser plusieurs connaissances. La plupart ne sont pas spécifiques au développement d'applications pour mobiles ou à l'environnement Mac et auront pu être apprises au préalable.

La programmation orientée objet, au cœur du SDK iPhone

Une bonne maîtrise de la programmation orientée objet est un pré-requis essentiel. Ce sujet ne sera pas repris dans ce livre.

L'héritage, la composition et les design patterns classiques doivent être maîtrisés, car ils sont utilisés de manière intensive dans tout le SDK iPhone.

 H. Bersini, *La programmation orientée objet*, Eyrolles 2009

L'Objective-C : un langage comme un autre

L'Objective-C est le langage imposé pour le développement d'applications iPhone. Ce langage est une nouveauté pour la plupart des développeurs arrivant sur la plateforme iPhone et sa syntaxe peut sembler peu naturelle au premier contact.

VOUS VENEZ D'AUTRES LANGAGES **Pour les développeurs Java, PHP et C#**

Le développeur Java, C# ou PHP objet devrait retrouver rapidement ses marques. Le prochain chapitre, « L'essentiel d'Objective-C », présente en partant du Java les éléments essentiels d'Objective-C et de l'API standard : manipulation de chaînes, dates, dictionnaires, etc.

Programmation multithread

Toutes les applications iPhone utiliseront plusieurs threads d'exécution. C'est grâce à eux, par exemple, que des contenus pourront être chargés en arrière-plan alors que l'interface reste réactive.

Le fait que plusieurs morceaux de code puissent accéder simultanément à la mémoire peut entraîner de subtils bogues, difficiles à reproduire. Le développeur doit donc bien visualiser l'exécution de l'application et comprendre par quels threads chaque morceau de code pourra être exécuté.

VOUS VENEZ D'AUTRES LANGAGES **Synchronisation de threads**

Les techniques de synchronisation entre threads en Objective-C ne sont pas différentes de celles des autres langages et le développeur ayant déjà une expérience de cette problématique ne sera pas surpris. Pour les autres, le chapitre 2 en présente les notions élémentaires, la documentation Apple reprenant également ce sujet.

Développement d'un « client lourd »

On parle de client lourd par opposition au client léger, qui n'embarque pas la logique métier de l'application. Dans une application web, le navigateur est un client léger qui ne prend en charge que l'interface, et la logique métier est exécutée dans un environnement totalement distinct : sur le serveur.

Une application iPhone est un client lourd qui embarque à la fois la logique d'affichage et la logique métier. La réunion des deux offre au développeur une maîtrise beaucoup plus grande de l'ergonomie, mais qui se paie par une augmentation de la complexité de l'application.

Un développeur qui a déjà rencontré ce type de problématique, en développant des clients lourds avec Java/Swing ou C# par exemple, retrouvera facilement ses marques. Les autres doivent se préparer à un changement important dans la façon de concevoir l'application et les échanges avec l'utilisateur.

L'adhésion au programme développeur d'Apple

L'adhésion au programme développeur Apple est nécessaire pour télécharger le SDK iPhone et l'installer. L'adhésion permet également d'accéder à toute la documentation, des exemples de code, et les vidéos de présentation Apple.

Il existe plusieurs modes d'adhésion en fonction du besoin :

- iPhone Registered Developer (gratuit) ;
- iPhone Standard Program à titre individuel ;
- iPhone Standard Program au titre d'une entreprise ;
- iPhone Enterprise Program ;
- iPhone University Program.

Développeur iPhone enregistré : un accès gratuit à l'environnement de développement et à la documentation

C'est le mode d'adhésion le plus simple et la première étape des autres programmes.

Devenir un développeur enregistré Apple est gratuit et vous permettra déjà de développer vos applications et de les tester, mais uniquement dans le simulateur.

Pour vous enregistrer, il suffit de vous rendre sur le site <http://developer.apple.com/iphone/sdk1/> et de suivre le lien pour s'enregistrer. On vous demandera alors d'indiquer votre identifiant Apple (votre compte MobileMe ou le compte utilisé pour acheter sur l'iTunes Store par exemple) ou d'en créer un et de répondre à quelques questions sur vos expériences précédentes de développement.

Le programme Standard pour tester et publier vos applications

L'adhésion au programme standard (iPhone Standard Program) vous permettra de tester vos applications sur iPhone et de les publier sur l'App Store. Elle est payante (99 \$ ou 79 €).

CONSEIL Développer pour un tiers

Si vous souhaitez développer des applications pour le compte d'une autre société, vous devez demander à votre client d'ouvrir son propre compte sur le programme développeur iPhone et de vous ajouter comme développeur.

C'est le seul moyen pour que l'application apparaisse avec le nom de votre client comme éditeur.

Les deux modes d'adhésion au programme Standard

L'adhésion à ce programme peut se faire à titre individuel ou au nom d'une société.

Dans le premier cas, un seul développeur pourra utiliser ce compte pour créer des applications, les signer et les installer sur des iPhone.

Dans le deuxième cas, vous pourrez enregistrer plusieurs développeurs associés à ce compte (on ne paie qu'une fois pour toute l'équipe) et distribuer les droits aux membres de l'équipe. C'est le mode recommandé pour toute équipe de développement.

Le processus d'adhésion au programme Standard

Pour adhérer, il faut se rendre sur le site du programme développeur Apple

► <http://developer.apple.com/iphone/program/apply.html>

et suivre le lien *Apply Now* :

ATTENTION Ne pas confondre le programme Standard et le programme Entreprise

L'inscription au programme Standard au nom de votre entreprise (deuxième mode d'adhésion décrit ci-dessus) se fait en suivant le lien *Standard Program*. Le choix entre l'inscription à titre individuel ou au nom d'une entreprise se fait plus tard dans le processus d'inscription.

Ne confondez pas avec le programme *Enterprise Program* (299 \$) qui permet lui de diffuser des applications en interne au sein d'un grand groupe, sans passer par l'App Store.

Les étapes de l'adhésion pour un développeur individuel sont les suivantes :

- 1 Devenir un développeur iPhone enregistré (cf. paragraphe précédent).
- 2 Demander l'adhésion au programme Standard et répondre aux questions sur le site Apple.
- 3 Attendre la confirmation par courriel d'Apple (quelques jours).

4 Payer en ligne les frais de l'adhésion.

Pour une adhésion au nom d'une entreprise, le processus est un peu plus compliqué :

- 1 Devenir un développeur iPhone enregistré (cf. paragraphe précédent).
- 2 Demander l'adhésion au programme Standard et répondre aux questions sur le site Apple – il faut indiquer le contact juridique de la société.
- 3 Attendre le courriel d'Apple au service juridique : Apple envoie après quelques jours un courriel au contact juridique en lui demandant d'envoyer l'extrait Kbis de l'entreprise par fax.
- 4 Attendre la confirmation par courriel d'Apple (quelques jours).
- 5 Payer en ligne les frais de l'adhésion.

CONSEIL N'hésitez pas à contacter le support Apple aux développeurs

Dans certains cas, des demandes d'adhésion au nom d'une entreprise peuvent mettre longtemps avant d'être traitées, voire même rester sans réponse.

Le support Apple Developer Connection est très efficace et peut aider à connaître l'état d'une demande en cours. Leurs numéros de téléphone sont disponibles sur le site Apple :

► <http://developer.apple.com/contact/phone.html>

Pour la France, le numéro est : +33 (0) 800 90 7226.

Le programme Entreprise pour des applications internes

Le programme Entreprise (*iPhone Enterprise Program*) permet à l'équipe de développement d'une grande entreprise de développer des applications pour une distribution et un usage interne (*In-House Distribution*).

Ce programme ne permet pas de distribuer des applications sur l'App Store.

Le programme universitaire pour l'enseignement

Ce programme gratuit permet à un enseignant de s'inscrire afin que ses étudiants puissent développer, tester sur iPhone et publier sur l'App Store. Il permet également aux étudiants d'échanger leurs applications entre eux.

Il est disponible aux États-Unis depuis 2008 et en France depuis le début de l'année 2009 pour quelques écoles et universités.

Les sites web Apple pour le développeur iPhone

L'adhésion au programme développeur iPhone vous donne accès à plusieurs sites web d'Apple.

Le centre de développement iPhone

Le centre de développement iPhone (*iPhone Dev Center*) regroupe toute la documentation destinée aux développeurs.

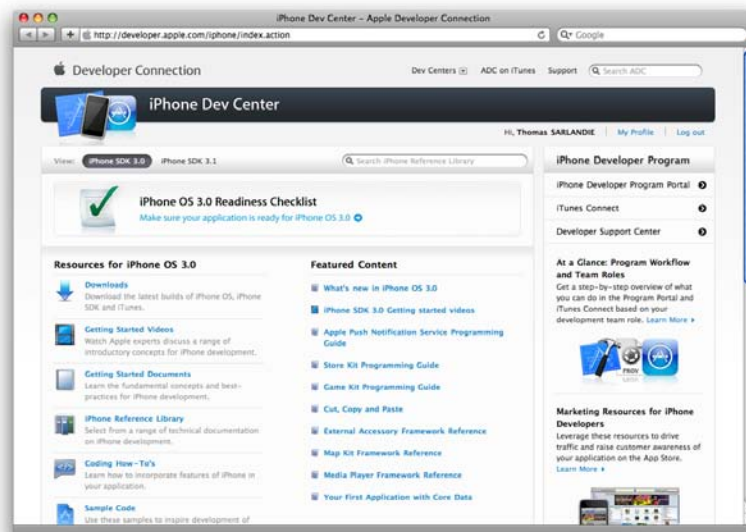
► <http://developer.apple.com/iphone/>

Vous y trouverez également des contenus vidéo, des exemples de code, et des liens pour télécharger la dernière version du SDK.

C'est enfin le point d'accès au portail du programme iPhone.

Figure 1-1

Le centre de développement pour les développeurs d'applications iPhone



Le portail du programme iPhone

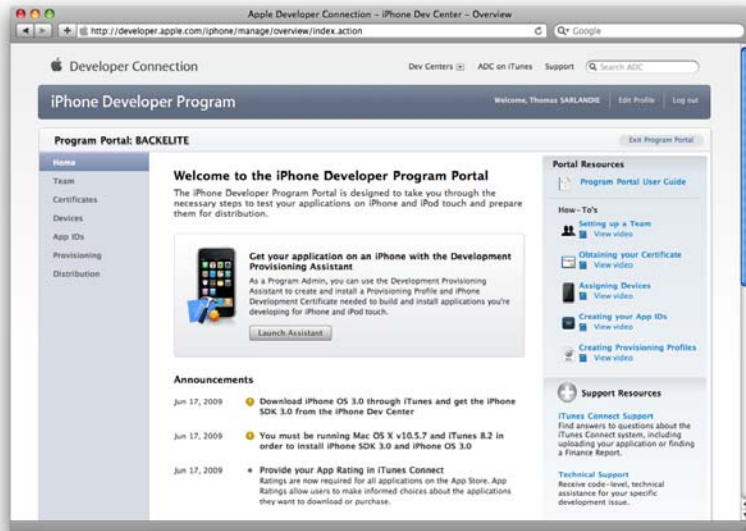
Ce site est accessible depuis le centre de développement (lien *iPhone Developer Program Portal* en haut à droite). C'est un outil web qui régit tous vos échanges avec Apple avant la soumission de l'application.

Il permet ainsi de :

- 1 déclarer les membres de l'équipe de développement ;
- 2 créer des certificats électroniques pour les développeurs ;
- 3 déclarer les iPhone que vous utiliserez pour tester les applications.

Nous reviendrons un peu plus loin sur cet outil indispensable aux développeurs.

Figure 1-2
Le portail du programme
développeur iPhone

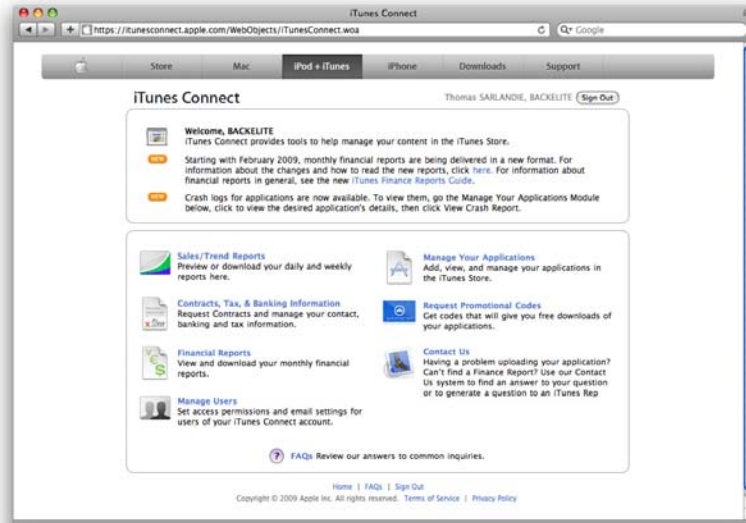


iTunes Connect, pour la publication des applications

iTunes Connect est l'outil utilisé pour publier des contenus vers Apple. Il est utilisé par l'industrie musicale pour publier de la musique sur iTunes, et vous l'utiliserez pour publier vos applications une fois satisfait de votre travail.

C'est également grâce à cet outil que vous suivrez les téléchargements de l'application, et dans le cas d'une application payante, vos revenus.

Figure 1-3
Le portail iTunes Connect pour distribuer vos applications sur l'App Store



Présentation du SDK iPhone

Le SDK est un package d'installation (fichier `.dmg`) téléchargeable une fois connecté au portail développeur (*iPhone Dev Center*) à l'adresse <http://developer.apple.com/iphone/index.action>.

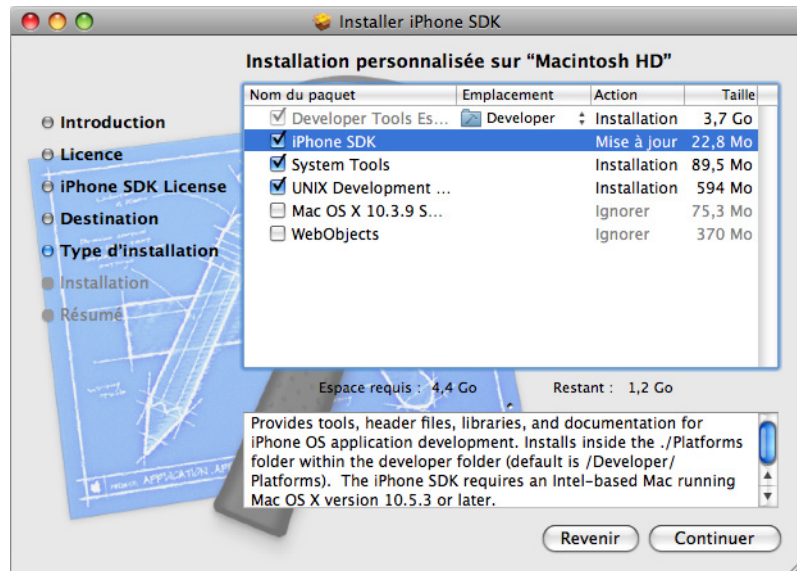
L'installation du SDK se fait de manière classique : en double-cliquant sur le package pour le monter, le programme d'installation se lance automatiquement.

Tableau 1-1 Principaux composants installés avec le SDK

Nom du composant	Description
Xcode	L'outil de développement Apple, il permet la création de projets iPhone, l'édition du code source Objective-C, la compilation et le débogage des applications.
SDK Mac OS X 10.5	L'ensemble du SDK standard Mac fait partie des pré-requis de l'installation.
Interface Builder	C'est un outil visuel pour construire des interfaces graphiques.
Organizer	Cet outil vous permet de gérer les iPhone que vous utilisez pour développer et d'y installer des applications sans passer par iTunes.
iPhone Simulator	C'est un simulateur permettant de tester les applications directement sur l'ordinateur.

Tableau 1-1 Principaux composants installés avec le SDK (suite)

Nom du composant	Description
Instruments	Cet outil permet d'analyser un programme pour surveiller l'état de la mémoire, l'utilisation du réseau, du CPU, etc.
Shark	Il permet d'optimiser l'application en identifiant les fonctions dans lesquelles elle passe le plus de temps.

Figure 1-4
Installation du SDK iPhone**REMARQUE Installation du SDK iPhone sous Snow Leopard**

Pour installer le SDK sous Snow Leopard, Xcode et le SDK iPhone devaient être installés séparément. Il fallait commencer par installer les outils de développement depuis le CD de Snow Leopard, avant d'installer la version spécifique du SDK iPhone pour Snow Leopard. Depuis la version 3.1 du SDK, Xcode est de nouveau intégré.

La documentation Apple, une aide à ne pas négliger

La documentation fournie par Apple est très riche, et nous vous recommandons de vous y référer. En voici un sommaire rapide permettant de retrouver l'information pertinente.

Les guides pour le développeur

Les guides font un tour d'horizon complet sur un sujet. Ils sont tous accessibles depuis le portail des développeurs (*iPhone Dev Center*), en suivant le lien *iPhone Reference Library*.

Le guide consacré aux règles à respecter en matière d'ergonomie

Le guide *iPhone Human Interface Guidelines* décrit les principes ergonomiques qui font de l'iPhone une plate-forme uniforme dans laquelle les utilisateurs retrouvent facilement leur chemin.

La description des API et de la bibliothèque graphique

Le *iPhone Application Programming Guide* présente les API les plus importantes de l'iPhone, les limitations imposées aux applications, et le fonctionnement de la bibliothèque graphique UIKit.

La référence Objective-C

Le *Objective-C 2.0 Programming Language* décrit le langage Objective-C et les nouveautés de sa version 2.0.

Les exemples de code : des projets Apple exemplaires

De nombreux exemples de projets sont fournis par Apple, chacun montrant comment utiliser une des API ou comment répondre à un problème classique.

Les exemples peuvent être téléchargés un par un en suivant le lien *Sample Code* depuis l'*iPhone Dev Center*.

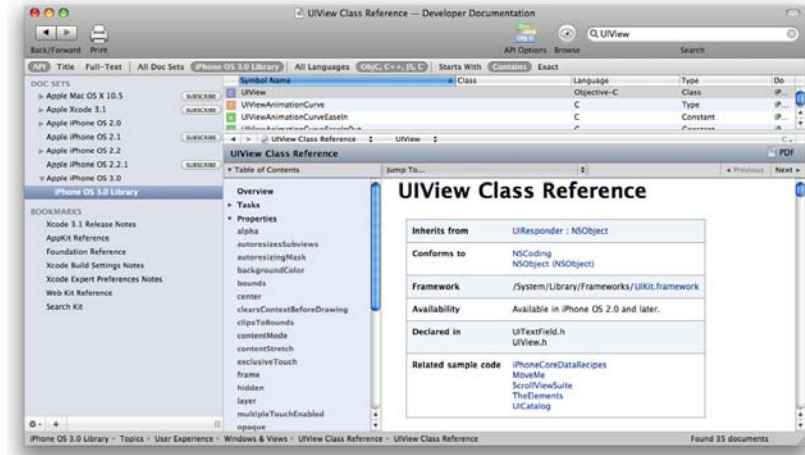
La documentation de référence exhaustive

La documentation la plus complète et la plus exhaustive couvre l'ensemble des API publiques de l'iPhone. Elle peut être consultée en ligne ou bien téléchargée pour être lue directement depuis Xcode :

- 1 Lancer Xcode.
- 2 Dans le menu *Help* sélectionner *Documentation*.

Le champ de recherche en haut à droite permet de trouver très rapidement la documentation d'une classe, d'une méthode ou d'une constante.

Figure 1–5
La fenêtre Documentation
de Xcode



ASTUCE Avoir toujours la documentation à jour

Pour télécharger l'ensemble de la documentation sur votre ordinateur et l'avoir toujours à portée de main, cliquez sur le bouton *Subscribe* à côté de l'ensemble de documents *iPhone OS Library* dans la fenêtre *Documentation* de Xcode. Tous les guides, les documents de référence et les exemples seront téléchargés et mis à jour automatiquement si nécessaire.

Pré-requis pour la distribution d'une application

Une des révolutions apportées par iPhone et le SDK est le modèle de distribution intégré au terminal qui contribue fortement au succès du téléphone et de ses applications.

Il existe deux modes de distribution des applications : le mode Ad Hoc et le mode App Store. Quel que soit le mode retenu, la sécurité des applications est assurée par une signature électronique.

RAPPEL Signature électronique et certificat

La signature électronique d'un fichier permet d'en garantir l'origine et de s'assurer qu'il n'a pas été modifié pendant le transfert. Le certificat est la contre-signature de ce fichier par une autorité tierce de certification. Dans le cas des applications iPhone, le certificat est émis par Apple et permet à l'utilisateur de s'assurer que l'application provient bien du développeur et qu'elle n'a pas été modifiée (par un virus par exemple) entre-temps.

Préparation à la diffusion d'une application en test (mode Ad Hoc)

Le mode de distribution Ad Hoc permet de diffuser une application à un ensemble fini d'utilisateurs. Le développeur doit donner une liste d'iPhone et d'iPod Touch qui seront explicitement autorisés à lancer l'application : elle ne pourra pas être installée sur d'autres téléphones. Cette liste ne peut pas contenir plus de 100 identifiants de terminaux.

ATTENTION Les places d'identifiants sont précieuses

Chaque terminal ajouté à la liste occupe une place dans les 100 identifiants mis à votre disposition. Même en supprimant des identifiants de la liste, vous ne récupérez pas immédiatement ces places. Une fois par an, à la date anniversaire de votre compte, le compteur est réinitialisé et vous avez de nouveau droit à 100 terminaux. Si au moment du renouvellement, il y a 25 téléphones déjà présents dans la liste, vous pourrez ajouter 75 nouveaux identifiants.

Les applications distribuées ainsi ne sont pas soumises à la validation d'Apple. Ce mode est indispensable pour tester votre application sur un terminal réel et sera aussi intéressant pour diffuser une application auprès de bêta-testeurs ou à un petit cercle d'utilisateurs.

Pour pouvoir distribuer des applications en mode Ad Hoc, le développeur doit déjà être inscrit au programme développeur standard. Le processus est décrit ici étape par étape.

Une fois l'environnement configuré pour la publication en mode Ad Hoc, l'apprentissage du développement iPhone peut réellement commencer.

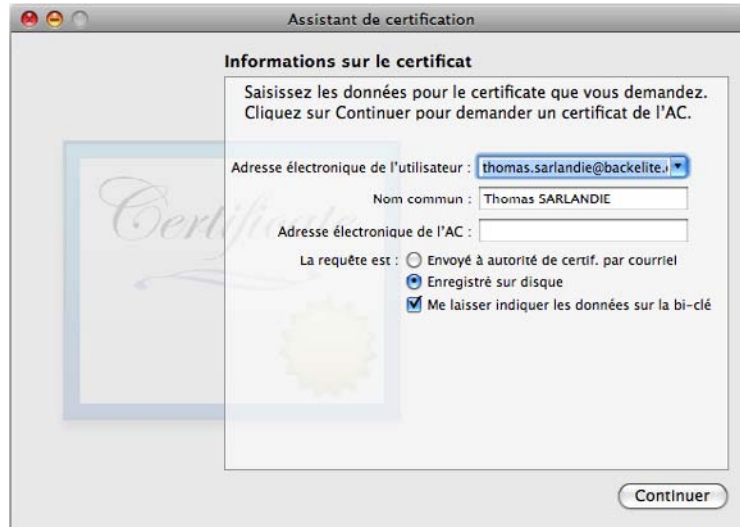
Générer un certificat de développeur

La première étape du processus consiste à générer une demande de certificat. Ce certificat sera transmis à Apple et une fois validé, permettra au développeur de signer électroniquement les applications.

Pour créer la demande de certificat, lancer l'application *Trousseau d'accès* dans *Applications > Utilitaires*.

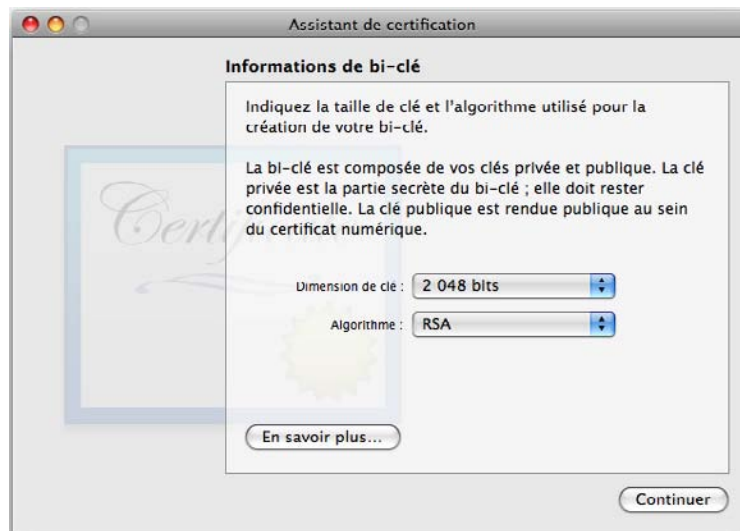
Dans le menu *Trousseau d'accès*, sélectionner l'option *Assistant de certification - Demander un certificat à une autorité de certificat*. L'assistant de certification se lance.

Figure 1-6
Lancement de l'assistant de certification



Sélectionner l'option *Enregistré sur disque* et *Me laisser indiquer les données de la bi-clé*, valider. Indiquer l'endroit où enregistrer la demande de certificat.

Figure 1-7
Paramètres de la demande de certificat

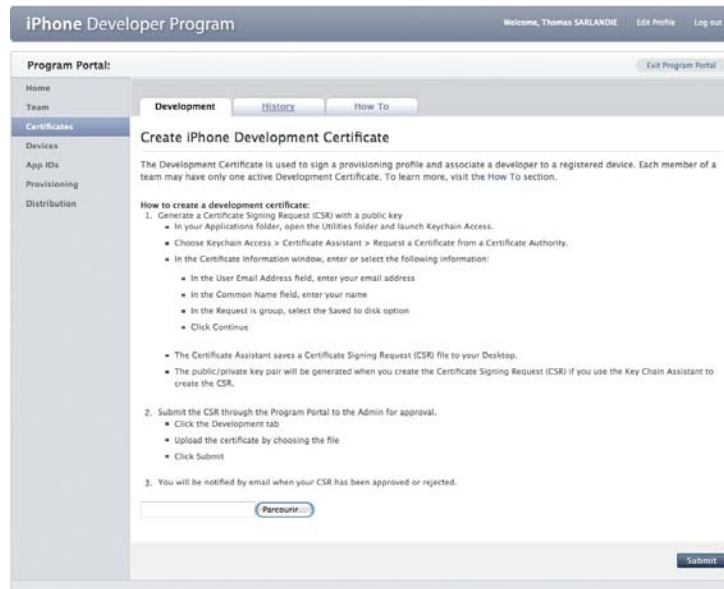


Vérifier que la bi-clé créée est bien basée sur l'algorithme RSA avec 2048 bits car Apple exige ce niveau de sécurité.

Faire signer par Apple la demande de certificat

Comme nous l'avons vu, les certificats sont gérés via le portail du programme iPhone. Sélectionner l'option *Certificates*, puis le bouton *Add Certificate*. Utiliser le champ de téléchargement de fichier en bas de la page pour envoyer le certificat.

Figure 1–8
Envoi du fichier
de demande de certificat



Une fois la demande transmise, elle est visible dans la page *Certificats* de l'App Store, et doit être approuvée par le responsable de l'équipe de développement. Pour cela, il suffit de cliquer sur le bouton *Approuve*.

Figure 1–9
Validation d'une demande
de certificat

Name	Expiration Date	Provisioning Profiles	Status	Actions
Thomas SARLANDIE			Pending Approval	Approve Reject

Une fois la demande validée, un traitement a lieu pour générer le certificat. Après quelques minutes, le certificat validé est disponible. Il suffit alors de le télécharger et de double-cliquer dessus pour l'installer sur le poste du développeur.

Figure 1–10
Téléchargement du certificat
validé

Name	Expiration Date	Provisioning Profiles	Status	Actions
Thomas SARLANDIE	12/09/2009 02:45 PM		Issued	Download Revoke

ATTENTION Le certificat développeur est lié à la machine du développeur

Le certificat téléchargé depuis l'outil Program Portal ne contient pas la clé privée et ne suffit pas pour utiliser cette clé développeur sur un autre ordinateur.

Pour pouvoir exporter la clé privée et la réutiliser sur un autre ordinateur, référez-vous à la documentation disponible dans l'onglet *HowTo* dans le Program Portal.

Pour développer à plusieurs, chaque développeur doit avoir sa clé. C'est possible si vous avez créé un compte développeur au nom d'une entreprise.

Créer un identifiant d'application

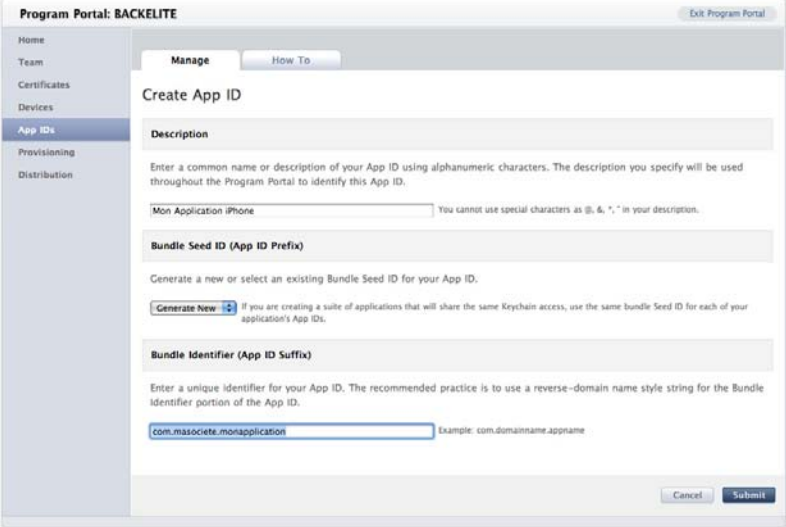
Avant de pouvoir distribuer une application, vous devez générer un identifiant d'application (AppID) et l'associer à votre certificat.

L'identifiant déclaré dans le portail du programme iPhone devra correspondre à l'identifiant d'application (que vous déclarerez dans le fichier `Info.plist` de votre application).

Un préfixe est généré aléatoirement par Apple, ce qui permet de garantir le caractère unique de chaque identifiant d'application. Si vous le souhaitez, vous pouvez tout de même ajouter votre propre préfixe, comme le nom de domaine de la société.

Ainsi, pour une société qui développerait deux applications, RSSReader et BlogReader, vous devriez créer deux identifiants d'application différents : `com.acme2_0.rssreader` et `com.acme2_0.blogreader`. Le préfixe unique aléatoire est ajouté automatiquement et le développeur n'a pas besoin de s'en préoccuper.

Figure 1–11
Création d'un identifiant
d'application



The screenshot shows the 'Program Portal: BACKELITE' interface. On the left is a navigation menu with options: Home, Team, Certificates, Devices, App IDs (selected), Provisioning, and Distribution. The main content area is titled 'Create App ID' and contains the following sections:

- Description:** A text input field with the value 'Mon Application iPhone'. A note below states: 'You cannot use special characters as @, &, *, " in your description.'
- Bundle Seed ID (App ID Prefix):** A section with a 'Generate New' button and a note: 'If you are creating a suite of applications that will share the same Keychain access, use the same bundle Seed ID for each of your application's App IDs.'
- Bundle Identifier (App ID Suffix):** A text input field with the value 'com.masociete.monapplication'. A note below states: 'The recommended practice is to use a reverse-domain name style string for the Bundle Identifier portion of the App ID.' An example 'com.domainname.appname' is shown to the right.

At the bottom right, there are 'Cancel' and 'Submit' buttons.

Dans la section *App IDs* du Program Portal, cliquer sur le bouton *New App ID* et entrer le nom de l'application et son identifiant.

Définir les iPhone autorisés

La définition de la liste des iPhone autorisés se base sur les identifiants uniques de téléphones (UDID ou *Unique Device Identifier*).

ASTUCE Obtenir l'UDID d'un iPhone

L'UDID est un identifiant propre à chaque téléphone. Il peut être obtenu dans *Organizer* ou bien dans iTunes (sur Windows ou sur Mac).

Pour voir l'UDID d'un téléphone dans iTunes, il faut cliquer sur le libellé *Numéro de série* (attention, il faut bien cliquer sur le libellé ; pas sur le numéro de série lui-même), il est alors remplacé par l'UDID (« *Identifiant* »). En cliquant dessus, et en appuyant sur *Cmd-C* on le copie dans le Presse-papiers.



Figure 1-12 Cliquer sur le libellé *Numéro de série* dans iTunes pour faire apparaître l'identifiant du téléphone (UDID).

Avec le programme standard, il est possible d'autoriser jusqu'à 100 terminaux. Au-delà, il faudra envisager une distribution avec le programme Entreprise (qui permet des distributions en interne à grande échelle) ou par l'App Store.

L'ajout des identifiants se fait dans le Program Portal qui est accessible depuis l'iPhone Dev Center. Dans l'onglet *Devices*, vous pouvez ajouter des téléphones à votre compte.

Figure 1-13
Gestion des terminaux associés à votre compte développeur



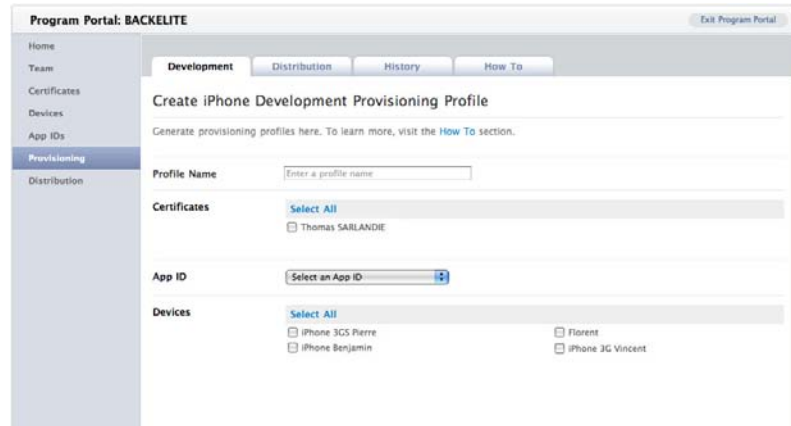
Générer le profil de provisionnement

Dernière étape, le profil de provisionnement est un fichier qui lie un ou plusieurs certificats avec un AppID et une liste de téléphones.

Il autorise les applications signées par un des développeurs, et dont l'identifiant correspond au AppID, à être installées sur un des téléphones de la liste.

Par ce procédé, il est possible de créer plusieurs groupes de testeurs et de définir précisément qui pourra tester les applications.

Figure 1-14
Création d'un profil de provisionnement



Dans le *Program Portal*, sélectionner l'onglet *Provisioning* pour créer et télécharger les profils de provisionnement.

Installer le profil sur des iPhone

Le profil s'installe en faisant glisser le fichier téléchargé (extension `.mobileprovision`) sur iTunes ou Xcode. Cette opération est possible sous Mac et sous Windows.

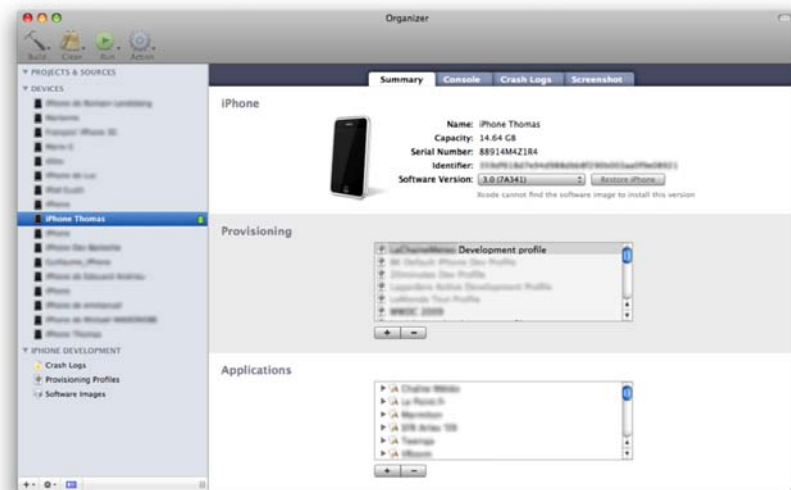
Le menu *Général > Profils* de l'application *Réglages* sur l'iPhone permet de vérifier que le profil est bien installé.

La fenêtre Organizer de Xcode (menu *Window > Organizer*) permet de lister tous les profils installés sur l'ordinateur et de voir ceux installés sur les iPhone connectés.

Figure 1–15
Vérification des profils installés dans l'iPhone



Figure 1–16
Vérification des profils installés via Organizer



Installer l'application sur des iPhone

Pour le développeur, l'installation des applications sur iPhone peut se faire très simplement depuis Xcode, ce qui sera détaillé au chapitre 3 « Premiers pas avec le SDK iPhone ».

Pour les autres (client, testeur, ami, etc.) il est possible d'envoyer l'application par courriel.

Le développeur devra tout d'abord compresser l'application (le répertoire dont le nom se termine en `.app`) et l'envoyer avec le fichier de provisionnement.

Pour le destinataire, il faut tout d'abord installer le profil de provisionnement en le faisant glisser sur l'icône iTunes (sous Mac ou sous Windows). Il faut ensuite décompresser l'application et la faire glisser sur iTunes (sous Mac ou sous Windows). L'application sera automatiquement installée lors de la prochaine synchronisation de l'iPhone.

CONSEIL Vérifications à faire en cas de problème

Si le destinataire n'arrive pas à installer l'application, vérifiez les points suivants :

1. L'UDID de l'iPhone ou de l'iPod Touch a bien été ajouté au compte développeur, et il n'y a pas eu d'erreur de saisie.
2. Le terminal a été ajouté au fichier de provisionnement.
3. Le fichier de provisionnement a été téléchargé à nouveau et réinstallé par le développeur avant de recompiler l'application. Éventuellement, il peut être utile d'effacer tous les fichiers de provisionnement d'iTunes (`~/Library/MobileDevice/Provisioning Profiles/`) puis réajouter le fichier dans iTunes.
4. Le fichier de provisionnement a été correctement installé sur le terminal cible (vous pouvez le vérifier dans le menu *Général - Profils* de l'application *Réglages*).
5. L'identifiant de l'application saisi dans Xcode correspond bien à l'identifiant d'application fourni sur le site du programme développeur.

Mode de distribution via l'App Store pour une diffusion large

La distribution App Store permet de proposer votre application à l'ensemble des utilisateurs d'iPhone dans le monde. Elle peut se faire gratuitement ou contre rémunération. Dans ce deuxième cas, Apple reverse au développeur environ 70 % des revenus.

Dans tous les cas, avant d'être disponible sur l'App Store, l'application sera vérifiée par Apple qui s'assurera que :

- l'application respecte les principes ergonomiques iPhone ;
- l'application ne plante pas (les tests Apple ne peuvent bien sûr pas être exhaustifs et c'est avant tout à l'équipe de développement de s'assurer du bon fonctionnement de l'application) ;
- du respect de l'« accord » iPhone SDK (qui précise par exemple que certains types d'applications ne sont pas autorisés, que l'utilisation excessive du réseau est interdite, que les API privées ne doivent pas être utilisées, etc.).

Cette vérification peut prendre de 1 à 6 semaines et il est donc essentiel de s'assurer que l'application respecte parfaitement toutes les règles du SDK iPhone avant de la publier.

Publication sur l'AppStore

La publication sur l'AppStore suit les mêmes étapes que la publication en mode Ad Hoc, avec quelques spécificités :

- 1 Génération d'un certificat spécifique.
- 2 Réutilisation du AppID.
- 3 Création d'un nouveau fichier de provisionnement pour la distribution. Cette fois-ci, vous n'indiquerez pas de terminaux autorisés à lancer l'application. Tous les terminaux le seront, une fois l'application publiée.
- 4 Compilation et signature avec le nouveau certificat.
- 5 Envoi de l'application à Apple via iTunes Connect.

Le dernier chapitre de ce livre « Publier sur l'App Store » est consacré à la publication d'application. Il décrit tous les autres éléments à fournir et donne des conseils pour réussir la publication et le lancement de l'application.

Conclusion

Dans ce premier chapitre, vous avez appris comment rejoindre le programme développeur Apple pour pouvoir télécharger le SDK, l'installer et accéder à la documentation. Nous avons également mis en place les pré-requis pour tester votre application sur un iPhone réel, la partager avec d'autres utilisateurs et même la publier sur l'App Store.

Il est temps de (re)découvrir le langage Objective-C dans le prochain chapitre.

2

L'essentiel d'Objective-C

L'Objective-C est le langage de programmation du SDK iPhone. Sa syntaxe peut sembler déroutante au premier abord, mais les développeurs habitués aux langages de programmation orientés objet s'y habitueront très vite et ne tarderont pas à en découvrir les avantages.

Ce chapitre est une introduction rapide à l'Objective-C, mais amplement suffisante pour développer vos premières applications iPhone.

APPROFONDIR Guide de programmation Objective-C

Apple fournit gratuitement en ligne un guide appelé *The Objective-C 2.0 Programming Language* et qui est la référence du langage utilisé pour développer des applications iPhone.

Ce guide explique en détail toutes les possibilités du langage et fournit de nombreuses explications et conseils. Sa lecture est très fortement recommandée.

Les origines

L'Objective-C a été inventé au début des années 1980 par Brad Cox, créateur de la société Stepstone. Son objectif était de combiner la richesse du langage Smalltalk (un des premiers langages orientés objets) et la rapidité du C.

Ses travaux ont abouti en 1986 par la publication du livre *Object-Oriented Programming, An Evolutionary Approach*, le premier livre à décrire complètement le langage puis à la mise sur le marché d'un compilateur Objective-C et d'un ensemble de bibliothèques.

Steve Jobs, ayant été prié de quitter Apple, fonde en 1985 la société NeXT éditrice de logiciels et fabricant de matériels. En 1988, NeXT achète à Stepstone le droit de réaliser son propre compilateur Objective-C et ses bibliothèques.

Le système d'exploitation des ordinateurs NeXT (NextStep) est alors développé en Objective-C, puis NeXT se concentre sur les logiciels et vend NextStep comme une plate-forme de développement.

En juin 1996, Apple rachète NeXT Software et récupère toute la technologie NextStep. Elle revoit le jour quelques années plus tard dans Mac OS X sous le nom Cocoa et enfin dans l'iPhone sous le nom Cocoa Touch.

L'Objective-C 2.0 est une nouvelle version du langage introduite avec Mac OS X 10.5.

Principes fondamentaux

Comme tout langage, l'Objective-C pose quelques principes fondamentaux qui structurent tout le langage et ses bibliothèques de base.

Tout est objet

En Objective-C, tous les objets héritent de la classe `NSObject` (on retrouve ce principe en Java où tous les objets héritent de `java.lang.Object`).

Toutes les méthodes de l'objet `NSObject` sont donc accessibles dans tous les objets que vous créerez et manipulerez.

Cela signifie aussi qu'une méthode ou une classe capable de manipuler des objets de type `NSObject` est capable de manipuler tous les objets qui peuvent être instanciés. Cette particularité très importante est abondamment utilisée dans la bibliothèque standard pour fournir des outils comme les listes et les dictionnaires qui peuvent être exploités avec tout type d'objet.

Envoi de messages

Les objets Objective-C communiquent entre eux grâce à des envois de messages qui sont gérés par le runtime.

Cette notion est équivalente à un appel de méthode dans d'autres langages orientés objets, mais le compilateur n'a pas besoin de vérifier à la compilation que l'objet cible est capable de traiter ce message (dans la plupart des cas, le compilateur pourra néanmoins signaler les cas qui lui semblent incorrects).

À l'exécution, le message est transmis à l'objet destinataire et si celui-ci ne sait pas y répondre, une exception est levée.

La syntaxe Objective-C

L'Objective-C est une extension du C. Toute la syntaxe C est reconnue et peut être utilisée. Le C ayant servi de base aux langages les plus répandus et les plus enseignés aujourd'hui, nous ne redétaillerons pas sa syntaxe.

Toutes les extensions ajoutées par l'Objective-C au langage C visent à permettre la programmation orientée objet. Un des éléments les plus importants est l'envoi de messages – c'est aussi cela qui rend le langage très surprenant au premier abord.

Envoyer des messages à un objet

Les envois de message se font en utilisant la syntaxe suivante :

```
[destinataire message];
```

Le premier élément entre crochets est l'objet qui recevra le message, et le deuxième élément est le nom du message à envoyer.

```
// Equivalent en Java :  
destinataire.message();
```

Il est bien sûr possible d'ajouter des paramètres. En Objective-C, les paramètres des méthodes sont nommés :

```
[destinataire message:argument1 etArgument2:argument2];  
[string stringByReplacingOccurrencesOfString:@"iPhone"  
withString:@"iPod"];
```

Ici, on remarque que le nom de la méthode contient aussi le nom du premier argument. Ainsi, `stringByReplacingOccurrencesOfString` est le nom de la méthode et il contient le nom du premier argument. `withString` est le nom du deuxième argument.

```
// Equivalent en Java :  
string.replaceAll("iPhone", "iPod");
```

Le nommage des paramètres est une des spécificités de l'Objective-C. À l'usage, cela permet de gagner énormément de temps puisqu'il n'est plus nécessaire d'ouvrir la documentation pour voir quels sont les paramètres à passer, dans quel ordre, etc.

ATTENTION Signature de méthode en Objective-C

Bien que les arguments soient nommés, ils ne sont pas interchangeables. Leur ordre fait partie de la signature de la méthode.

Ainsi, dans l'exemple ci-dessus, les signatures des méthodes sont : `message`, `message:`, et `Argument2:`, ou encore : `stringByReplacingOccurrencesOfString:withString:`.

Il est également possible de récupérer une valeur de retour :

```
NSObject *returnedObject = [destinataire message:argument1];
```

Et de chaîner plusieurs appels de méthodes :

```
NSObject *returnedObject =  
[[destinataire message] faisQue quelquechoseAvec];
```

Le type `id` et la valeur `nil`

Le type `id` est un pointeur vers n'importe quel objet du langage.

```
id anObject;  
- (id) uneMethodeQuiPeutRenvoyerNImporteQue1Objet;
```

On utilise le type `id` quand on veut renvoyer un objet dont on ne connaît pas encore le type ou quand une méthode peut prendre n'importe quel type d'objet en argument.

REMARQUE `id` est un pointeur

La notation `id` est un pointeur vers un objet, il ne faut donc pas ajouter d'étoile.

Le compilateur n'affiche pas de message d'avertissement lorsqu'un type `id` est converti en un type plus précis par le développeur :

```
NSString *maChaine = [self uneMethodeQuiPeutRenvoyerNImporteQue1Objet];
```

La valeur `nil` est définie comme l'objet `nu1`. C'est l'équivalent de `NULL` en C ou de `null` en Java. Il est possible d'envoyer un message à `nil`, dans ce cas, la valeur de retour est toujours `nil` ou `0`.

Déclaration d'une classe

Pour chaque classe on définit une interface et une implémentation. L'interface décrit le comportement de la classe tel qu'il sera vu par les autres classes, l'implémentation décrit la logique interne de la classe.

Par convention, on sépare la déclaration et l'implémentation dans deux fichiers différents : un fichier `.h` pour la déclaration et un fichier `.m` pour l'implémentation.

Déclaration d'une classe : `MaClasse.h`

```
@interface MaClasse : NSObject {
    // Liste des variables d'instance de la classe
    NSString *monNom;
    NSInteger *monAge;
}

// Liste des méthodes de la classe
- (void) uneMethodeDInstance;
+ (void) uneMethodeDeClasse;

@end
```

On remarque que la déclaration commence au mot-clé `@interface` et se termine avec le mot-clé `@end`. Elle contient un bloc limité par des accolades qui contient la liste des variables d'instance, puis entre la fin des accolades et le mot-clé `@end`, on trouve la liste des méthodes.

Implémentation d'une méthode

```
#import "MaClasse.h"

@implementation MaClasse

- (void) uneMethodeDInstance
```



```
{  
    // ...  
}  
  
+ (void) uneMethodeDeClasse  
{  
    // ...  
}  
  
@end
```

L'implémentation commence par le mot-clé `@implementation` et se termine par le mot-clé `@end`.

Les variables d'instance

Elles sont déclarées dans l'interface. Par défaut, elles ne sont pas visibles par les autres objets. Nous verrons comment les exposer.

ATTENTION Ne pas confondre variable d'instance et propriété

Les variables d'instance sont l'équivalent des propriétés en Java ou en PHP, mais en Objective-C une propriété est une notion différente et bien spécifique qui sera décrite un peu plus loin.

Les différents types de méthodes

Les méthodes d'une classe peuvent être des méthodes d'instance ou des méthodes de classe. Les méthodes d'instance ne peuvent être appelées que sur une instance de la classe. Elles ont accès à toutes les variables d'instance de la classe. Par défaut, toutes les méthodes déclarées dans l'interface sont publiques.

Les méthodes de classe sont appelées directement sur la classe, elles n'ont pas accès aux variables d'instance de la classe.

REMARQUE Méthode de classe et méthode statique

Les méthodes de classe sont l'équivalent des méthodes statiques dans d'autres langages.

Les méthodes d'instance sont préfixées par un `-` dans leur déclaration et leur implémentation. Les méthodes de classe sont préfixées par un `+`.

```
// Appel de la méthode de classe  
[MaClasse uneMethodeDeClasse];
```

```
// Appel de la méthode d'instance
MaClasse *instance = [[MaClasse alloc] init];
[instance uneMethodeDInstance];
```

Héritage

En Objective-C, une classe hérite forcément de `NSObject` ou d'un de ses descendants. On indique la classe parente dans la déclaration de la classe.

L'héritage multiple n'est pas possible en Objective-C, une classe hérite toujours d'une seule classe.

```
@interface MaDeuxiemeClasse : MaClasse {
    ...
}
...
```

Faire référence à l'objet courant et à son père

Pour faire appel à une méthode ou une propriété de l'objet courant, on utilise le mot-clé `self` qui est l'équivalent de `this` en Java ou C++.

Pour faire appel à une méthode de l'objet parent, on utilise le mot-clé `super`.

```
- (void) uneMethode
{
    // Appel d'une autre méthode du même objet
    [self uneAutreMethode];

    // Appel d'une méthode de l'objet parent
    [super uneMethode];
}
```

Initialisation d'une instance d'objet

Pour obtenir l'instance d'un objet, il faut allouer la mémoire pour l'objet puis l'initialiser à l'aide d'un initialiseur.

On peut comparer l'initialisateur à un constructeur (cette notion existe en Java, C++ et PHP) qui ne prendrait pas en charge l'allocation de la mémoire.

L'allocation d'un nouvel objet se fait à l'aide de la méthode de classe `alloc`.

```
MaClasse *instanceNonInitialisee = [MaClasse alloc];
```

L'allocation est l'équivalent d'un `malloc()`, il faut *toujours* initialiser la structure de données en appelant un initialisateur avant d'utiliser l'objet.

Chaque classe définit ses initialisateurs. Par défaut, on appelle la méthode `init` qui est fournie par `NSObject`.

```
MaClasse *instance = [[MaClasse alloc] init];
```

Pour implémenter un nouvel initialisateur, vous déclarez une méthode d'instance dont le type de retour est `id`. Elle devra toujours respecter le format suivant :

```
- (id) initWithUnParametre:(id) anObject  
{  
    if (self = [super init])  
    {  
        // Initialisation ici  
    }  
    return self;  
}
```

Ce format permet de s'assurer que l'initialisateur parent est toujours appelé et qu'il peut éventuellement empêcher l'initialisation en renvoyant `nil` ou renvoyer une instance déjà existante (pour un singleton par exemple).

Les protocoles

L'héritage multiple étant impossible en Objective-C (comme en Java), un mécanisme est ajouté permettant de définir des protocoles qu'une classe s'engage à respecter (comme les interfaces en Java).

Déclaration d'un protocole

Un protocole se déclare dans un fichier d'en-tête selon un format qui ressemble fortement à une interface de classe mais sans variables d'instance.

```
@protocol MonProtocole  
  
- (void) uneMethodeDuProtocole;  
  
@end
```

Implémenter un protocole

Une classe déclare dans son interface la liste des protocoles qu'elle implémente.

```
@interface MaClasse : NSObject <MonProtocole, MonAutreProtocole> {
    // Variable d'instance
}

// Méthodes

@end
```

Les propriétés

On a vu que les variables d'instance d'une classe ne sont pas accessibles par les autres objets. La classe doit déclarer des accesseurs pour les variables d'instance qu'elle veut partager.

Le langage met à la disposition du développeur un mécanisme permettant de simplifier l'écriture des accesseurs : les propriétés. Une propriété est une variable d'instance pour laquelle des accesseurs sont définis.

Déclarer une propriété

On déclare les propriétés dans l'interface de la classe, au même endroit que les méthodes (en effet, les accesseurs sont des méthodes) :

```
@interface MaClasse : NSObject {
    NSString *myName;
}

@property NSString *myName;

@end
```

Cet exemple de code est équivalent à celui-ci :

```
@interface MaClasse : NSObject {
    NSString *myName;
}

- (NSString*) myName;
- (void) setMyName:(NSString*) myName;

@end
```

ATTENTION Les propriétés ne sont que du sucre syntaxique

La notion de propriété en Objective-C est du sucre syntaxique, c'est-à-dire une aide que le compilateur apporte au développeur en lui évitant de taper du code répétitif.

Implémenter les accesseurs

Vous pouvez choisir d'implémenter vous-même les accesseurs dans l'implémentation de votre classe ou demander au compilateur de le faire pour vous à l'aide du mot-clé `@synthesize`.

```
@implementation MaClasse

@synthesize myName;

// équivalent à
- (NSString*) myName
{
}
- (void) setMyName:(NSString*)newName
{
    myName = newName;
}

@end
```

Attributs des propriétés

Il est possible de fournir au compilateur des indications sur la propriété pour changer le code généré.

```
// Propriété en lecture seule
@property (readonly) NSString *string;

// Propriété en mode non-atomic
@property (nonatomic) NSString *string;

// L'objet assigné à cette propriété doit être retenu
@property (retain) NSString *string;

// Propriété en mode non-atomic et devant être retenu
@property (nonatomic, retain) NSString *string; ❶
```

- ❶ Le dernier exemple est le cas le plus fréquent. Nous détaillerons un peu plus loin la gestion de la mémoire en Objective-C et le mot-clé `retain`.

ASTUCE L'attribut nonatomic

Par défaut, le code généré pour les propriétés est construit pour être callable depuis plusieurs threads en même temps. Pour cela, un mécanisme de synchronisation est mis en place automatiquement autour des accès aux propriétés.

Dans la plupart des cas, ce mécanisme n'est pas utile, et le plus souvent vous déclarerez vos propriétés avec l'attribut `nonatomic`, ce qui permet un gain de performance.

La notation point

La notation point permet d'alléger l'appel des accesseurs (les accesseurs peuvent avoir été déclarés manuellement ou à l'aide du mot-clé `@property`). Ainsi, si un objet a des accesseurs pour une propriété `color`, on peut y faire référence de plusieurs manières.

Accès à une propriété via les accesseurs

```
UIColor *c = [objet color]
[objet setColor: c];
```

Accès à une propriété via la notation point

```
UIColor *c = objet.color;
objet.color = c;
```

Les sélecteurs : des pointeurs sur fonction

L'Objective-C étant un langage très dynamique, il est souvent utile de passer à une méthode un pointeur sur une fonction qui sera rappelé ultérieurement. Un sélecteur est un identifiant unique de méthode.

Le mot-clé `@selector` est ajouté en Objective-C pour obtenir un sélecteur. Il est associé au type `SEL`.

Un des moyens de déclencher un appel de méthode à l'aide d'un sélecteur est la méthode `performSelector:withObject:` de l'objet `NSObject`.

Exemple d'utilisation de `performSelector:`

```
SEL monSelecteur = @selector(maFonctionAvecUnArgument:);
[unObjet performSelector:monSelecteur withObject:arg1];

// Équivaut à :
[unObjet maFonctionAvecUnArgument:arg1];
```

Il est également possible, à l'aide de la méthode `respondsToSelector` : de savoir si un objet sait répondre à un sélecteur.

Exemple d'utilisation de `respondsToSelector`:

```
SEL monSelecteur = @selector(maFonctionAvecUnArgument:);
if ([unObjet respondsToSelector:monSelecteur]) {
    [unObjet performSelector:monSelecteur withObject:arg1];
}
```

Synchronisation de threads

Dans une application multithread, il est important de pouvoir empêcher certains morceaux de code de s'exécuter en parallèle.

L'Objective-C met à la disposition du développeur le mot-clé `@synchronize` qui prend en paramètre un objet et est suivi par un bloc de code.

Tant que le bloc de code n'a pas fini de s'exécuter, tous les autres threads ne pourront pas exécuter le même bloc (l'exécution des autres threads est suspendue à la première ligne jusqu'à ce que le premier thread ait terminé le traitement).

Exemple d'utilisation de `@synchronized`

```
@synchronized(self)
{
    // Ce code est protégé contre les exécutions en parallèle
}
```

La bibliothèque standard : le framework Foundation

Dans la plupart des langages, il existe une bibliothèque standard qui fournit des éléments génériques, utilisables dans toutes les applications.

En Objective-C, la bibliothèque standard s'appelle `Foundation`. Elle fournit les éléments nécessaires pour manipuler des chaînes de caractères, des listes, des dictionnaires, des dates, etc. Nous ne passerons en revue que les éléments les plus essentiels.

Chaînes de caractères

Les chaînes de caractères en Objective-C sont encapsulées dans des objets `NSString`.

Cet objet fournit les méthodes nécessaires aux manipulations classiques : mesurer la taille d'une chaîne, concaténer plusieurs chaînes, retrouver une occurrence d'une chaîne, etc.

Pour déclarer une chaîne de caractères dans votre application, la syntaxe `@""` permet de déclarer une constante du type `NSString` et de l'initialiser avec la chaîne fournie.

```
NSString *languageName = @"Objective-C";
```

La classe `NSString` fournit plusieurs méthodes statiques qui renvoient une chaîne déjà initialisée avec les paramètres donnés. Une des plus utiles est `stringWithFormat:` qui reprend les paramètres de formatage bien connus des développeurs C (syntaxe `printf`).

```
NSString *languageName = [NSString stringWithFormat:@"Mon langage est %@ %u", @"Objective-C", 2];
```

On remarque dans cet exemple que l'Objective-C ajoute un paramètre de formatage pour les paramètres de type `NSString` : `%@`.

APPROFONDIR Le spécificateur de formatage `%@`

Ce spécificateur ajouté par l'Objective-C peut être utilisé avec tous les objets du langage. En effet, il appelle la méthode `descriptionWithLocale:` ou `description` sur l'objet. Dans le cas de `NSString`, ces méthodes retournent simplement la valeur de l'instance.

La classe `NSString` est non mutable et ses instances ne peuvent donc pas être modifiées. Il existe une version mutable : `NSMutableString`.

Dans l'exemple suivant, on a une chaîne de caractères et on veut lui ajouter une autre chaîne, deux solutions sont possibles.

On peut créer une chaîne, puis en créer une deuxième à partir de la première :

```
NSString *string = @"Hello, ";  
string = [string stringByAppendingString:@"World"];
```

Ou bien utiliser un objet mutable qu'on initialise et qu'on modifie ensuite :

```
NSMutableString *string =  
    [NSMutableString stringWithString:@"Hello, "];  
[string appendString:@"World"];
```


APPROFONDIR Objets mutables

La plupart des objets de base en Objective-C sont non mutables, ce qui signifie qu'ils ne peuvent plus être modifiés après avoir été instanciés. C'est le cas par exemple de `NSString`, `NSNumber`, `NSArray` et `NSDictionary`.

Si vous souhaitez modifier l'instance, il faut utiliser une version mutable de l'objet ; ou créer une nouvelle instance à partir de l'instance existante.

La création de nouvelles instances d'objets non mutables a un coût important en mémoire et en temps d'exécution, mais la manipulation de ces objets est bien plus rapide que la manipulation d'objets mutables. En pratique, on choisira d'utiliser des objets mutables en fonction du nombre de modifications à apporter.

Listes

La classe `NSArray` fournit une abstraction de liste ordonnée de taille fixe.

```
NSArray *uneListe = [[NSArray alloc] initWithObjects:obj1, obj2, obj3, nil];
NSObject *obj = [uneListe objectAtIndex:2]; // obj3
```

Il existe également la classe `NSMutableArray` qui permet de modifier la liste dynamiquement.

Initialisation et utilisation d'une liste modifiable

```
NSMutableArray *uneListeModifiable = [[NSMutableArray alloc]
initWithCapacity:10];
[uneListeModifiable addObject:obj1];
[uneListeModifiable addObject:obj2];
[uneListeModifiable addObject:obj3];
NSLog(@"Taille de la liste: %u", [uneListeModifiable count]);
```

Parcourir rapidement une liste

L'Objective-C 2.0 a ajouté un moyen permettant d'énumérer rapidement le contenu d'une liste.

```
for (Type *element in array)
{
    // ...
}
```

On peut ainsi imprimer le contenu et la taille de chaque chaîne dans une liste d'objets `NSString` :

```
for (NSString *uneChaine in chaineListe)
{
    NSLog(@"Chaine: %@ Longueur: %u", uneChaine, [uneChaine length]);
}
```

Dictionnaire

La classe `NSDictionary` est une table de hachage qui pour une clé donnée garde une valeur. Comme pour les listes, il existe une version qui peut être modifiée (`NSMutableDictionary`).

```
NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];
[dict setObject:obj1 forKey:key1];
[dict setObject:obj2 forKey:key2];

NSObject *o = [dict objectForKey:key2]; //obj2
```

Le mécanisme de comptage de références

Dans le langage C, le développeur alloue des objets en appelant la méthode `malloc()` et les libère en appelant la méthode `free()`. En Objective-C, la bibliothèque standard fournit un mécanisme intégré à la classe `NSObject` qui permet de faciliter la gestion de la mémoire. C'est le mécanisme de comptage de références.

La gestion de la mémoire dans une application iPhone est un des points les plus importants. La plupart des problèmes d'optimisation ou de plantage rencontrés durant les projets iPhone sont liés à une mauvaise compréhension du mécanisme de comptage de références ou des règles d'utilisation qui y sont associées.

RAPPEL De l'importance d'une bonne gestion de la mémoire

Rappelons que dans un système Unix (ce qui est le cas de l'iPhone), la mémoire physique n'est pas directement accessible par les applications. Pour stocker des objets en mémoire, l'application doit demander au système de lui allouer de la mémoire. Le système garde une trace de toutes les zones mémoires allouées à l'application, mais ne peut pas savoir lesquelles sont encore utilisées.

Si l'application n'indique pas au système que les zones mémoires ne sont plus utilisées, l'espace mémoire sera toujours réservé et ne pourra pas être recyclé pour d'autres objets. Lorsque toute la mémoire est utilisée, le système n'a plus d'autres choix que de tuer l'application pour récupérer cette mémoire.

La gestion manuelle de la mémoire : point de ramasse-miettes !

Bien qu'un garbage collector soit intégré à la plate-forme Mac OS X 10.5, celui-ci n'est pas disponible pour la plate-forme iPhone. L'application est donc responsable de sa mémoire et doit libérer les objets quand ils ne sont plus utilisés.

RAPPEL Garbage collector ou ramasse-miettes

Dans des langages comme Java, PHP ou C#, la mémoire est gérée automatiquement par l'environnement d'exécution à l'aide d'un garbage collector : le développeur n'a pas besoin de détruire explicitement les objets. Ce mécanisme garde la trace des objets en mémoire et sait reconnaître si un objet n'est plus utilisé ; quand c'est le cas, il est supprimé.

Heureusement, le mécanisme de comptage des références est intégré à l'Objective-C 2.0 et est associé à quelques normes. L'ensemble facilite grandement le travail du développeur.

Vie et mort des objets

Création d'un objet

La classe `NSObject` définit la méthode `alloc` qui alloue la mémoire pour un objet. L'objet doit ensuite être initialisé en appelant une méthode `init...`

```
NSString *str = [[NSString alloc] init];
NSString *str2 = [[NSString alloc] initWithFormat:@"Hello: %@", str];
```

Libération de la mémoire d'un objet

Lorsqu'un objet n'est plus utilisé, la méthode `dealloc` est appelée. Cette méthode va libérer toute la mémoire utilisée par l'objet. En pratique, vous ne devez *jamais* appeler la méthode `dealloc` mais toujours la méthode `release` qui ne libérera effectivement la mémoire que si plus aucun objet n'y fait référence (grâce au mécanisme de comptage de références que nous détaillerons un peu plus loin).

```
NSString *str = [[NSString alloc] init];
NSString *str2 = [[NSString alloc] initWithFormat:@"Hello: %@", str];

[str release];
[str2 release];
```

Libération de la mémoire utilisée par vos objets

Vous devez surcharger la méthode `dealloc` dans vos propres classes pour libérer la mémoire qui est référencée par l'objet. N'oubliez pas d'appeler la méthode `dealloc` de la classe parent.

```
- (void)dealloc {
    [myView release];
    [super dealloc];
}
```

Risques d'une mauvaise gestion de la mémoire

Il existe plusieurs risques associés à la gestion de la mémoire, qui sont autant d'erreurs courantes.

Envoyer un message à un objet dont la mémoire a été libérée

Si l'application essaie d'envoyer un message à un objet qui a déjà été libéré, elle recevra une erreur et se terminera immédiatement.

Considérons le programme suivant :

```
int main(int argc, char *argv[]) {
    NSString *str = [[NSString alloc] initWithFormat:@"Hello %@", @"World"];

    [str release];

    [str isEqualToString:@""];
}
```

Lorsqu'on l'exécute, l'application est terminée par le système avec l'exception : `EXC_BAD_ACCESS (SIGSEGV)`, qui signifie que le programme a essayé de lire une zone mémoire qui ne lui appartient pas (en effet, elle était déjà libérée).

Libérer plusieurs fois la mémoire d'un objet

Lorsque l'application libère plusieurs fois la mémoire d'un objet, elle reçoit une exception et se termine immédiatement.

En effet, libérer une deuxième fois la mémoire d'un objet revient à appeler une méthode (`release`) sur un objet dont la mémoire a déjà été libérée.

ASTUCE Toujours mettre à nil les références d'objets après les avoir libérés

En donnant la valeur `nil` aux pointeurs des objets après les avoir libérés, vous vous assurez que votre programme n'essaiera pas d'envoyer un message à une instance qui n'existe plus et évitez ainsi des plantages de l'application.

```
[str release]
str = nil;
```

Cette solution permet d'éviter un plantage dans les deux exemples précédents.

Comptage de références

Il est relativement facile au sein d'une fonction ou d'un objet de savoir quand une zone mémoire n'est plus utilisée et qu'elle peut être libérée. Les choses deviennent beaucoup plus compliquées quand des zones mémoires sont partagées entre plusieurs objets : comment savoir si les autres objets ont fini de l'utiliser ?

Pour faciliter le travail du programmeur, les objets intègrent un compteur de références qui permet à l'objet de savoir combien d'autres objets maintiennent une référence vers lui. Chacun des objets tiers a l'obligation d'incrémenter ce compteur quand il commence à faire référence à l'objet, et de le décrémenter quand il n'y fait plus référence.

Incrémenter et décrémenter le compteur de références

Pour incrémenter et décrémenter le compteur de références, la classe `NSObject` propose les méthodes `retain` et `release`. On dit que l'objet est retenu puis libéré.

```
int main(int argc, char *argv[]) {
    NSLog(@"start");
    NSString *str = [[NSString alloc] initWithFormat:@"Hello %@",
@"World"];

    NSLog(@"Compteur de références: %u", [str retainCount]);

    [str retain];
    NSLog(@"Compteur de références: %u", [str retainCount]);

    [str release];
    NSLog(@"Compteur de références: %u", [str retainCount]);

    [str release];
    NSLog(@"done");
}
```

Ce programme donne la sortie suivante :

```
start
Compteur de références: 1
Compteur de références: 2
Compteur de références: 1
done
```

ATTENTION Utilisation de la méthode `retainCount`

La méthode `retainCount` est intéressante pour explorer le mécanisme de comptage des références, mais en pratique il ne faut jamais l'utiliser dans un programme pour décider de libérer ou pas un objet. Ce sont les règles de gestion de la mémoire que nous verrons un peu plus loin qui dictent quand il faut libérer ou pas un objet.

On remarque dans cet exemple que lorsque l'objet est alloué, son compteur de références est initialisé à la valeur 1. Il faudra donc appeler une fois et une seule fois `release` pour un objet alloué par le développeur.

Destruction des objets

L'implémentation de la méthode `release` décrémente le compteur de références et appelle la méthode `dealloc` immédiatement s'il est égal à 0. Il faut donc faire extrêmement attention à ne pas réutiliser un objet après l'avoir libéré car il peut avoir été détruit.

Libération retardée d'un objet

Dans certains cas, on souhaite écrire une méthode qui renvoie un objet sans que l'appelant n'ait à se soucier de le libérer.

Le code suivant est incorrect :

```
-(NSString*) giveMeSomething
{
    NSString *s = [[NSString] alloc] init];
    [s release];
    return s;
}
```

En effet, lorsque l'appelant récupérera l'objet `s`, celui-ci aura déjà été détruit et ne pourra plus être utilisé.

L'Objective-C met à disposition du développeur la méthode `autorelease` qui permet de dire à l'objet de se libérer un peu plus tard. Grâce à cette méthode, la classe appelante peut récupérer l'objet et l'utiliser sans se soucier de le libérer : l'objet se libérera tout seul ultérieurement.

Le code correct devient donc :

```
-(NSString*) giveMeSomething
{
    NSString *s = [[NSString alloc] init];
    [s autorelease];
    return s;
}
```

APPROFONDIR Les pools d'autorelease

Il n'y a rien de magique derrière le mécanisme de libération retardée. L'objet s'enregistre simplement auprès du pool d'autorelease le plus proche (par défaut, c'est celui qui a été créé dans la méthode `main`) et c'est ce dernier qui le libérera lors de sa prochaine exécution.

Le pool d'autorelease principal se vide quand la boucle de traitement des messages (*run loop*) n'a plus de messages à traiter et donc à un moment où toutes les méthodes du thread principal auront fini de s'exécuter. Si un objet souhaite conserver un objet qui a été programmé pour une libération retardée, il doit retenir explicitement l'objet (et le libérer plus tard).

Créer un pool d'autorelease

Dans certains cas, vous devrez créer vos propres pool d'autorelease, c'est le cas dans les threads ou dans des boucles longues qui utilisent beaucoup d'objets à libération retardée.

Dans l'exemple suivant, on crée un pool d'autorelease dans une boucle pour permettre de libérer les objets à chaque tour de boucle, et non pas une fois que la méthode a fini de s'exécuter.

```
for (NSString s in liste) {
    // Création d'un pool d'autorelease pour cette boucle
    NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];
    // Traitements ...
    // Les objets autorelease créés ici sont ajoutés à loopPool

    // Libération des objets autorelease
    [loopPool release];
}
```

Règles de gestion de la mémoire en Objective-C

Lorsqu'on appelle une méthode, il est essentiel de savoir comment l'objet a été créé et s'il est nécessaire de le libérer ou pas.

L'Objective-C impose une règle très importante pour répondre à cette question :

Vous devenez propriétaire des objets que vous avez créés en utilisant une méthode dont le nom commence par `alloc`, `new` ou contient `copy`. Vous devenez également propriétaire des objets que vous retenez. Vous devez libérer la mémoire des objets que vous détenez en appelant la méthode `release` ou `autorelease`. Dans les autres cas, vous ne devez jamais les appeler.

CONSEIL Relisez trois fois le paragraphe ci-dessus

Ce sont sans aucun doute les quelques lignes les plus importantes de ce chapitre !

La mauvaise compréhension des principes exposés ici est la source de très nombreux bogues difficiles à diagnostiquer. En prenant immédiatement le temps de bien comprendre le fonctionnement de la gestion de la mémoire, vous vous épargnerez de nombreux problèmes ultérieurs.

Pour compléter ce chapitre, vous pouvez également lire le *Memory Management Programming Guide* disponible dans les guides Apple.

Application de la règle

Grâce à cette règle, vous devriez toujours savoir avec certitude quand libérer la mémoire d'un objet. En particulier, vous n'avez pas besoin de libérer la mémoire des objets renvoyés par les méthodes statiques dont le nom ne commence pas par `alloc`.

```
NSString *str = [[NSString alloc] init];  
// Objet créé par la méthode alloc - Vous devez le libérer  
[str release];  
  
NSString *str = [NSString stringWithFormat:@""];  
// Objet créé par une méthode dont le nom ne commence pas par alloc ou  
new - Vous n'avez pas besoin de le libérer
```

Utilisation des objets à libération retardée

Les objets renvoyés par une méthode dont le nom ne commence pas par `alloc`, `new` ou ne contient pas `copy` peuvent être utilisés dans la méthode en cours. Ils peuvent également être renvoyés à une autre méthode appelante pour qui les mêmes règles s'appliqueront.

Par contre, ils ne peuvent pas être conservés (stockés dans une variable d'instance) et réutilisés plus tard puisqu'ils auront été libérés par le pool d'autorelease.

Si vous souhaitez conserver un de ces objets, vous devez le retenir. Il peut ainsi être affecté à une variable d'instance et réutilisé plus tard.

```
str = [NSString stringWithFormat:@"%"];
[str retain];
// L'objet est affecté à une variable d'instance et retenu.
// Il pourra être réutilisé plus tard.
// Il devra être libéré dans la méthode dealloc de la classe.
```

Propriétés et gestion de la mémoire

On utilisera souvent les accesseurs pour retenir les objets et les libérer.

Exemple de setter pour une propriété définie avec le mot-clé `retain`

```
// Setter pour la propriété object
- (void) setObject:(NSObject*) newObject
{
    if (object != nil)
    {
        [object release];
    }
    object = [newObject retain];
}
```

Avec ce type de setter, le compteur de références de l'objet est correctement mis à jour : on l'incrémente lorsqu'on reçoit l'objet pour être sûr que l'objet ne sera pas détruit tant qu'on détient un pointeur sur lui ; puis le compteur est décrémenté juste avant que la référence vers l'objet soit perdue.

C'est ce code qui est généré lorsque vous déclarez une propriété avec l'attribut `retain` et que vous utilisez `@synthesize` pour générer l'implémentation.

ATTENTION Ne confondez pas la propriété et la variable d'instance

Lorsque vous appelez une propriété définie avec le mot-clé `retain` en utilisant les accesseurs ou la notation point, le code de l'accesseur est utilisé et va correctement incrémenter et décrémenter le compteur de références de l'objet ; mais si vous faites directement appel à la variable d'instance, le compteur de références ne sera pas modifié.

```
// Correct :
self.object = [unAutreObjet donneMoiUnObjet];
// Incorrect :
object = [unAutreObjet donneMoiUnObjet];
// Correct : l'ancien objet est correctement libéré
self.object = nil;
// Incorrect : le compteur de références n'a pas été décrémenté.
// Il y a une fuite mémoire
object = nil;
```

Si vous devez allouer un objet et l'assigner à une propriété, il est d'usage de l'assigner tout d'abord à une variable locale, puis de l'affecter à la propriété (ce qui va entraîner une incrémentation du compteur de références) avant de le libérer.

```
NSString *aString = [[NSString alloc] init];
self.maString = aString;
[aString release];
```

Il est possible de faire autrement, mais le code est alors beaucoup moins clair et le risque d'erreurs plus grand.

Exemple de déclaration d'une propriété avec le mot-clé `retain`

```
@interface MaClasse {
}

@property (nonatomic, retain) NSObject *object;

@end

@implementation MaClasse

@synthesize object;

@end
```

Propriétés non retenues et références faibles

Par défaut, les propriétés sont déclarées avec le mot-clé `assign`.

```
@property NSString *string;  
@property NSString (assign) *string; // Equivalent
```

Avec ce mot-clé, les accesseurs ne vont pas prendre en charge la gestion de la mémoire et vont simplement assigner la nouvelle valeur à la variable d'instance.

APPROFONDIR Les références faibles

Les propriétés définies avec le mot-clé `assign` permettent de créer des références faibles entre les objets. Ainsi, un premier objet pointe sur un autre objet sans le retenir. On ne les utilise que lorsqu'on est sûr que l'objet cible ne risque pas d'être libéré pendant la durée de vie du premier objet. Le délégué d'un objet est souvent conservé par référence faible afin d'éviter des références circulaires.

Conclusion

Nous avons couvert les notions fondamentales de l'Objective-C. Vous êtes désormais prêt à développer vos premières applications.

La bibliothèque standard n'a été que survolée, mais vous verrez à l'usage que toutes les fonctions classiques que vous utilisiez dans d'autres langages sont également disponibles ; un petit tour dans la documentation permettra de les identifier rapidement.

Quand vous aurez lu les chapitres suivants, n'hésitez pas à revenir à ce chapitre et en particulier la partie sur la gestion de la mémoire pour vous assurer que tout est clair.

Enfin, certaines notions n'ont pas été abordées ici car elles ne sont pas indispensables pour développer des applications iPhone. Le guide Objective-C les décrit toutes en détail, et vous devriez le parcourir pour approfondir votre maîtrise du langage. Les catégories, par exemple, permettent d'étendre une classe existante (sans l'hériter) et sont un moyen très élégant d'enrichir la bibliothèque standard.

3

Premiers pas avec le SDK iPhone

Vous venez de vous familiariser avec l'Objective-C dont vous maîtrisez à présent les bases. Il est temps d'appréhender concrètement le développement iPhone. Ce chapitre est aussi là pour assouvir la curiosité des développeurs expérimentés qui découvrent l'environnement de développement Apple.

Nous ferons connaissance avec l'interface de Xcode et nous ferons également un tour rapide des fonctionnalités clés à bien connaître. Puis nous verrons quels sont les fichiers nécessaires pour une application minimaliste, leur rôle et le cycle de vie typique d'une application iPhone. Enfin, à travers différentes manières de faire le très classique « Hello World », nous verrons les diverses façons de créer une interface graphique et de l'afficher. Ce sont ces mêmes techniques que vous réutiliserez ensuite, en les enrichissant avec de nouveaux composants. Au passage, nous apprendrons à lancer l'application sur un iPhone et à la déboguer.

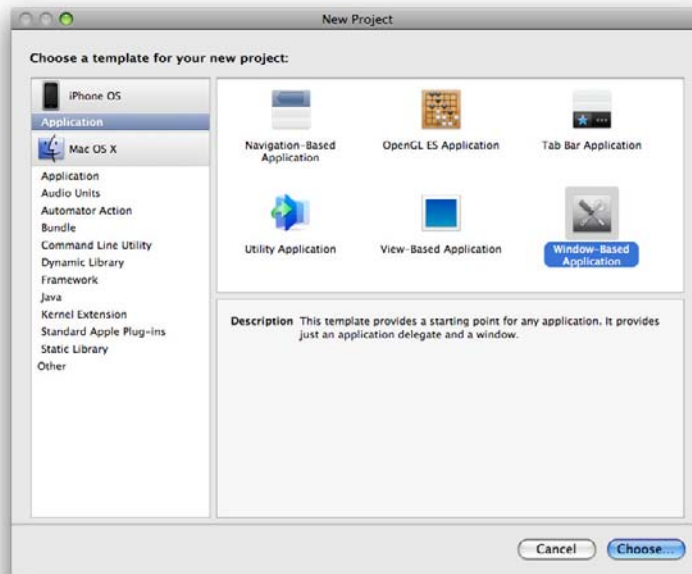
Après avoir lu ce chapitre, vous aurez fait le grand pas en avant dans le monde des développeurs d'applications iPhone. Vous commencerez à trouver vos marques, à comprendre. Le reste du livre vous permettra d'approfondir.

À la découverte de Xcode

Entrons directement dans le vif du sujet, lancez Xcode, fermez la fenêtre de bienvenue puis *File > New Project*. L'assistant de création d'un nouveau projet s'ouvre. À

gauche, vous retrouvez plusieurs catégories qui permettent de créer des projets pour Mac OS X ou pour iPhone. Sélectionnez *Application* dans *iPhone OS* puis *Window Based Application*, c'est-à-dire une nouvelle application basée sur une fenêtre.

Figure 3-1
Assistant de création d'un nouveau projet



Les modèles de projet

Xcode propose plusieurs templates de projet, qui sont autant de modèles d'applications standard. Le modèle le plus simple est « Window Based Application », c'est-à-dire une application qui ne fait que créer une fenêtre vide.

ESSENTIEL Fenêtre et vues

En général, les applications iPhone sont composées d'une seule fenêtre (*UIWindow*).

À l'intérieur de cette fenêtre, l'application crée des vues (*UIView*) qui peuvent occuper toute la surface de l'écran, ou seulement une partie. Nous verrons comment créer des hiérarchies de vues complexes, gérer les transitions d'une vue à l'autre (avec des effets de glissement ou d'apparition) rappelez-vous simplement que vous ne devez jamais chercher à créer d'autres fenêtres.

Les autres modèles de projet économisent quelques minutes de travail au développeur, mais ils risquent surtout de perdre le développeur débutant qui, n'ayant pas écrit lui-même le code, risque de ne plus comprendre comment l'application fonctionne. Aussi, il vaut toujours mieux s'appuyer sur le modèle *Window-Based Application* quand on démarre un projet.

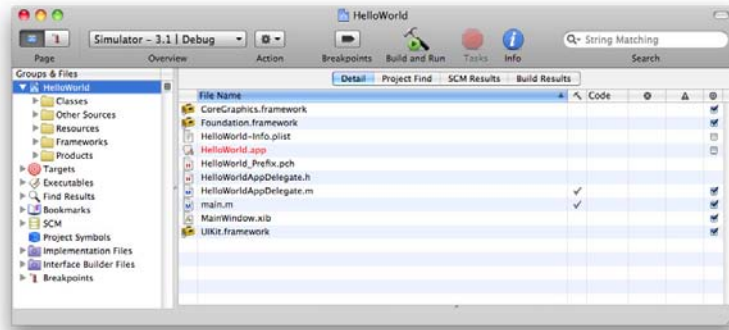
Les modèles pourront servir de source d'inspiration : créer un projet modèle, regarder le code généré et reproduire dans un modèle d'application basée sur une fenêtre. Ils serviront aussi aux développeurs expérimentés qui maîtrisent déjà bien le SDK, comprennent ce que fait l'assistant et pourront l'utiliser pour des développements express (une application construite en quelques heures pour valider un concept ou relever un défi !).

Découverte d'un projet iPhone vide

Après avoir choisi de créer un nouveau projet d'application basée sur une fenêtre, Xcode demande où enregistrer le projet et lance l'éditeur de code. Donnez le nom HelloWorld au projet.

Figure 3-2

La fenêtre Xcode après la création d'un projet basée sur une fenêtre.



Avant toute chose...

Si ce n'est pas déjà fait, la première chose à faire est probablement de cliquer sur le bouton *Build and Go* dans la barre d'outils : Xcode compile le projet et lance l'application dans le simulateur.

Figure 3-3

Votre première application iPhone – vous êtes sur la bonne voie.



Vous avez compilé et exécuté votre première application iPhone. Bravo ! Nous verrons plus loin dans ce chapitre comment lancer l'application sur votre iPhone.

Les groupes de fichiers créés par Xcode

La partie gauche de l'éditeur permet de lister les différents éléments du projet (voir illustration précédente). Sous « HelloWorld » on retrouve différents groupes de fichiers. Ces groupes ne correspondent pas à des répertoires dans le projet : il s'agit uniquement d'un moyen de ranger les fichiers dans Xcode.

En cliquant sur chaque groupe, la zone centrale supérieure liste le contenu du groupe. Par défaut, c'est l'élément racine « HelloWorld » qui est sélectionné et on voit donc le contenu de tous les groupes.

ASTUCE Les raccourcis clavier essentiels

Xcode propose plusieurs raccourcis clavier pour naviguer plus aisément dans les fichiers. Les maîtriser deviendra rapidement indispensable et vous devriez prendre le temps de vous familiariser avec dès maintenant.

Ouvrez le fichier `HelloWorldAppDelegate.m`, en appuyant sur *Option-Cmd-Haut*, vous basculez entre l'implémentation (le fichier `.m`) et la déclaration (le fichier `.h`).

Ouvrez maintenant le fichier `main.m`. Appuyez sur *Option-Cmd-Gauche* pour revenir en arrière dans l'historique des fichiers ou *Option-Cmd-Droite* pour aller en avant dans l'historique.

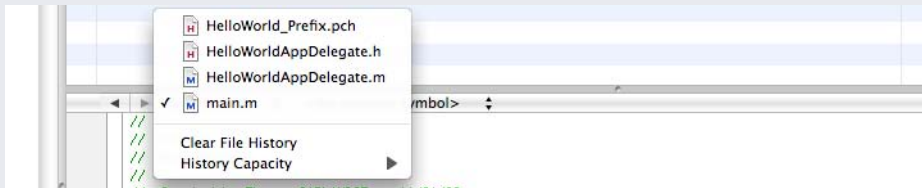


Figure 3-4 L'historique des fichiers visités

Vous pouvez aussi accéder à l'historique des derniers fichiers ouverts en cliquant sur le nom de fichier dans la barre qui sépare l'éditeur de texte de la liste des éléments.

Enfin, le raccourci *Cmd-Maj-D* permet de trouver rapidement un fichier ou un symbole dans l'ensemble du code source de l'application.

Tableau 3-1 Les groupes de fichiers créés par Xcode

Nom du groupe	Description
Classes	Dans ce groupe, on retrouvera toutes les classes du projet : fichiers d'en-tête (<code>.h</code>) et fichiers sources (<code>.m</code>). Vous pourrez créer des sous-groupes pour organiser vos classes.
Other Sources	Ce sont les autres fichiers sources du projet, ceux qui ne sont pas basés sur des classes Objective-C. Vous n'aurez probablement pas besoin d'ajouter de fichiers ici : tous vos développements seront faits sous forme de classe. Par défaut, on retrouve ici le fichier <code>main.m</code> qui ne contient que la fonction C classique <code>main()</code> : c'est le point d'entrée de l'application.

Tableau 3–1 Les groupes de fichiers créés par Xcode (suite)

Nom du groupe	Description
Ressources	Toutes les ressources du projet seront stockées dans ce groupe. Les ressources sont un élément très important d'un projet iPhone. On retrouvera notamment : les fichiers de description d'interface (.xib), le descripteur d'application (fichier <code>Info.plist</code>) et tous les éléments graphiques du projet (fichiers png, jpeg, etc.).
Frameworks	Ce groupe contient la liste des frameworks utilisés par l'application. Un framework est une bibliothèque qui sera compilée statiquement ou dynamiquement avec l'application.
Products	Liste les éléments produits par le compilateur. Dans notre cas, c'est le package : <code>HelloWorld.app</code> .

Fichiers générés lors de la création du projet

Lors de la création du projet, Xcode a généré plusieurs fichiers. Il utilise le nom du projet comme racine pour les nommer.

Tableau 3–2 Fichiers générés lors de la création du projet

Fichier	Description
<code>main.m</code>	Fichier contenant la fonction <code>main()</code> de l'application : point d'entrée du programme.
<code>HelloWorld_Prefix.pch</code>	Fichier d'en-tête qui est préfixé à l'ensemble des fichiers sources du projet. Permet de ne pas répéter les mêmes directives <code>#import</code> dans tous les fichiers. Par défaut, il fait appel aux en-têtes du framework Foundation et aux en-têtes de UIKit.
<code>HelloWorld-Info.plist</code>	Ce fichier de propriétés fournit des informations sur l'application. Il est utilisé par iPhone pour savoir quel est le nom de l'application, quelle icône afficher dans le menu, quelle fenêtre ouvrir lors du premier lancement de l'application, s'il faut lancer l'application en mode paysage, etc. Depuis la version 3 du SDK, le nom de ce fichier contient le nom de l'application. On utilise souvent l'ancien nom générique <code>Info.plist</code> pour s'y référer.
<code>HelloWorldAppDelegate.h</code> <code>HelloWorldAppDelegate.m</code>	La classe <code>HelloWorldAppDelegate</code> est la classe déléguée de votre application. Nous reviendrons très rapidement sur ce design pattern très important dans l'environnement iPhone. Disons simplement pour l'instant qu'elle permet au développeur d'enrichir le comportement de la classe <code>UIApplication</code> .
<code>MainWindow.xib</code>	Fichier de description de la fenêtre. Ce fichier est destiné à être édité visuellement avec Interface Builder. Le fichier <code>MainWindow.xib</code> généré contient une fenêtre vide et référence la classe <code>HelloWorldAppDelegate</code> .

Comprendre le code généré par Xcode

Lorsque l'application est exécutée par iPhone ou par le simulateur, le point d'entrée est toujours le fichier `main.m`. Les étapes déroulées ensuite dépendent de choix faits par le développeur (en particulier en fonction de l'utilisation ou pas d'Interface Builder).

Dans le cas d'une application générée avec le modèle de projet basé sur une fenêtre, les étapes sont les suivantes :

- 1 L'exécution démarre dans la fonction `main()` du fichier `main.m`.
- 2 La fonction `main` crée un pool de libération automatique de la mémoire (objet `NSAutoreleasePool` ; voir à ce sujet le paragraphe sur la gestion de la mémoire dans le chapitre précédent).
- 3 La fonction `main` appelle la fonction `UIApplicationMain` (qui est une fonction de base d'UIKit) et lui passe la main.
- 4 `UIApplicationMain` crée un objet `UIApplication`.
- 5 `UIApplicationMain` lit le fichier `Info.plist` pour obtenir le nom du fichier NIB à charger (les fichiers NIB sont la version compilée des fichiers XIB). Elle instancie la fenêtre décrite dans le fichier NIB, ainsi que le délégué de l'application (la classe `HelloWorldAppDelegate` dans notre cas).
- 6 Une boucle de gestion des événements est démarrée par l'objet `UIApplication`.
- 7 La méthode `applicationDidFinishLaunching:` du délégué est appelée.
- 8 La boucle de gestion des événements traite un par un les événements reçus par l'application.
- 9 Quand l'utilisateur ferme l'application (en cliquant sur le bouton *Home* par exemple), la boucle s'arrête, l'application rend la main à la fonction qui vide le pool mémoire créé plus tôt et se termine.

ESSENTIEL Boucle de gestion des événements

Une boucle de gestion des événements est une boucle sans fin qui traite une file d'attente d'événements. Toutes les applications iPhone comportent une boucle principale de gestion des événements, ou *main run loop*, qui est chargée de traiter tous les événements ajoutés à la file par le système (quand l'utilisateur touche l'écran par exemple).

Lorsque la boucle principale de gestion des événements dépile un événement, elle cherche la vue (bouton, image, zone de texte, etc.) qui pourra le traiter et lui passe l'événement.

La boucle se termine quand un événement spécial est reçu : quand l'utilisateur souhaite fermer l'application par exemple.

Rôle du fichier Info.plist

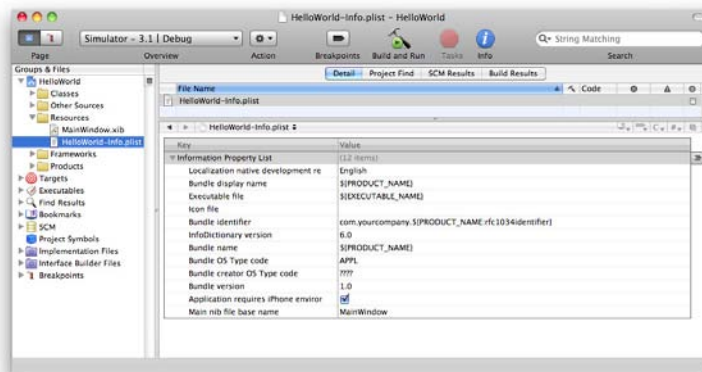
Le fichier `Info.plist` (ou `NomApplication-Info.plist`) est placé à la racine de l'application. Il définit des propriétés qui sont lues par l'iPhone lors de l'installation et de l'exécution de l'application.

Vous trouverez la liste exhaustive des propriétés dans le guide de programmation d'applications iPhone.

Édition du fichier Info.plist

C'est un fichier XML, mais Xcode intègre un éditeur adapté qui facilite la modification de ce fichier.

Figure 3-5
Édition du fichier Info.plist dans Xcode



ASTUCE Édition des fichiers XML de propriétés

Lorsque vous ouvrez un fichier de propriétés, Xcode vous présente un éditeur adapté au type de fichier ouvert. Dans le cas du fichier `Info.plist`, il affiche des libellés complets au lieu du nom de la propriété, propose la liste des éléments autorisés quand on ajoute une propriété, etc.

Pour basculer entre le mode d'édition standard et le mode d'édition spécifique `Info.plist`, vous pouvez utiliser le menu `View > Property List Type`.

Nom de l'application et choix de l'icône

Comme vous avez pu le constater en lançant l'application dans le simulateur, celle-ci n'a pas encore d'icône et un carré blanc s'affiche à la place.

La propriété `CFBundleIconFile` (Icon file) permet d'indiquer le nom d'un fichier `.png` qui sera utilisé comme icône pour l'application. Si vous n'indiquez pas cette propriété, un fichier nommé `Icon.png` sera recherché. L'icône doit être ajoutée comme ressource au projet, et faire 57 x 57 pixels.

Par défaut, l'effet de halo lumineux et les bords arrondis sont ajoutés automatiquement par iPhone. Vous pouvez lui indiquer que vous les avez déjà ajoutés (ou que vous ne souhaitez pas qu'il les ajoute) en définissant la propriété `UIPreRenderedIcon`.

La propriété `CFBundleDisplayName` (Bundle display name) indique le titre de l'application, tel qu'il apparaîtra sur l'écran d'accueil de l'iPhone.

Personnalisation de l'écran de démarrage

Lorsque l'utilisateur choisit de lancer une application, iPhone déclenche une animation et l'application semble s'ouvrir instantanément. Malheureusement, il faut plus de temps à l'application pour se charger en mémoire, lire ses ressources, ouvrir puis afficher la première fenêtre.

En attendant que l'application prenne la main et que la fenêtre soit affichée, iPhone affiche une image statique qui permet de faire illusion et donne l'impression que l'application est déjà chargée : c'est le fichier `Default.png` à la racine des ressources de l'application.

Apple recommande que vous utilisiez ce fichier pour afficher une image de l'interface telle qu'elle apparaîtra quand l'application sera chargée. C'est ce que font les applications standard de l'iPhone. Vous pouvez également vous en servir pour afficher un *splash screen*.

Rôle du délégué de l'application

Le délégué de l'application (`HelloWorldAppDelegate`) est le seul des fichiers sources générés que vous devrez modifier (et bien sûr, vous en ajouterez d'autres). Grâce au délégué, le développeur peut ajouter du code qui sera exécuté dans certains moments clés de la vie de l'application.

DESIGN PATTERN Délégation de contrôle

Le design pattern délégation de contrôle permet à une classe de s'appuyer sur une autre pour enrichir son propre comportement. Son intérêt principal est d'éviter d'avoir à dériver une classe pour modifier son comportement.

Prenons un exemple : soit une classe `Voiture`, qui permet via différentes méthodes d'avancer, tourner, reculer, etc. Cette classe définit un protocole `VoitureDélégué` dans lequel on retrouvera les méthodes `voitureSarrete`, `voitureDémarrre`. Lorsque la voiture démarre ou s'arrête, les méthodes correspondantes du délégué sont appelées.

Le développeur peut donc enrichir le comportement de la classe `Voiture`, sans la dériver ; et le développeur de la classe `Voiture` n'a pas besoin de documenter le comportement interne de sa classe puisque les points d'extension sont clairement définis via le délégué.

Ce design pattern est très largement utilisé dans Cocoa : un bouton prévient son délégué quand on clique dessus, une liste d'éléments fait appel à son délégué pour savoir quel élément afficher, une application prévient son délégué quand l'application a fini de se charger ou quand le système veut la fermer, etc.

Tous les points d'extensions proposés par le délégué de `UIApplication` sont décrits dans la documentation (faites une recherche sur `UIApplicationDelegate`), les plus importants et leurs utilisations classiques sont décrits ci-après.

applicationDidFinishLaunching: – Votre application a fini de se lancer

Cette méthode est appelée lorsque la classe `UIApplication` a fini de charger la vue décrite dans le fichier XIB, et que la boucle de gestion des événements est prête.

Vous pouvez alors ajouter du contenu dans la fenêtre, charger d'autres fichiers XIB, lancer des requêtes réseau pour obtenir des contenus, etc. C'est ici que vous ajouterez le code de démarrage de votre application. Dans le code généré par défaut, on retrouve les lignes suivantes :

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {  
    // Override point for customization after application launch  
    [window makeKeyAndVisible];  
}
```

Le délégué envoie le message `makeKeyAndVisible` à l'objet `window`, qui a pour effet de remplacer l'image de démarrage par la fenêtre de votre application.

CONSEIL Limitez au maximum les traitements lors du lancement de l'application

Pour minimiser le temps de démarrage de l'application, il est essentiel de réduire au maximum les traitements faits dans cette méthode.

Tous les traitements qui ne sont pas indispensables doivent être retardés. C'est le seul moyen de faire une application qui démarre vite.

applicationWillTerminate: – Votre application va se terminer

Quand l'utilisateur quitte l'application (en appuyant sur le bouton *Home*, en répondant à un appel ou un SMS), le délégué de l'application reçoit cet événement avant que l'application ne se ferme.

CONSEIL Prévoyez du temps avant que l'application se termine

Quand votre délégué reçoit ce message, l'application est déjà en train de se fermer et il est possible qu'elle soit tuée avant même que la méthode ait fini de s'exécuter. Si vous avez besoin d'exécuter des traitements, utilisez plutôt la méthode `applicationWillResignActive` qui vous laissera plus de temps.

applicationDidReceiveMemoryWarning: – Votre application doit libérer de la mémoire

Cette méthode est liée à un mécanisme très important dans l'iPhone : les avertissements de mémoire. Elle est appelée quand votre application consomme trop de mémoire et qu'elle atteint les limites du système. Dans ce cas, il faut libérer de la mémoire en relâchant des objets qui ne sont pas utilisés pour l'instant.

Si votre application ne réagit pas à cet événement, et qu'elle ne libère pas de mémoire, le système risque de tuer l'application. C'est une des sources importantes de plantages.

ASTUCE Utiliser les classes fournies par le SDK

Les classes fournies par le SDK sont déjà capables de libérer de la mémoire quand cet événement est reçu par l'application. Par exemple, le contrôleur associé à une barre d'onglets (`UITabBarController`) va automatiquement libérer la mémoire utilisée par les vues (les onglets) qui ne sont pas visibles.

En pratique, la méthode `didReceiveMemoryWarning:` du délégué d'application est peu utilisée.

Nous reviendrons sur les avertissements mémoire au chapitre 7 consacré aux contrôleurs de vue, qui sont souvent responsables de la majorité de la consommation mémoire (puisqu'ils détiennent les vues et tous les éléments graphiques).

applicationWillResignActive: – Votre application passe en arrière-plan

Lorsqu'une notification est présentée à l'utilisateur (un appel entrant, un SMS, un événement de calendrier, etc.), l'application continue à s'exécuter, mais elle ne reçoit plus aucun événement (une fenêtre est affichée par le système au-dessus de l'application).

Si l'utilisateur décide de répondre à la notification (en répondant à l'appel ou en choisissant d'ouvrir le SMS ou l'événement calendrier), votre application recevra l'événement `applicationWillTerminate:` avant de se fermer. Sinon, l'application reçoit l'événement `applicationDidBecomeActive:`.

Vous pouvez utiliser cet événement (et `applicationDidBecomeActive:` qui est envoyé quand l'application redevient active) pour mettre votre application en pause.

BONNE PRATIQUE Enregistrer l'état de l'application

Il est recommandé d'utiliser cette méthode pour enregistrer l'état dans lequel se trouve l'application (quel est l'onglet sélectionné, quelle est la page affichée). Quand l'utilisateur relancera l'application, il se retrouvera directement là où il était en quittant. N'oubliez pas en effet que sur l'iPhone, l'utilisateur ne choisit pas de fermer l'application : il peut y être contraint pour répondre au téléphone ! L'application pourra recharger l'état courant lors du prochain lancement (dans la méthode `applicationDidFinishLaunching:`). Nous verrons plus tard où et comment enregistrer les informations sur l'état actuel de l'application.

Rôle d'Interface Builder

À proprement parler, Interface Builder ne joue aucun rôle durant l'exécution de l'application. Interface Builder est l'outil qui permet de construire les fichiers `.xib` et il est important de comprendre que via cet outil, on peut instancier des objets graphiques (des fenêtres, des vues, des boutons, etc.), mais aussi d'autres objets : le délégué d'application, comme c'est le cas dans notre exemple, ou des contrôleurs de vue comme nous le verrons plus tard.

Interface Builder joue un deuxième rôle qui est de définir les branchements entre les instances créées par le développeur et les instances créées à partir de fichier NIB (version compilée des XIB).

ESSENTIEL Comprendre Interface Builder

Interface Builder est un outil très puissant qui peut sembler magique au premier abord. Son principe de fonctionnement est pourtant simple :

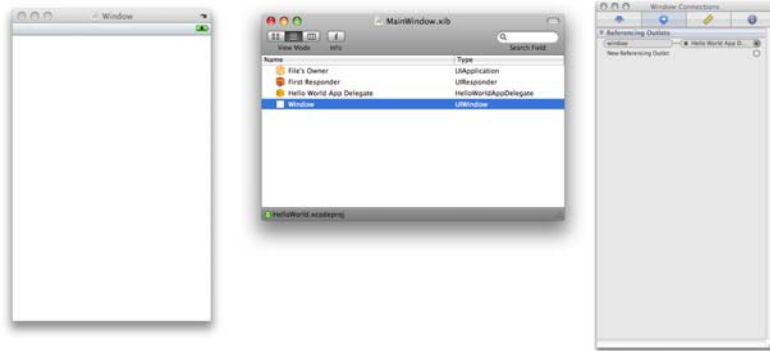
1. Le développeur utilise Interface Builder pour assembler visuellement des composants, définir leur type, leurs propriétés et les liens entre ces objets.
2. Interface Builder sérialise (décrit) l'ensemble de ces objets, leurs propriétés et leurs relations dans un fichier XML : le fichier XIB.
3. Le fichier XIB est compilé par Xcode pour réduire sa taille et faciliter sa lecture. On obtient un fichier binaire contenant exactement les mêmes informations : le fichier NIB.
4. Lors de l'exécution, le fichier NIB est lu par le framework Cocoa et tous les objets décrits à l'intérieur sont recréés en mémoire avec les mêmes propriétés.

Découverte d'Interface Builder

En double-cliquant dans Xcode sur le fichier `MainWindow.xib`, on ouvre Interface Builder qui affiche la fenêtre et plusieurs inspecteurs.

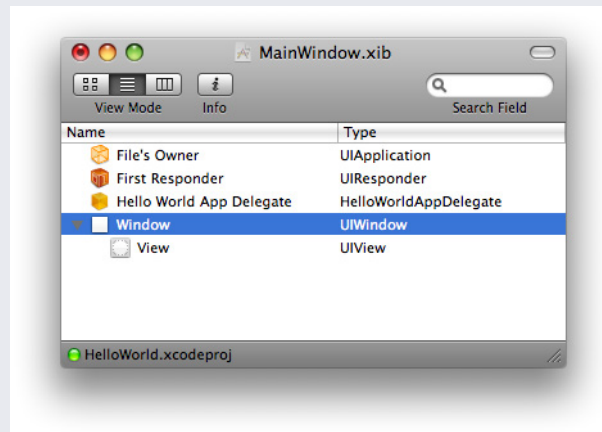
Figure 3-6

MainWindow.xib ouvert dans Interface Builder : à gauche, la prévisualisation de la fenêtre ; au centre, les éléments contenus dans le fichier ; à droite, l'inspecteur d'objet.



ASTUCE Utiliser toujours la vue en liste dans Interface Builder

Vous devriez toujours utiliser la vue en liste dans Interface Builder, en effet celle-ci permet de bien voir le nom des éléments mais surtout, elle permet de voir en même temps le type des objets et d'afficher la hiérarchie des éléments (quand une vue est contenue par une autre).

**Figure 3-7** Visualisation d'une hiérarchie d'éléments en mode liste

Sélectionner l'objet `Window` dans la fenêtre principale (celle qui affiche la liste des éléments contenus dans le fichier). Dans l'inspecteur d'objets (s'il ne s'affiche pas, il est disponible via le menu *Tools > Inspector* ou *Maj-Pomme-I*), sélectionnez le deuxième onglet pour afficher les connexions de l'objet.

On voit ici que l'objet `Window` est relié à la classe « Hello World App Delegate ». C'est grâce à cette liaison que la propriété `window` pointe vers la fenêtre décrite dans le fichier NIB.

Figure 3-8
Visualisation de la connexion
entre l'objet Window
et le délégué



Voyons maintenant :

- 1 Comment est déclarée la propriété `window`.
- 2 Comment créer un lien entre un objet dans Interface Builder et une propriété.

Définition d'une propriété pour Interface Builder

La classe `HelloWorldAppDelegate` définit une propriété `window` de type `UIWindow`. Dans la déclaration de la propriété, on remarque l'ajout du mot-clé `IBOutlet`.

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

@interface HelloWorldAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@end
```

Le mot-clé `IBOutlet` permet à Interface Builder de savoir qu'il *peut* raccorder une instance à cette propriété.

Créer un lien entre Interface Builder et le code de l'application

La manipulation permettant de créer un lien entre une instance d'objet créée par Interface Builder et une propriété du code de l'application est très simple, mais un peu surprenante pour un développeur découvrant l'environnement Xcode. Le lien existant a été mis en place automatiquement lors de la création du projet. Vous pouvez le supprimer en cliquant sur la croix à gauche de la bulle « Hello World App Delegate ».

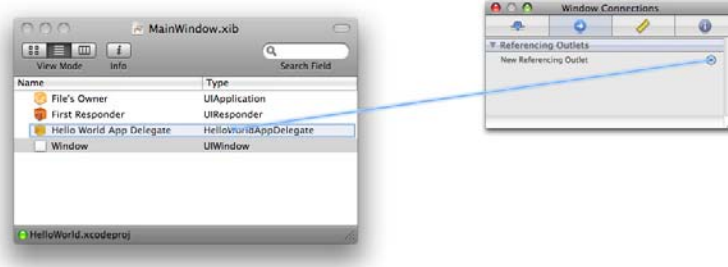
Sauvez le fichier `MainWindow.xib` modifié, relancez la compilation et l'exécution (bouton *Build And Go* dans Xcode). Le simulateur s'ouvre mais cette fois-ci, la fenêtre est complètement noire.

Que s'est-il passé ? Le lien entre l'objet `HelloWorldAppDelegate` et l'instance de fenêtre n'existe plus. La propriété `window` du délégué est donc égale à `nil` (ce que vous pourriez vérifier en ajoutant un message de débogage). Le message

`makeKeyAndVisible`: envoyé dans la méthode `applicationDidFinishLaunching`: est envoyé à `nil`, ce qui n'a aucun effet. La fenêtre existe (elle a été instanciée lors du chargement du fichier NIB) mais elle ne reçoit jamais le message lui indiquant de s'afficher.

Figure 3-9

Glisser-déplacer depuis le point *New Referencing Outlet* jusqu'à l'objet `HelloWorldAppDelegate`



Pour recréer le lien, il faut à nouveau afficher les connexions de l'objet `Window`, cliquer sur le point à droite du message *New Referencing Outlet* puis glisser jusqu'à l'objet `AppDelegate` (dans la fenêtre principale). Un menu s'ouvre et affiche les propriétés de l'objet `HelloWorldAppDelegate` auxquelles il est possible de lier l'objet `Window`. La seule propriété déclarée dans `HelloWorldAppDelegate` avec le mot-clé `IBOutlet` et de type compatible avec `UIWindow` est `window`.

Figure 3-10

La liste des propriétés compatibles s'affiche quand on lâche le glisser-déplacer.

Name	Type
File's Owner	UIApplication
First Responder	UIResponder
Hello World App Delegate	HelloWorldAppDelegate
Window	UIWindow

CONSEIL Aux allergiques du glisser-déplacer

Certains développeurs auront peut-être une réaction allergique à la programmation par glisser-déplacer. Qu'ils se rassurent, l'utilisation d'Interface Builder n'est jamais obligatoire et le développeur peut choisir de créer et de lier les objets en utilisant uniquement du code.

Le développeur expérimenté sera donc libre d'utiliser ou pas Interface Builder, mais le mécanisme mis en jeu est essentiel au programmeur Mac OS X : il permet de comprendre les exemples Apple, le code généré par l'assistant Nouveau Projet ou écrit par d'autres développeurs.

Enfin, il permet de gagner un temps précieux pour construire les vues. À l'usage, vous verrez que Interface Builder est un outil extrêmement puissant et efficace.

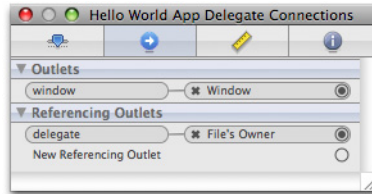
Déclaration du lien entre l'application et son délégué

De la même façon que la fenêtre est liée au délégué de l'application, le délégué de l'application est lié à l'objet `UIApplication` via une connexion Interface Builder.

En sélectionnant l'objet `HelloWorldAppDelegate`, on voit une fenêtre de connexion un peu différente.

Figure 3-11

L'objet `HelloWorldAppDelegate` a des connexions entrantes et sortantes.



La zone *Outlets* liste les points de connexion définis par la classe et les objets auxquels ils sont raccordés. La zone *Referencing Outlets* liste les objets qui référencent celui-ci.

On voit que le délégué d'application est référencé par le File's Owner et on voit dans la fenêtre principale que le File's Owner est un objet `UIApplication`.

ESSENTIEL Interface Builder – File's Owner

Dans Interface Builder, le File's Owner est toujours la classe qui va charger le fichier XIB. Il est de la responsabilité du développeur de s'assurer que cette information est correctement renseignée, sinon le chargement du fichier NIB ne pourra pas se dérouler correctement.

Le File's Owner ne fait pas partie du fichier XIB et est instancié par le développeur.

Comme nous l'avons vu précédemment, le fichier XIB principal est chargé par `UIApplication`. Dans la très grande majorité des cas, vos autres fichiers XIB seront chargés par un objet de type `UIViewController` ou une classe dérivée.

Hommage à MM. Kernighan et Ritchie

Comme le veut la tradition, notre premier programme affichera le message « hello, world » et bien sûr, il y a plus d'une façon de le faire.

Utilisation de la console

Modifiez ainsi la méthode `applicationDidFinishLaunching:` de la classe `HelloWorldAppDelegate` :

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {  
    // Override point for customization after application launch  
    [window makeKeyAndVisible];  
  
    NSLog(@"hello, world");  
}
```

Et profitez-en pour ajouter la méthode suivante :

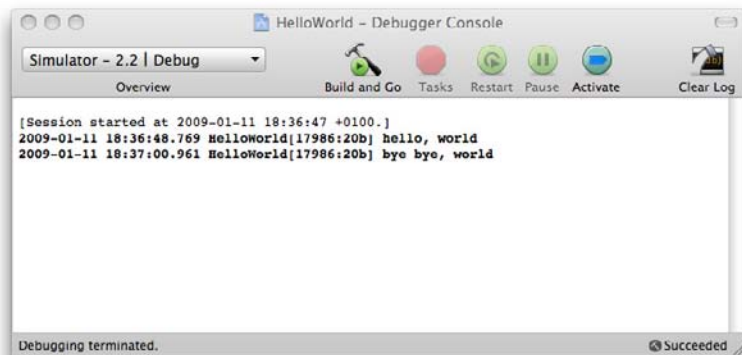
```
- (void)applicationWillTerminate:(UIApplication *)application {  
    NSLog(@"bye bye, world");  
}
```

Lancez la compilation et l'exécution, puis ouvrez la console (via le menu *Run > Console* ou avec le raccourci *Maj-Pomme-R*). Fermez ensuite l'application en cliquant sur le bouton *Home* du simulateur.

Voici ce que vous devriez obtenir :

Figure 3-12

La console affiche les messages envoyés par l'application.



CONSEIL Usez et abusez de la console

La console est un outil extrêmement utile pour comprendre et suivre le fonctionnement de votre application. N'hésitez pas à l'utiliser intensivement.

Son seul défaut est l'impact négatif qu'elle peut avoir sur les performances, mais il est toujours possible de supprimer tous les messages avant de publier l'application, par exemple en définissant dans `HelloWorld_Prefix.pch` une macro `NSLog` vide :

```
#define NSLog(s,...) /* nothing */
```

Création d'une nouvelle vue avec Interface Builder

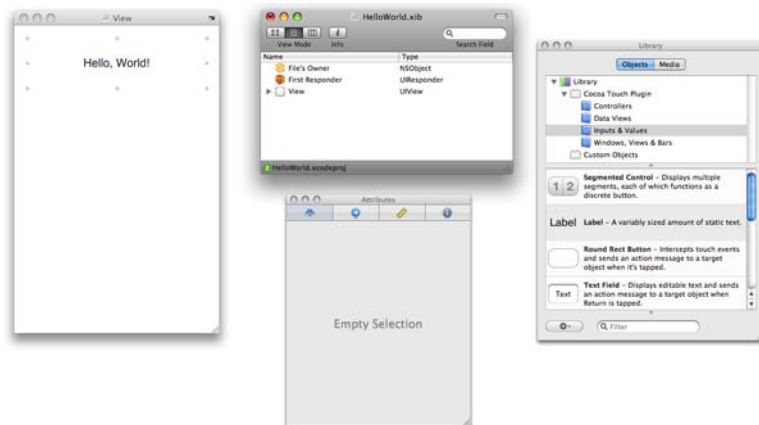
Ajouter une nouvelle vue au projet

Dans Xcode, sélectionnez *File > New File...* puis *View XIB* dans la catégorie *iPhone OS – User Interfaces*. Donnez-lui le nom *HelloWorld*, et Xcode l'ajoute au projet comme ressource.

Double-cliquez sur le fichier pour l'ouvrir dans Interface Builder. En utilisant la bibliothèque de composants (menu *Tools > Library*) ajoutez un objet `UILabel` et modifiez le texte.

Figure 3-13

Ajout d'un label dans la nouvelle vue



Faire appel à la vue

La nouvelle vue étant créée, il faut maintenant la charger et l'afficher.

Cela se fait toujours dans la méthode `applicationDidFinishLaunching:`.

Exemple de chargement d'un contrôleur de vue avec un fichier NIB lors du lancement de l'application.

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    UIViewController *helloWorldViewController =
        [[UIViewController alloc]
         initWithNibName:@"HelloWorld" bundle:nil];
    [window addSubview:helloWorldViewController.view];
    [helloWorldViewController release];
    [window makeKeyAndVisible];

    NSLog(@"hello, world");
}
```

La première ligne ajoutée crée un objet `UIViewController`, lui alloue de la mémoire et l'initialise en lui passant le nom du fichier NIB à charger. Celui-ci sera trouvé automatiquement dans les ressources de l'application.

La deuxième ligne ajoute la vue du contrôleur (donc la vue créée dans Interface Builder) à la fenêtre principale.

ESSENTIEL La classe `UIViewController` et le design pattern MVC

Nous reviendrons sur cette classe très importante, et le design pattern qui va avec (Modèle-Vue-Contrôleur). Ayez en tête que les objets de type `UIViewController` (les contrôleurs) sont chargés de gérer une ou plusieurs vues, de les initialiser, de les afficher et de faire le lien entre les données de l'application (le modèle) et les composants graphiques qui les représentent (la vue).

Vous pouvez à présent essayer de lancer l'application. Celle-ci va provoquer une erreur et se terminer. Sur la console, vous trouvez le message suivant (voir figure 3-14) :

Figure 3-14

Le programme reçoit une exception « loaded the HelloWorld nib but the view outlet was not set ».



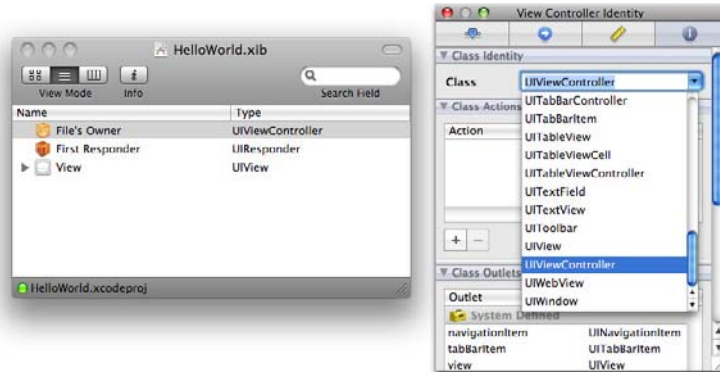
ERREUR CLASSIQUE Ne pas oublier de lier la vue

Cette erreur est due au fait que nous n'avons pas indiqué à Interface Builder où lier la vue qu'il a créée. La classe `UIViewController` indique donc qu'elle a chargé le nib mais que la vue n'a pas été définie, ce qui provoque une exception.

Dans la fenêtre principale de Interface Builder, sélectionnez l'objet `File's Owner` et en utilisant le quatrième onglet de l'inspecteur, vérifiez que son type est bien `UIViewController`.

Figure 3–15

On définit le type des objets dans le quatrième onglet d'Interface Builder.

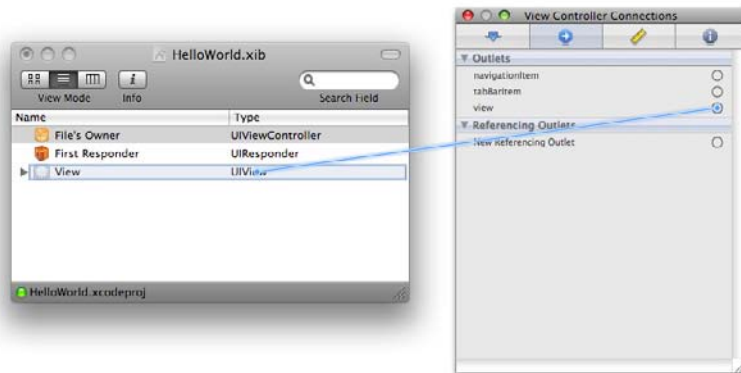


Tous les objets `UIViewController` ont une propriété `view` qui peut être définie grâce à Interface Builder. Nous allons donc attacher la vue créée au contrôleur :

- 1 Sélectionnez l'objet `File's Owner` dans la fenêtre principale.
- 2 Sélectionnez l'onglet *liaisons* (le deuxième onglet) dans l'inspecteur.
- 3 Établissez la liaison entre la propriété `view` et la vue qui est représentée dans la fenêtre principale.

Figure 3–16

Reliez la vue créée dans Interface Builder à la propriété `view` de l'objet `ViewController`.



Relancez l'application, vous devez voir apparaître la fenêtre suivante dans le simulateur.

Figure 3-17
La vue HelloWorld affichée
dans le simulateur



Si vous prenez le temps de créer une vue plus riche dans Interface Builder (avec des boutons, des images, etc.) vous verrez que sans ajouter une ligne de code, on peut déjà afficher des interfaces complexes. C'est la force d'Interface Builder.

Création d'une nouvelle vue en code

La vue utilisée ici étant très simple, nous pouvons très facilement la reproduire avec du code.

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    UIView *helloWorldView = [[UIView alloc]
                               initWithFrame:CGRectMake(0, 0, 320, 460)];
    UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(20, 50, 280, 20)];
    label.text = @"hello, world";
    label.textAlignment = UITextAlignmentCenter;
    [helloWorldView addSubview:label];
    [window addSubview:helloWorldView];
    [label release];
    [helloWorldView release];

    // Override point for customization after application launch
    [window makeKeyAndVisible];

    NSLog(@"hello, world");
}
```

Créer une vue

La création d'une vue se fait en instanciant un objet de type `UIView`. Le constructeur le plus fréquemment utilisé prend le paramètre `frame` qui indique les coordonnées de cette vue dans le système de coordonnées de la vue parente. Ce paramètre est une structure du type `CGRect`.

On la crée en appelant la fonction `CGRectMake(x, y, width, height)`.

On crée ici une première vue qui va remplir l'ensemble de l'écran et dans laquelle on ajoutera d'autres objets.

```
UIView *helloWorldView = [[UIView alloc]
    initWithFrame:CGRectMake(0, 0, 320, 460)];
```

ESSENTIEL Taille de l'écran

L'écran de l'iPhone en mode portrait fait 320 pixels de large et 480 pixels de haut. La barre de statut (qui affiche l'heure en haut de l'écran) fait 20 pixels de haut. Il reste donc une zone de 320 pixels de large par 460 pixels de haut pour afficher vos vues avec la barre de statut.

Nous verrons plus loin que la barre de statut peut être pivotée (pour passer en mode paysage), cachée ou rendue transparente, ce qui a bien sûr un impact sur la zone disponible pour l'application.

Il est possible de connaître l'espace disponible pour votre application en utilisant la méthode `applicationFrame` de `UIScreen`.

```
CGRect mainScreen =
    [[UIScreen mainScreen] applicationFrame];
```

Créer un label

Le label est créé de la même façon que la vue (c'est un descendant de `UIView`). On définit le texte et son alignement via des propriétés de `UILabel`.

```
UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(20, 50, 280, 20)];
label.text = @"hello, world";
label.textAlignment = NSTextAlignmentCenter;
```

Assembler la vue

Le label est ajouté à notre vue principale, puis celle-ci est ajoutée à la fenêtre.

```
[helloWorldView addSubview:label];
>window addSubview:helloWorldView];
```


Libérer la mémoire

Les objets `label` et `helloWorldView` ont été alloués par notre code. L'objet `label` sera retenu en mémoire par l'objet `helloWorldView` et l'objet `helloWorldView` sera retenu en mémoire par l'objet `window`. Nous pouvons donc les relâcher (décrémenter le compteur de références). Si vous oubliez de les relâcher, ils ne seront pas détruits lorsque la vue sera supprimée de la fenêtre, et l'application aura une fuite mémoire.

```
[label release];  
[helloWorldView release];
```

Lancer l'application dans iPhone

Lancer votre application dans votre iPhone ne demande presque aucun effort supplémentaire. Vous devez avoir au préalable créé un certificat développeur et installé le profil de provisionnement associé sur votre iPhone (voir les instructions dans le premier chapitre).

Définir les paramètres de signature de l'application

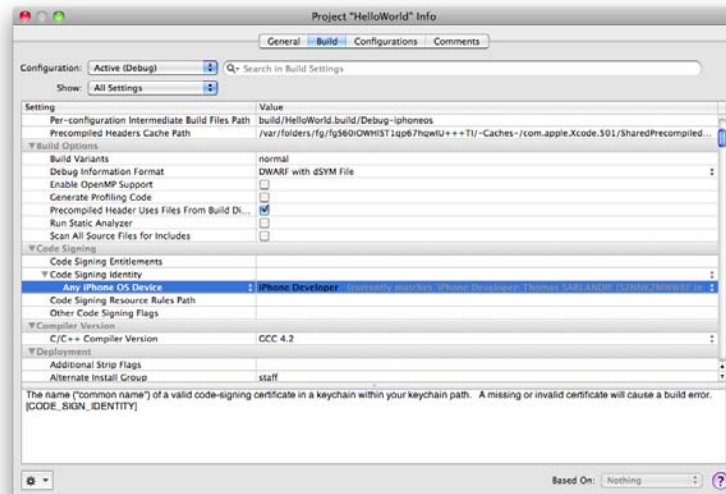
La première étape consiste à indiquer à Xcode avec quelle clé doit être signée l'application. Pour cela, ouvrez les paramètres de compilation du projet, en double-cliquant sur le nom du projet (c'est le nœud racine de l'arbre) dans la partie gauche de la fenêtre Xcode (ou *Project-Edit Project Settings*).

Sélectionnez le deuxième onglet *Build* et assurez-vous que la configuration sélectionnée est la configuration « *Debug* ».

Dans la section *Code Signing*, la ligne « *Any iPhone OS Device* » doit contenir par défaut « *iPhone Developer* ». Si vous ne touchez à rien, Xcode cherchera un certificat dont le nom commence par « *iPhone Developer* » ce qui devrait lui permettre de trouver votre certificat automatiquement.

Vous pouvez aussi cliquer sur la ligne « *iPhone Developer* » et choisir une des clés parmi celles proposées par Xcode.

Figure 3-18
Configuration du certificat à
utiliser pour signer l'application



Compiler et lancer l'application sur iPhone

Une fois ces réglages effectués, il vous suffit d'utiliser le bouton en haut à gauche de votre fenêtre Xcode pour passer du simulateur (Simulator) à votre iPhone (Device).

En cliquant sur le bouton *Build and Go*, l'application est compilée pour iPhone, installée sur votre iPhone (qui doit être relié en USB) et démarrée. Il est possible d'afficher la console exactement comme avec le simulateur et d'utiliser tous les autres outils du SDK (le débogueur, les instruments pour évaluer les performances de l'application, etc.).

Et maintenant ?

Vous connaissez maintenant les bases élémentaires de la programmation pour iPhone. Dans ce chapitre, vous avez appris :

- comment créer un nouveau projet iPhone ;
- le nom et le rôle des fichiers dans un projet minimaliste ;
- comment utiliser la console pour imprimer des informations ;
- comment créer de nouvelles vues avec Interface Builder et les charger dans l'application ;
- comment créer à la main des vues et les charger.

Si ce n'est déjà fait, reprenez ce chapitre avec votre ordinateur pour reproduire les exemples et essayer d'aller au-delà.

Dans les chapitres suivants, nous nous éloignerons du code pour comprendre les règles ergonomiques du monde iPhone et proposer une méthode pour concevoir l'interface de votre application.

DEUXIÈME PARTIE

Conception et ergonomie

Cette deuxième partie s'adresse à toute l'équipe en charge de l'application, les développeurs, les experts métiers, les créatifs (graphistes).

La plupart des équipes sont habituées au développement d'applications web ou client lourd. Le **chapitre 4** présente ainsi le cycle habituel de développement d'un projet iPhone, ses spécificités, les différentes phases et le rôle de chaque intervenant. Il permet de partager un cadre méthodologique qui sera adapté au contexte de l'équipe en charge du projet et à ses habitudes.

Le **chapitre 5** donne les bases de l'ergonomie iPhone que toute l'équipe doit maîtriser. Les conventions iPhone, leur logique et leur utilisation au sein des applications standard sont détaillées car il est essentiel de bien les avoir comprises pour concevoir l'interface d'une nouvelle application.

C'est justement le sujet du **chapitre 6** qui propose une méthode permettant de partir d'une liste de fonctionnalités pour arriver à une interface utilisateur. Nous verrons comment exploiter les éléments standard pour créer une nouvelle application et comment éviter les erreurs classiques.

4

Méthode de développement d'un projet iPhone

Comme pour tout projet informatique, le succès d'un projet d'application iPhone dépend fortement de la méthode mise en place pour le réaliser ; cette méthode doit être adaptée aux spécificités de l'objet qu'est l'iPhone.

Après avoir rappelé les objectifs de la méthode, nous passerons en revue les différentes étapes permettant de produire une application : de l'idée à la publication.

Qu'est-ce qu'un projet d'application réussi ?

La méthode mise en place doit avoir deux objectifs principaux :

- 1 Satisfaire les utilisateurs de l'application ;
- 2 Maîtriser la durée du projet (et donc le coût du développement).

Satisfaire les utilisateurs

Pour satisfaire ses futurs utilisateurs, une application iPhone doit fournir un service clair et utile en situation de mobilité. L'interface doit être comprise rapidement et les fonctionnalités principales mises en avant. Enfin, l'application doit se lancer instan-

tanément, fonctionner sans ralentissement, ne pas quitter brutalement et ne doit pas avoir d'effet de bord sur le téléphone dans son ensemble (empêcher les autres applications de fonctionner normalement, vider la batterie trop rapidement, etc.).

Cet objectif n'est pas spécifique à l'iPhone, mais il est trop souvent négligé, et dans le cadre d'un projet iPhone il est essentiel de lui redonner toute son importance. Conception de l'interface, ergonomie, tests et optimisation seront donc des phases clés de la méthode.

Maîtriser le projet

Dans la majorité des environnements de développement (sociétés de services, équipe de développement en interne, freelance ou développeur individuel), on considère que la source de coût principale d'un projet informatique est le temps passé par les membres de l'équipe. On cherche donc à le minimiser tout en respectant le premier objectif : satisfaire les utilisateurs.

La maîtrise des coûts passe par la maîtrise des délais, et donc le respect d'une date de mise en ligne de l'application.

La méthode aura donc atteint ses objectifs si l'application fournit bien un service pertinent à ses utilisateurs et si elle a été développée avec le budget prévu en respectant les délais. Dit autrement, l'objectif doit être de maximiser la valeur apportée au client avec le budget alloué ; les amateurs de méthodologies agiles se reconnaîtront.

 Véronique Rota, *Gestion de projet – Vers les méthodes agiles*, Eyrolles 2009

Les étapes du projet

Identifier les fonctionnalités clés

Toutes les fonctionnalités d'un service n'ont pas la même importance, et on cherchera sur mobile à en sélectionner certaines pour les mettre en avant.

Cette première étape dans le projet est essentielle puisque l'interface et toute l'ergonomie de l'application sont orientées pour faciliter l'accès à cette sélection de fonctionnalités.

En partant des fonctionnalités existantes (comme dans le cas de l'adaptation d'un service à l'iPhone) ou de celles imaginées, on effectuera donc un tri par ordre d'importance.

EXEMPLE APPLE Ne pas adapter sur iPhone toutes les fonctionnalités : Photo versus iPhoto

Les applications fournies dans l'iPhone ne sont pas aussi riches que leurs équivalents Mac ou web, mais elles intègrent les fonctionnalités les plus importantes, celles que l'utilisateur attend en situation de mobilité.

L'application Photo par exemple intègre beaucoup moins de fonctions que iPhoto. Apple a jugé que la priorité devait être donnée à la consultation des photos et a orienté l'interface en ce sens : navigation dans une liste d'albums, mosaïque d'images, diaporamas. À l'opposé, les fonctionnalités d'édition des images ont toutes disparu ; si elles avaient été intégrées, l'interface serait beaucoup plus complexe et l'application nettement moins intuitive.

Dans le monde mobile encore plus qu'ailleurs, il faut toujours privilégier une interface très légère qui corresponde au besoin de 80 % des utilisateurs plutôt qu'une interface complexe qui tenterait de répondre à 100 % des besoins.

Les une, deux ou trois premières fonctionnalités méritent certainement une place de choix : l'utilisateur y aura accès dès le lancement. Elles occuperont un pourcentage significatif de l'écran.

Les autres fonctionnalités vont être revues lors de la conception de l'interface. Si on peut les intégrer sans complexifier l'interface, elles seront conservées ; sinon on les sacrifiera pour améliorer l'ergonomie.

CONSEIL La réflexion fonctionnelle précède celle sur l'interface

La réflexion doit toujours partir de la fonctionnalité qui est apportée à l'utilisateur plutôt que de l'idée qu'on peut se faire de comment cette fonctionnalité sera réalisée.

Ainsi, *permettre à l'utilisateur de naviguer très rapidement dans toute sa collection musicale* est une fonctionnalité de l'application iPod. Le résultat est la navigation par liste et en mode CoverFlow (le mode paysage de l'application iPod).

À l'inverse, un développeur qui insiste pour qu'une présentation de type CoverFlow soit intégrée dans l'application est parti de l'interface et doit se redemander quelle est la fonctionnalité voulue, et ensuite seulement quel est le meilleur moyen de la présenter à l'utilisateur.

Cet exercice est détaillé et illustré avec des exemples dans la première partie du chapitre 6 « Conception de l'interface graphique ».

Définition des principes ergonomiques

En s'appuyant sur les résultats de l'étape précédente, on pose les bases de l'interface de l'application.

Apple fournit plusieurs design patterns d'interface utilisateur :

- la navigation via une barre d'onglets (comme l'application Horloge) ;
- la navigation via des listes (comme l'application Mail ou Contacts) ;

- mais aussi le design pattern d'édition sous forme de table avec éléments groupés (comme pour les Contacts ou les Préférences).

Figure 4-1

Différents design patterns d'interface : onglets, barre de commande et édition sous forme de table



Bien sûr, l'équipe peut innover et proposer de nouveaux principes d'ergonomie, mais avec le risque de surprendre voire de dérouter les utilisateurs alors qu'une des forces de l'iPhone réside dans la cohérence des applications. En outre, le SDK facilite la vie des développeurs qui respectent les design patterns d'ergonomie iPhone en fournissant des composants techniques prêts à l'emploi.

Le chapitre suivant décrit l'ensemble des principes ergonomiques iPhone, et dans quels cas les utiliser. Le chapitre 6 montre comment piocher dans ces éléments et les combiner pour créer une application originale.

Story-boarding et spécifications

La phase de spécifications vise à produire une description exhaustive de l'application. Elle se divise en deux parties : la description de ce que l'utilisateur voit, qui est faite en général via des *story-boards*, et qui servira ensuite à préparer des maquettes graphiques ; et la description de ce que fait l'application suite aux demandes de l'utilisateur.

La phase de story-boarding est une étape très importante de la conception d'applications iPhone. Elle peut se faire entièrement avec des outils comme PowerPoint, ou simplement sur papier. En s'appuyant sur les grands principes ergonomiques décidés à l'étape précédente, elle décline chaque écran de l'application.

On s'attachera sur les story-boards :

- à montrer les informations affichées à l'utilisateur ;
- à lister les moyens d'action de l'utilisateur (bouton, zone cliquable, etc.) ;
- à définir vers quel écran est envoyé l'utilisateur suite à une action.

Là encore, on réutilisera des éléments standard : bouton, zone de texte, clavier de saisie, zone de recherche, etc. Ces éléments sont décrits dans le prochain chapitre.

MÉTHODE Mise en œuvre d'une méthode agile

Le développement d'applications iPhone n'est bien sûr pas incompatible avec les méthodes dites « agiles ».

Dans le cadre d'un projet agile, l'équipe devra chercher à conserver une phase de réflexion sur l'interface qui se traduit par un story-board. Cette réflexion peut être maintenue en parallèle des développements et mise à jour à chaque itération.

La spécification de l'application reprend et complète le story-board. Elle détaille les animations, fournit les règles de gestion de l'application, décrit les cas d'erreurs possibles et le comportement à adopter.

Dans le cas d'une application qui échange des données avec un service web, c'est dans cette phase qu'on décrira auprès de quel service et selon quel protocole échanger des informations. Toute l'équipe s'attachera alors à vérifier la cohérence entre les écrans d'interface, les règles de gestion et ces services afin de s'assurer que toutes les informations nécessaires à l'application sont bien disponibles en ligne.

Enfin, tout ce qui est exigé de l'application mais qui n'a pas encore été détaillé sera décrit :

- statistiques à collecter ;
- mise en place d'un cache des données, et règles associées (durée de validité, taille maximale, etc.) ;
- internationalisation de l'application (support multilingue) ;
- etc.

Intervention artistique

Sur la base des story-boards, et en s'aidant de l'existant (site web, logo, marque, etc.), un graphiste définit une charte graphique pour l'application.

On sélectionne quelques écrans structurants de l'application (par exemple : la page d'accueil, l'édition d'une information clé, le résultat d'une recherche) et à l'aide d'outils comme Illustrator ou Photoshop ou Gimp, on dessine les écrans de l'application.

Une fois que l'équipe est satisfaite du travail sur les premiers écrans, on décline ce travail pour tous les écrans de l'application.

Ces images réalisées au pixel près, permettent de valider certaines hypothèses du story-board : nombre d'éléments visibles à l'écran, taille du texte, etc.

ASTUCE Gagner du temps en réutilisant des bibliothèques de composants

Depuis le lancement de l'iPhone, de nombreux graphistes ont dû redessiner des interfaces d'applications iPhone et refaire les éléments standard d'Apple. Heureusement, ce travail fastidieux n'est plus à refaire et vous trouverez en ligne des bibliothèques d'éléments pour Photoshop.

► <http://www.teehanlax.com/blog/?p=447>

Elles servent aussi de référence au développeur lors de l'assemblage des écrans et permettent le découpage de tous les éléments graphiques spécifiques à l'application (pictos, boutons, etc.).

CONSEIL Testez votre design

Il est très facile de tester votre design en exportant les images et en les ajoutant dans la collection de photos de votre iPhone. Vous pourrez ainsi vous rendre compte de la taille du texte et des zones d'interaction sur l'écran.

Il est même envisageable d'assembler très rapidement ces maquettes (sans découper les différents éléments ni les dynamiser) dans un squelette d'application pour vérifier que la navigation entre les écrans est bien compréhensible et s'assurer que toute l'équipe a la même vision du produit final.

Développement de l'interface

Le développement de l'interface est une tâche qui peut prendre un temps important, surtout si vous avez décidé de mettre en place de nouveaux composants graphiques ou de nouvelles animations qui ne font pas partie du SDK iPhone.

Le développeur travaille avec le graphiste pour découper les écrans et assembler l'interface de l'application, sans intégrer la logique métier. Des données d'exemple sont utilisées pendant cette phase, et les différents boutons vont juste permettre de passer d'une vue à l'autre.

L'interface d'une application est l'élément qui doit avoir été le plus travaillé. En réalisant l'interface en premier, on valide les choix faits plus tôt dans le projet et on prend le temps de définir les ajustements nécessaires.

En effet, les maquettes graphiques ne permettent pas d'identifier tous les problèmes :

- Est-ce que tous les boutons sont assez gros pour être touchés facilement ?
- Est-ce que les animations permettent bien de mettre en valeur ce qui a changé dans l'interface ?
- Est-ce que la navigation est naturelle pour un utilisateur habitué de l'iPhone ?

Développement de l'application

C'est la phase la plus classique. Toutes les bonnes pratiques standard du développement logiciel s'appliquent bien sûr au développement d'applications iPhone.

L'utilisation d'un gestionnaire de source est nécessaire même si le développeur travaille seul : il permet de conserver en sécurité le code source de l'application, et de conserver un historique de toutes les modifications. Si plusieurs développeurs collaborent, il permet de synchroniser le code source tous les jours et de faciliter sa mise en commun.

MÉTHODE Développement piloté par les tests

La mise en place de développement piloté par les tests n'est pas encore complètement possible sur iPhone. Il n'existe pas à l'heure actuelle d'outils pour tester l'interface graphique. Il est par contre possible de réaliser des tests unitaires.

Vous trouverez plus d'informations sur la mise en place de tests unitaires dans le guide de développement iPhone :

- ▶ http://developer.apple.com/iphone/library/documentation/Xcode/Conceptual/iphone_development/135-Unit_Testing_Applications/unit_testing_applications.html

Le développeur doit s'attacher à livrer régulièrement des versions stables de l'application. Elles permettent au reste de l'équipe de faire des retours le plus tôt possible. Attention cependant à ne pas tomber dans l'excès inverse : une livraison tous les jours ne laisse pas suffisamment de temps à l'équipe pour tester, ni le temps au développeur d'intégrer les retours.

Tests et optimisation

Contrairement à une application web, l'application iPhone quitte les mains de l'équipe au moment où elle est transmise à l'App Store. Un bogue nécessitera une mise à jour qui devra être validée par Apple, puis exigera des utilisateurs qu'ils cliquent sur le bouton de mise à jour de l'application. De plus, le succès d'une application dans l'App Store repose beaucoup sur les commentaires des utilisateurs. Une première version boguée risque d'entraîner de nombreux commentaires négatifs qui enterreront l'application au fond du classement.

L'optimisation permet de s'assurer que les performances seront les meilleures possibles. Il existe plusieurs outils dans le SDK pour aider le développeur et de nombreux conseils sont prodigués sur Internet (depuis Apple, ou d'autres sources).

La phase de test est donc particulièrement importante et doit être menée de manière exhaustive.

La préparation d'un plan de test, basé sur les spécifications, et mené pendant les développements, permet de faire une liste de tous les cas qui devront être testés. Cette liste sert de support aux tests et permet de s'assurer que toute l'application a bien été testée.

Le fonctionnement normal de l'application, mais aussi tous les cas d'erreurs devront être testés. Les situations de test aussi :

- en Wi-fi, en 3G, en Edge ;
- sur un iPhone ou sur un iPod Touch ;
- avec différentes versions de firmware.

La distribution de l'application à un public de bêta-testeurs permettra également de récolter de premiers avis d'utilisateurs et de valider la stabilité de l'application dans différents environnements. Cette distribution peut se faire en mode Ad Hoc, décrit au chapitre 1 dans la section « Pré-requis pour la distribution d'une application ».

Publication

Enfin, l'application est transmise à l'App Store via l'outil iTunes Connect. Les éléments connexes comme le nom de l'application, sa description, son icône, les captures d'écran sont fournis au moment de la soumission.

La validation de l'application par Apple peut prendre plusieurs semaines et peut se traduire par un refus motivé par un courriel d'explication. Les refus peuvent être liés à un bogue jugé trop important, au non-respect des conditions de l'accord SDK, etc. Dans ce cas, il faut corriger le problème et soumettre à nouveau l'application.

Une fois l'application validée, l'équipe devra attendre les premiers retours d'utilisateurs avant de répéter à nouveau le cycle complet.

Conclusion

Dans ce chapitre, nous avons passé en revue les étapes qui permettent de transformer une idée de projet en application téléchargeable sur l'App Store. Il est essentiel de travailler en équipe et que chacun, avec son métier, prenne le temps de comprendre les spécificités de la plate-forme iPhone.

Dans le prochain chapitre, nous en rappellerons les principales règles d'ergonomie – il est essentiel de les connaître avant de se lancer dans la conception d'une application.

5

Principes ergonomiques et design patterns d'interface

Réfléchir à l'ergonomie d'une application, concevoir son interface en fonction du domaine fonctionnel plutôt que de subir des contraintes techniques est devenu essentiel dans la réalisation de projets informatiques.

Le succès d'iPhone est sans aucun doute très largement dû à son ergonomie et aux efforts faits par Apple pour offrir une plate-forme sur laquelle toutes les applications reprennent les mêmes règles ergonomiques et s'appuient sur les mêmes métaphores. C'est ce qui rend l'iPhone si homogène et agréable pour les utilisateurs.

Les développeurs (au sens large : toute l'équipe) doivent comprendre ces règles qui régissent l'utilisation d'un iPhone, comment les décliner pour créer une application originale et ce que le SDK propose pour les implémenter. Pour l'utilisateur, l'application sera alors intuitive et l'apprentissage très rapide ; pour le développeur, la réalisation sera facilitée et rendue plus sûre grâce à l'utilisation de composants existants.

Dans ce chapitre nous nous intéresserons aux principaux design patterns d'interfaces et aux métaphores utilisées dans l'iPhone. Nous verrons ensuite comment les combiner pour créer une interface utilisateur riche et simple à utiliser.

L'ergonomie dans l'univers de l'iPhone

L'ergonomie de l'iPhone s'appuie sur des métaphores qui, en reprenant des concepts ou objets de la vie quotidienne, facilitent la prise en main de l'interface par les utilisateurs.

Outre ces métaphores, Apple a également conçu et utilisé des design patterns d'interface qui permettent de définir les grandes lignes de l'interface d'une application.

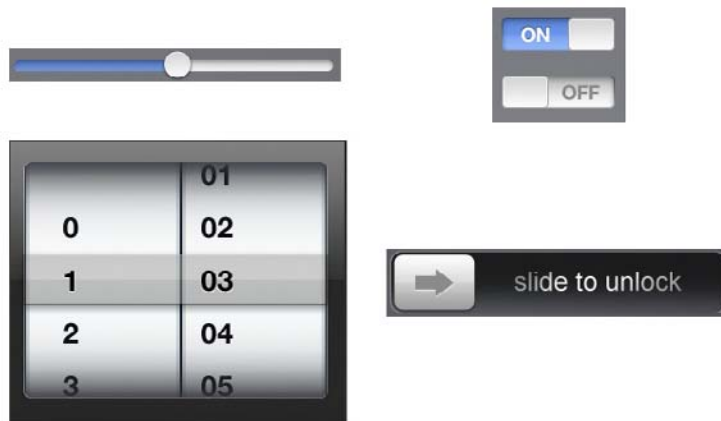
Les métaphores peuvent être utilisées pour proposer à l'utilisateur une fonctionnalité ou l'interaction avec une donnée de l'application ; les design patterns fournissent des solutions pour organiser l'application, ses différents écrans et la navigation entre eux.

Une interface différente basée sur des métaphores

Dès le premier contact avec l'iPhone, l'utilisateur comprend qu'il a entre les mains un terminal dont l'interface est très différente des autres terminaux qu'il a pu utiliser auparavant. Après plusieurs essais, il comprend comment déverrouiller l'écran et commence à utiliser les applications. Seul, ou guidé par un ami, il découvre l'interface, utilise son doigt pour faire défiler les contacts, les photos ou les pochettes d'albums, il comprend comment revenir à l'écran d'accueil, comment naviguer dans des listes, etc.

Pour faciliter la compréhension de l'interface par les utilisateurs, les métaphores sont très présentes : un bouton ON/OFF, une roue pour sélectionner une valeur dans une liste, un bouton à faire glisser, etc.

Figure 5-1
L'interface de l'iPhone reprend des éléments de la vie réelle.



APPROFONDIR Le guide ergonomique iPhone

Tous les développeurs (au sens large, c'est-à-dire toute l'équipe) devraient se pencher au moins une fois sur le guide ergonomique écrit par Apple : *iPhone : Human Interface Guidelines*. Il explique en détail l'ensemble des principes et des métaphores d'interfaces proposés par Apple. Il fournit aussi l'explication de nombreux choix. C'est une lecture passionnante pour toute personne qui s'intéresse à l'ergonomie de manière générale.

Ce guide est disponible sur le portail développeur iPhone.

Les design patterns d'interface

ESSENTIEL Qu'est-ce qu'un design pattern d'interface ?

Un design pattern d'interface (ou design pattern d'interaction) est une solution pour répondre aux problèmes classiques de présentation ou d'organisation de l'information dans une application ou un site web. Dans le cas de l'iPhone, ce sont principalement des solutions pour organiser la navigation dans les écrans de l'application.

L'iPhone introduit plusieurs design patterns d'interface. La navigation dans des listes de données hiérarchiques (qu'on retrouve dans l'application Téléphone pour naviguer dans les contacts) en est le meilleur exemple.

Les design patterns d'interface combinent plusieurs éléments :

- des composants graphiques ;
- une logique qui doit être facilement comprise par l'utilisateur ;
- des animations qui contribuent à expliquer la logique.

Nous détaillerons dans ce chapitre les principaux design patterns d'interfaces rencontrés sur iPhone.

Deux bonnes raisons pour s'aligner sur l'ergonomie de l'iPhone

Faciliter la prise en main de l'application

L'utilisateur d'un iPhone s'attend à retrouver dans toutes les applications qu'il installe les mêmes règles ergonomiques et à pouvoir réutiliser les comportements auxquels il s'est habitué.

Pour l'équipe en charge de la réalisation de l'application, le respect des principes ergonomiques iPhone permet surtout de faciliter l'apprentissage de l'application. Si elle est bien conçue, un utilisateur familier de l'iPhone comprendra rapidement comment l'utiliser. Il est d'ailleurs très rare que les applications soient fournies avec un mode d'emploi ou une aide en ligne intégrée.

REMARQUE Les jeux sont une exception

Les jeux obéissent à une logique très différente des applications. Les concepteurs veulent en général proposer une interface originale et qui plonge l'utilisateur dans leur univers (on parle d'application immersive). Les principes décrits dans ce chapitre ne concernent donc pas les jeux.

Accélérer les développements

Le SDK fournit des composants logiciels complets qui prennent en charge l'implémentation des principes ergonomiques de l'interface Apple.

En s'appuyant sur les principes ergonomiques classiques lors de la conception de l'application, on s'assure que le développement sera beaucoup plus rapide et plus sûr, puisqu'on pourra réutiliser du code fourni par Apple et déjà testé de manière intensive.

Développer des applications originales

Le respect de ces métaphores et principes, et l'utilisation de composants logiciels fournis par le SDK, ne doivent pas être un frein à l'originalité de l'application. Nous verrons dans ce chapitre comment combiner les composants, puis dans les suivants comment adapter leur apparence pour votre application.

Il existe de nombreuses applications qui ont su utiliser de manière pertinente l'existant tout en innovant quand cela était nécessaire.

EN RÉSUMÉ

Tous les principes décrits dans cette section doivent être évalués en fonction des objectifs fonctionnels, choisis et adaptés au contexte de l'application. Ce sont des atouts qui permettent d'améliorer les chances de succès de l'équipe.

Une application réussie est un équilibre subtil entre respect des règles ergonomiques et originalité.

Des applications pour consulter et manipuler des données

Le sujet principal de ce livre est le développement d'applications permettant de consulter et de manipuler des données.

On distingue les applications utilitaires et les applications de productivité. Les premières sont exclusivement orientées vers la consultation de données. C'est le cas par exemple des applications Météo et Bourse qui reprennent toutes les deux le même design pattern d'interface.

Les secondes permettent également la manipulation et la modification des informations. Elles ont une interface beaucoup plus complexe, c'est le cas par exemple de l'application Mail ou Facebook.

CONSEIL Ne négligez pas les applications utilitaires

Les applications utilitaires peuvent sembler très simples, voire même trop simples pour en faire des applications intéressantes que l'utilisateur téléchargera.

Pourtant, dans de nombreuses situations, elles peuvent répondre parfaitement au besoin de l'utilisateur avec une simplicité qui est entièrement à l'honneur de l'équipe de développement.

Les métaphores de l'interface iPhone

Les métaphores d'interface permettent de représenter des données et de proposer à l'utilisateur d'interagir avec elles.

Les plus importantes sont omniprésentes dans les applications iPhone et sont détaillées ci-après. Il en existe d'autres qui sont décrites dans le guide ergonomique iPhone.

Les listes d'éléments

Les vues en liste sont omniprésentes. Elles sont un élément clé du design pattern de navigation dans une liste hiérarchique que nous détaillerons plus loin.

Les listes peuvent se décliner sous plusieurs formes pour répondre aux différents besoins des applications, mais elles reprennent toujours quelques principes communs.

Comportements communs aux listes

La navigation dans les listes d'éléments se fait toujours verticalement, et on retrouvera toujours l'effet de rebond quand on arrive à une extrémité de la liste. Lors du lancement de l'iPhone, cet effet avait fortement impressionné ; il est aujourd'hui repris par toutes les interfaces récentes. L'utilisateur étant habitué à rencontrer des listes sur iPhone, il essaiera naturellement de « faire glisser » des objets placés à l'écran, mais il existe plusieurs solutions pour indiquer de manière explicite qu'il peut faire glisser les éléments.

La première solution consiste à faire en sorte que le dernier élément en bas de la liste soit coupé au milieu. L'utilisateur comprend ainsi que la vue actuelle est une fenêtre sur une liste plus longue et essaiera de la faire glisser.

La deuxième solution consiste à faire apparaître pendant quelques instants l'indicateur de position verticale sur le côté. L'utilisateur voyant cet indicateur, il comprend

qu'il est face à une liste et qu'il peut la faire défiler. Ce comportement est adopté par les applications standard de l'iPhone et est très discret. Nous verrons qu'il est également très simple à implémenter dans vos applications.

Utilisation de sections dans une liste

Vous pouvez grouper les éléments d'une liste par sections. Le ou les critères de regroupement sont de la responsabilité du développeur.

Les sections peuvent avoir un en-tête et un footer.

Figure 5-2
Une liste avec différentes sections (vue liste du calendrier)



Ajout d'un index sur une liste

Vous pouvez ajouter un index sur une liste pour permettre à l'utilisateur un accès rapide à un élément de la liste.

L'application iPod ou Contact propose ainsi des index alphabétiques pour naviguer dans la liste des éléments, mais vous pouvez proposer des index de n'importe quel type (seulement quelques lettres, des chiffres, etc.).

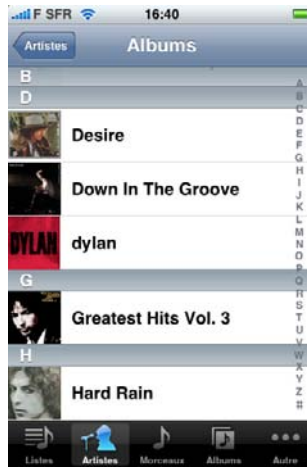
Accessoires des éléments et indicateurs de détails

Chaque cellule d'une liste peut avoir un accessoire qui est un élément d'interface placé à droite de la cellule.

Ce peut être une flèche grise indiquant à l'utilisateur qu'il peut cliquer sur la cellule, un bouton de type On/Off comme on en retrouve beaucoup dans les préférences ou bien n'importe quel autre élément d'interface.

Figure 5-3

Liste indexée dans l'application iPod ; on remarque aussi les sections.



Le cas particulier des indicateurs de détails est intéressant car très fréquemment utilisé avec les listes. Un indicateur de détails est représenté par une petite flèche blanche dans un rond bleu. En cliquant dessus, l'utilisateur déclenche un comportement spécifique qui n'est pas le même que celui obtenu en cliquant sur le reste de la cellule.

Le meilleur exemple de son utilisation est dans l'onglet *Appels* de l'application Téléphone : en sélectionnant un nom dans la liste, l'utilisateur va rappeler ce contact, alors qu'en touchant l'indicateur de détails (la flèche dans un rond bleu à droite), il en ouvre la fiche.

Figure 5-4

Utilisation d'indicateurs de détails dans l'application Téléphone



L'indicateur de détails vous permet donc d'enrichir la navigation dans l'application en proposant deux actions sur un seul élément de la liste.

Le mode édition

Les listes iPhone proposent un moyen standard de réorganiser, supprimer ou modifier leurs éléments. Les listes peuvent passer en mode édition. Dans ce cas, des contrôles supplémentaires peuvent être affichés à gauche des cellules (le contenu de la cellule est automatiquement décalé vers la droite). Au choix du développeur, ils peuvent permettre de supprimer la ligne, d'insérer un contenu après la ligne, ou de montrer qu'un élément est sélectionné (case à cocher). Vous pouvez tester ce mécanisme dans l'application Mail, par exemple.

En mode édition, il est également possible de faire apparaître sur chaque ligne un bouton permettant de réordonner les éléments de la liste (dans les signets de Safari, par exemple).

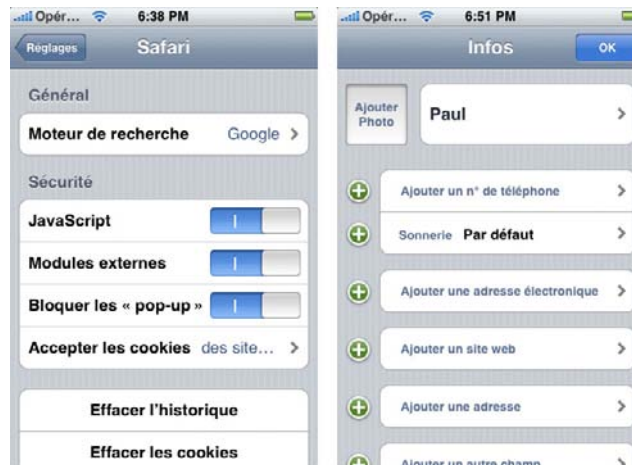
Les listes groupées

Les vues en liste groupées permettent de bénéficier de l'ergonomie liste (et en particulier de la faculté de voir très rapidement tous les choix possibles et d'en sélectionner un avec le doigt) pour représenter des données ou des actions différentes.

C'est le type de vue qui est utilisé pour afficher et modifier le détail d'un contact ou pour les préférences. Techniquement, il s'agit toujours de liste et les possibilités offertes par les listes classiques sont toujours disponibles, à l'exception de l'index qui ne peut pas être utilisé avec des listes groupées.

Chaque bloc d'éléments correspond à une section de la liste et le développeur peut très facilement définir l'apparence de chaque élément et de chaque titre de section.

Figure 5-5
Deux exemples de listes groupées



Édition d'une liste groupée

Tout comme les listes non groupées, les listes groupées peuvent passer en mode édition, et le développeur peut alors choisir de faire apparaître un élément d'interface supplémentaire à gauche de chaque élément.

Les design patterns d'interface iPhone

Les design patterns d'interface iPhone proposent des solutions pour organiser la navigation dans l'application.

Nous allons en détailler les principaux. La plupart peuvent être combinés entre eux.

Navigation dans une application utilitaire

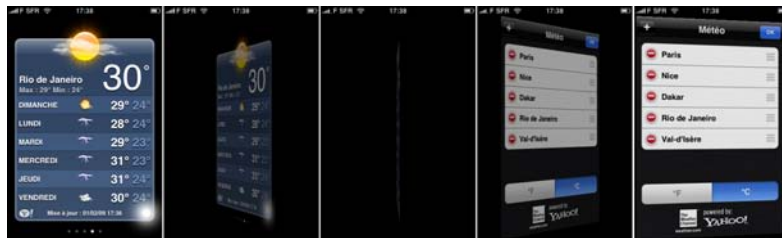
Les applications utilitaires Météo et Bourse s'appuient sur le même design pattern.

La vue principale de l'application permet de consulter des informations, d'accéder aux fonctionnalités principales. On peut passer d'un contenu à un autre en faisant défiler avec le doigt les différents écrans horizontalement.

On trouve en bas à droite, un bouton « i » qui permet de faire basculer la vue (une animation donne réellement l'impression que l'écran fait une rotation autour de son axe vertical) vers une deuxième vue.

La vue « de dos » permet de paramétrer les données à afficher dans l'application, elle n'est pas aussi souvent utilisée que la vue principale.

Figure 5-6
Animation des applications
utilitaires



Utilisation d'une barre d'outils

Les applications peuvent proposer une barre d'outils pour permettre à l'utilisateur d'accéder rapidement à un nombre important de fonctions. C'est le cas par exemple dans Safari (la barre d'outils permet de revenir en arrière ou d'aller en avant dans l'historique, de créer un favori, d'accéder aux autres fenêtres) ou dans Mail.

Figure 5-7
La barre d'outils dans
l'application Safari



La barre d'outils intègre plusieurs boutons. Chaque bouton doit déclencher une action immédiate et vous ne devriez pas utiliser la barre d'outils pour naviguer dans les vues de l'application.

Figure 5-8
La barre d'outils de l'application Mail



Navigation dans des listes hiérarchiques

La mécanique de navigation dans des listes hiérarchiques d'éléments est le premier design pattern que découvrent les utilisateurs : naviguer avec le doigt dans la liste de contacts, sélectionner un nom, accéder au détail.

Figure 5-9
Une liste hiérarchique
d'éléments

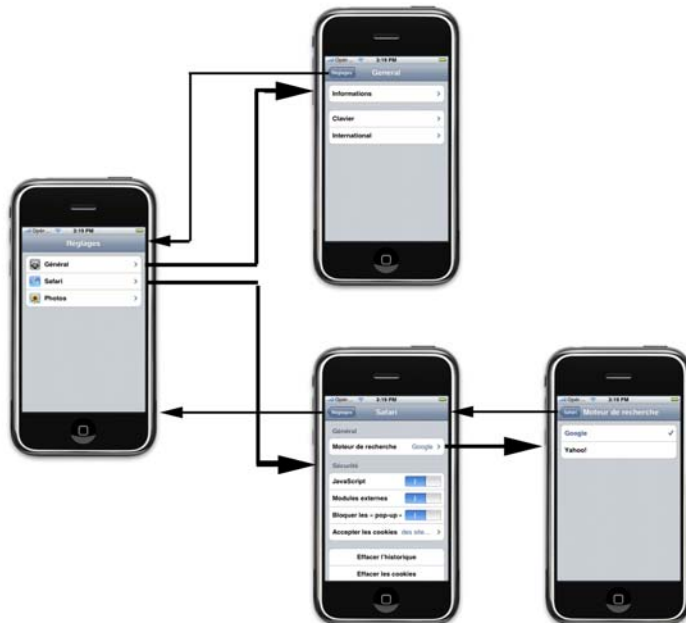


Une liste hiérarchique d'éléments est un arbre dont les nœuds sont des vues (c'est-à-dire des écrans de l'application).

À un instant donné, l'utilisateur ne peut voir que le nœud courant et il peut remonter au nœud père ou descendre vers les fils, qui sont le plus souvent représentés par une liste.

Le nœud racine de la liste est toujours le plus à gauche, on descend dans l'arbre en allant vers la droite et on remonte vers la gauche. Les animations permettent d'expliquer ce mouvement.

Figure 5-10
Navigation dans une liste hiérarchique



Ce design pattern repose sur plusieurs éléments :

- 1 une barre de navigation en haut de l'écran ;
- 2 une vue sous la barre de navigation qui présente la vue courante ;
- 3 les animations qui permettent de passer d'une vue à l'autre.

La barre de navigation

La barre de navigation est un composant divisé en trois parties distinctes :

- le bouton en haut à gauche qui permet de revenir à la vue parente ;
- le titre décrivant la vue actuelle ;
- et à droite un bouton contextuel (pour Ajouter ou Modifier un élément par exemple).

Quand il n'y a pas de nœud parent, le bouton de retour n'est pas affiché. Sinon, il contient le titre du nœud parent.

La vue de contenu

La vue de contenu est le nœud que l'utilisateur est en train de consulter. Si le nœud a des fils, ils sont souvent représentés par une nouvelle liste d'éléments que l'utilisateur peut sélectionner.

La vue utilise très souvent une liste pour représenter ses données, mais ce n'est pas une obligation. Nous verrons plus loin quels sont les différents types de listes.

Les animations

Quand l'utilisateur sélectionne le fils de l'élément courant, une animation est déclenchée, qui fait sortir la vue de contenu courante par la gauche de l'écran et fait entrer la nouvelle vue par la droite.

Dans la barre de navigation, le titre se déplace vers la gauche et semble se transformer en bouton retour (en fait, il est déplacé vers la gauche et disparaît progressivement alors que le bouton apparaît).

Cette animation est très importante : elle permet à l'utilisateur de comprendre que la vue qu'il vient de quitter est toujours accessible. Rapidement, il comprend qu'en sélectionnant des nœuds, il avance vers la droite, et qu'il revient en arrière vers la page de départ en cliquant sur les boutons de retour.

Principes de navigation à respecter

Pour ne pas dérouter les utilisateurs, il est important de respecter les principes de la navigation dans une liste hiérarchique :

- L'utilisateur doit toujours pouvoir revenir en arrière (sauf sur le nœud racine bien sûr).
- L'emplacement en haut à gauche est réservé au bouton retour.
- Les animations permettent de montrer à l'utilisateur la conséquence de son action sur un élément de la liste.

La classe `UINavigationController` prend en charge toute la logique de fonctionnement des listes hiérarchiques :

- afficher la vue courante ;
- mettre à jour les informations de la barre de navigation ;
- animer les transitions.

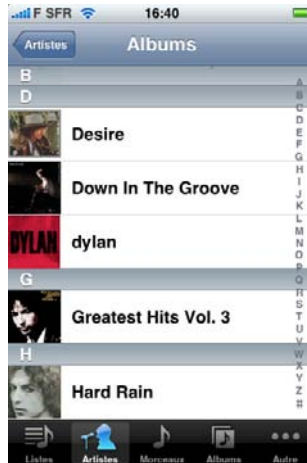
Nous verrons comment l'utiliser dans le chapitre 8 « Assembler les écrans de l'application ».

Navigation par onglet

L'utilisation d'une barre d'onglets permet au développeur de mettre au même niveau plusieurs fonctionnalités d'importance équivalente.

Le principe de fonctionnement est bien connu des utilisateurs puisqu'on le retrouve dans les applications Téléphone, iPod, iTunes, App Store et Horloge.

Figure 5-11
La barre d'onglets dans iPod



Pour que son fonctionnement reste clair pour l'utilisateur, il est essentiel que la barre d'onglets respecte quelques principes fondamentaux :

- L'application ne doit jamais forcer un changement d'onglet ; c'est l'utilisateur qui fait passer l'application d'un onglet à l'autre.
- Quand un utilisateur change d'onglet et revient à l'onglet précédent, il doit retrouver la même vue.

POINT D'ATTENTION De la bonne utilisation des onglets

Les onglets ne doivent surtout pas être utilisés comme barre de commande pour déclencher des actions : ils servent simplement à passer d'une vue à l'autre.

La barre d'onglets est conçue pour être l'élément de plus haut niveau dans une application. Quand elle est utilisée, elle doit apparaître au lancement et rester visible.

Utilisation en combinaison avec d'autres design patterns d'interfaces

La barre d'onglets est très souvent utilisée en combinaison avec des listes hiérarchiques. Dans ce cas, une liste hiérarchique est contenue dans un onglet. C'est le cas de l'application iPod dans lequel chaque onglet contient une liste hiérarchique différente.

Vous ne devriez jamais intégrer une barre d'onglets comme élément dans une liste hiérarchique.

REMARQUE Masquer la barre d'onglets

Il peut arriver qu'on souhaite masquer la barre d'onglets dans certains cas. C'est acceptable quand on combine la barre d'onglets avec une liste hiérarchique et qu'une vue veut afficher un contenu sur toute la surface de l'écran (diaporamas de photos par exemple), l'utilisateur doit pouvoir facilement revenir en arrière et la barre doit alors réapparaître.

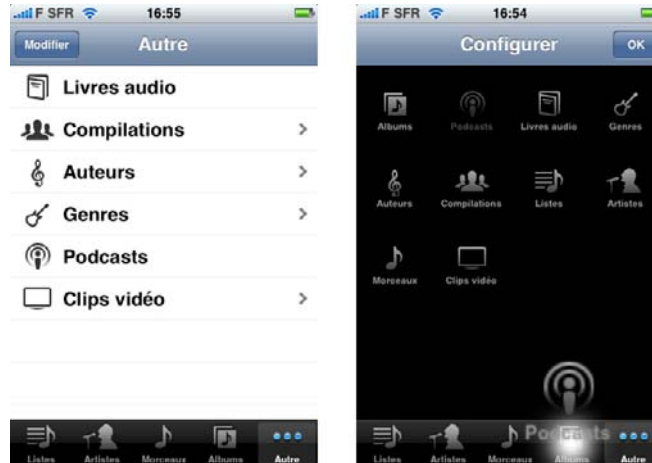
Personnalisation de la barre d'onglets

La personnalisation de la barre d'onglets est une fonctionnalité très intéressante utilisée dans iPod et qui peut être très facilement intégrée dans vos applications.

La barre d'onglets affiche toujours au maximum cinq icônes (même en mode paysage). Si le développeur ajoute plus d'onglets, les quatre premiers sont affichés et le cinquième est remplacé par un bouton « Autre ». En sélectionnant ce cinquième onglet, l'utilisateur accède à une liste des onglets supplémentaires et peut les consulter directement.

Figure 5-12

Vue Autre et Configurer de la barre d'onglets : il suffit de faire glisser les icônes sur la barre.



Il peut aussi utiliser le bouton « Modifier » pour configurer les onglets qui sont affichés par défaut : il suffit de sélectionner une icône dans la liste et de la glisser sur la barre d'onglets.

Passons à la conception

Dans ce chapitre, nous avons vu quels sont les principes ergonomiques les plus importants sur iPhone. Observez les applications installées sur votre iPhone, essayez de reconnaître les design patterns d'interface, comment ils sont combinés et adaptés.

Dans le chapitre suivant, nous verrons comment concevoir l'interface de l'application en partant des fonctionnalités les plus importantes et en réutilisant les design patterns d'interface.

6

Conception de l'interface graphique

Rappelons que vos applications doivent satisfaire des utilisateurs mobiles particulièrement exigeants. Dans ce chapitre, nous commencerons par étudier les contraintes qu'impose la situation de mobilité, puis nous proposerons une méthode qui permet d'organiser la conception et fournirons quelques outils pour vous aider à faire les bons choix.

Utilisation d'une application mobile

Le schéma classique d'utilisation d'une application mobile est très différent de celui de l'utilisation d'une application sur un ordinateur ou un site web.

Lorsqu'on conçoit une interface mobile, il faut prendre en compte ces spécificités et quitter les habitudes acquises dans d'autres domaines (applications Windows/Linux, sites web, etc.).

Temps et fréquence d'utilisation

Le temps moyen d'utilisation d'une application mobile est beaucoup plus réduit que celui d'une application sur Mac ou PC. Par contre, une application mobile est lancée plus souvent.

Prenez le temps d'étudier votre propre comportement : combien d'applications avez-vous lancées aujourd'hui ? Combien de fois chacune ? Combien de temps à chaque fois ? Vous constaterez que dans la plupart des cas, les applications sont lancées pour moins d'une minute et que les quelques applications les plus utilisées de votre iPhone peuvent être lancées plusieurs fois par jour.

L'application étant lancée très souvent, mais pour un temps plus court, il faut absolument qu'elle démarre rapidement.

BEST PRACTICE L'application doit se lancer rapidement

Les applications qui commencent par faire patienter l'utilisateur pendant plusieurs secondes avant de se charger ne seront pas utilisées souvent et disparaîtront des iPhone de vos utilisateurs.

Votre application devra toujours s'afficher et devenir réactive le plus vite possible. Les animations de lancement sont à proscrire : jouez-les uniquement au premier démarrage, et faites en sorte qu'elles ne bloquent pas l'utilisateur à chaque fois.

D'autre part, les applications iPhone peuvent être interrompues (appel entrant, SMS, etc.) et l'utilisateur peut donc être obligé de quitter l'application alors qu'il n'a pas fini ce qu'il faisait.

BEST PRACTICE L'application doit gérer les interruptions

Quand c'est pertinent, l'application devra enregistrer l'état actuel et permettre à l'utilisateur de reprendre exactement là où il en était. C'est valable pour l'édition ou l'ajout d'informations, mais aussi pour la consultation.

Concentration et attention disponible

En situation de mobilité, l'utilisateur fera beaucoup moins d'efforts pour comprendre l'interface que face à son ordinateur. Sa concentration n'est pas la même et des efforts qui semblent normaux sur un ordinateur ne seront que très rarement faits face à un terminal mobile.

Sur mobile, et encore plus sur l'iPhone, la simplicité de l'interface est un atout majeur et doit être un objectif lors de la conception. Si l'interface est simple, l'application sera facile à prendre en main, rapidement adoptée par les utilisateurs et surtout beaucoup plus utilisée.

Bien entendu, pour le développeur, simplicité n'est pas synonyme de facilité, bien au contraire ; il est bien plus difficile de concevoir une interface simple, surtout lorsque les fonctionnalités sont complexes.

CONSEIL L'interface doit rester simple, même pour des fonctionnalités riches

Il arrivera parfois qu'il semble impossible de proposer toutes les fonctionnalités dans une interface simple et claire. Le problème peut sembler insurmontable et toutes les solutions envisagées aboutissent à des interfaces trop complexes.

Dans ce cas, il faudra revoir le cahier des charges et faire un tri dans la liste des fonctionnalités : ont-elles toutes une réelle utilité en situation de mobilité ? Quel est leur niveau d'importance ?

Méthode pour concevoir l'interface d'une application

L'expérience acquise lors de la conception des applications iPhone, mais aussi de tous les services mobiles en général, nous permet d'identifier plusieurs phases clés qui doivent être exécutées une première fois, et répétées si nécessaire jusqu'à obtenir un résultat satisfaisant.

- 1 Identifier les fonctionnalités.
- 2 Trier les fonctionnalités par importance.
- 3 Concevoir l'interface en intégrant les fonctionnalités.

Identifier les fonctionnalités

Vous avez en tête un projet d'application. Dans la plupart des cas, plusieurs personnes ont imaginé l'application et pensé à ses fonctionnalités. La première tâche consiste à toutes les répertorier. Ce peut être le résultat d'un *brainstorming* avec les différents intervenants du projet.

Cette phase devrait aussi remettre au premier plan quelques questions élémentaires :

- Qui sont les utilisateurs de l'application ?
- Pourquoi lancent-ils l'application ?
- Dans quelles situations vont-ils utiliser l'application ?
- Quelle est la promesse client de l'application ?

Dans la mesure du possible, les fonctionnalités devront être décrites sous la forme de cas d'utilisation, c'est-à-dire d'une petite histoire racontant le contexte pour l'utilisateur, ce qu'il cherche et ce que l'application lui fournit.

Encore une fois, il faudra être très vigilant durant cette phase à ne pas penser en terme d'éléments d'interface, ce qui risquerait de fermer des portes à la créativité de l'équipe.

Trier les fonctionnalités par ordre d'importance

À l'issue de cette première phase, vous aurez une liste probablement assez longue de fonctionnalités. Le but de la deuxième phase est de classer les fonctionnalités par ordre d'importance.

Les trois groupes de fonctionnalités

Une application mobile doit trouver le bon compromis entre ergonomie et richesse fonctionnelle. La répartition des fonctionnalités en trois groupes est un outil pour aider l'équipe de conception à faire des choix.

Groupe 1 : La ou les fonctionnalités essentielles

Il ne devrait y en avoir qu'une ou deux. C'est pour elles que l'utilisateur lancera l'application dans 80 % des cas ou plus.

Ces fonctionnalités devraient être accessibles directement au lancement de l'application, si possible sans un seul clic de l'utilisateur.

Groupe 2 : Les fonctionnalités importantes

Ces fonctionnalités sont toutes d'une importance équivalente, c'est-à-dire qu'elles concernent tous les utilisateurs et seront utilisées régulièrement.

Elles seront mises en avant dans l'interface et l'utilisateur comprendra très vite leur rôle et comment les utiliser.

Groupe 3 : Les autres fonctionnalités

Les autres fonctionnalités sont annexes. Elles ne concernent pas tous les utilisateurs et seront utilisées moins d'une fois sur dix lancements de l'application.

Elles n'ont pas besoin d'être mises en avant dans l'interface. Éventuellement, l'utilisateur les découvrira par hasard. Elles sont facultatives et pourront être supprimées si on ne trouve pas de moyen élégant de les intégrer à l'interface.

De l'importance des trois groupes

L'expérience montre qu'il est toujours possible de répartir les fonctionnalités dans les trois groupes décrits précédemment et que lorsqu'on n'y parvient pas, c'est généralement parce qu'on essaie de concevoir une application qui sera trop compliquée et dont l'interface ne sera pas satisfaisante.

Si l'exercice semble impossible à réaliser, vous pouvez envisager de séparer l'application en plusieurs applications différentes.

CONSEIL La dure sélection des fonctionnalités

En photographie, on parle « d'editing » : c'est une phase durant laquelle le photographe choisit, parmi toutes les photos d'un reportage, celles qu'il va garder pour en extraire les meilleures et construire une série cohérente. Certaines de ses très belles photos ne seront pas retenues parce qu'elles ne s'intègrent pas bien avec le reste de la série. Le photographe apprend à en faire le deuil.

Le développeur d'applications iPhone devra souvent faire de même avec certaines fonctionnalités...

Concevoir l'interface pour ses fonctionnalités

Ayant identifié et réparti les fonctionnalités en trois groupes, on s'appuie sur les différents design patterns d'interface vus au chapitre précédent pour définir l'interface dans ses grandes lignes.

Les fonctionnalités du premier groupe doivent être accessibles en un temps minimum

Ces fonctionnalités sont celles qui vont pousser l'utilisateur à cliquer sur l'icône de votre application dans plus de 80 % des cas. Vous devez lui proposer un accès immédiat à ces fonctionnalités, c'est-à-dire idéalement sans aucune intervention de sa part, ou au maximum avec un clic.

Si votre application n'a qu'une fonctionnalité du groupe 1 et peu d'autres fonctionnalités, le design pattern des applications utilitaires (Météo, Bourse) est une solution très efficace.

EXEMPLE L'application Facebook

L'application Facebook met immédiatement en avant le Mur de l'utilisateur. Celui-ci est affiché dès le lancement de l'application avec les informations gardées en mémoire et se met à jour automatiquement sans intervention de l'utilisateur.



Figure 6-1
Ouverture de l'application Facebook

EXEMPLE Chargement de Omnifocus

Omnifocus est une application iPhone qui implémente la méthode « Getting Things Done » inventée par David Allen. C'est un assistant personnel à la collecte et à l'organisation de tâches.

Le temps de chargement peut être assez long pour les utilisateurs qui ont de nombreuses tâches et pourtant l'usage principal de l'application en situation de mobilité est l'ajout de tâches « je pense à quelque chose à faire ; je le note immédiatement ».

Les développeurs ont donc fait en sorte que pendant le chargement de la base de données, l'utilisateur puisse cliquer sur le bouton « Ajouter une tâche », ce qui permet d'accéder à la fonctionnalité numéro 1 très rapidement.

Mise en avant des fonctionnalités du deuxième groupe

Les fonctionnalités du deuxième groupe sont importantes et un des rôles de l'interface est de montrer à l'utilisateur qu'elles existent.

Le design pattern d'interface le plus souvent utilisé pour les applications ayant plusieurs fonctionnalités d'importance équivalente est l'interface avec barre d'onglets. Elle permet en effet de montrer rapidement à l'utilisateur quelles sont les fonctionnalités principales de l'application et d'y accéder très naturellement.

EXEMPLE Application iTunes

L'application iTunes permet de télécharger de la musique en partant de la sélection Apple, du classement des utilisateurs ou en faisant une recherche.

Toutes ces fonctionnalités sont placées à un niveau d'importance équivalent grâce à l'utilisation du design pattern navigation par onglets.

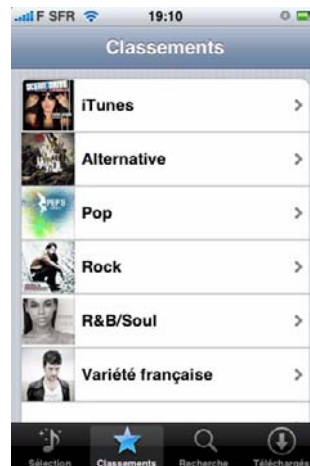


Figure 6-2

Navigation par onglets dans l'application iTunes

Fonctionnalités du troisième groupe

Les fonctionnalités du troisième groupe ne sont pas indispensables à l'application. De ce fait, il n'est pas essentiel de les faire apparaître dans l'interface, et on cherchera au contraire à ne pas les rendre trop visibles pour que l'interface reste simple.

Ce sont les fonctionnalités que l'utilisateur découvre par hasard alors qu'il a installé l'application depuis plusieurs semaines, et qu'il partage ensuite avec son entourage : « Tu as vu qu'on pouvait faire ça ? »

EXEMPLE Utilisation de l'accéléromètre dans les applications

Pour implémenter ces fonctionnalités dans l'interface, on pourra s'appuyer par exemple sur l'accéléromètre. Ainsi, l'application Facebook permet de recharger le contenu de la page principale en secouant le terminal, de nombreuses applications permettent de zoomer sur le contenu en basculant le téléphone en mode horizontal, etc.

Ce type de fonctionnalités est une part importante de l'expérience iPhone et l'utilisateur sera très satisfait de ces petites surprises. Vous souvenez-vous quand vous avez découvert l'utilisation de la loupe pour éditer du texte ?

Quelques outils pour concevoir l'interface

Il existe quelques principes faciles à appliquer et qui seront toujours utiles lors de la conception de l'interface.

Un accès unique à chaque fonctionnalité

Il semble évident que pour simplifier l'interface il faut réduire le nombre de fonctionnalités et le nombre de façons différentes d'accéder à une fonctionnalité. Pourtant, il arrivera souvent qu'on hésite sur le meilleur moyen de proposer une fonctionnalité et qu'on finisse par ne pas oser trancher et garder plusieurs moyens d'accéder à la même fonctionnalité.

Le succès de l'iPhone repose pourtant sur la pertinence des choix que l'équipe de conception a faits à la place des utilisateurs. Sans ces choix (qui sont autant de risques) l'interface ne serait pas aussi simple et élégante.

L'équipe en charge de l'application doit donc s'interroger sur le contexte d'utilisation, sur la cible, sur les expériences passées des utilisateurs et faire le choix qui sera le bon pour 80 % d'entre eux.

CONSEIL Faites des choix

Quand il s'agit de l'interface de votre application, ne pas trancher entre plusieurs solutions reviendra souvent à faire le mauvais choix.

Éviter les menus et autres listes de fonctionnalités

De nombreux projets d'applications commencent avec un écran d'accueil qui proposerait plusieurs boutons pour accéder aux différentes fonctionnalités de l'application.

C'est une fausse bonne idée qui ne correspond pas aux habitudes des utilisateurs et risque de devenir très rapidement frustrante. En effet, mieux vaut choisir une des fonctionnalités et la mettre en avant dès le lancement.

C'est en général le résultat d'un mauvais tri des fonctionnalités qui a abouti à ne mettre aucune fonctionnalité dans le premier groupe, ou à en mettre trop. Il convient alors de reprendre la liste des fonctionnalités et s'interroger à nouveau sur l'importance de chacune.

Distinguer le premier lancement des suivants

Un autre symptôme classique de la phase de conception est une interface qui devient très compliquée parce qu'on essaie d'expliquer à l'utilisateur le fonctionnement de l'application.

Il est légitime de vouloir accompagner l'utilisateur lors du premier lancement, mais cela ne doit pas nuire aux centaines de lancements suivants. Il est techniquement tout à fait possible de faire un cas particulier pour la première fois et de proposer un assistant, une vidéo de présentation ou un message.

EXEMPLE L'assistant de configuration de l'application Mail

Quand l'utilisateur lance l'application Mail, un assistant lui propose de paramétrer un nouveau compte s'il ne l'a pas encore fait.

Adapter l'interface à l'utilisateur

L'iPhone propose plusieurs mécanismes pour adapter l'interface à l'utilisateur.

C'est le cas par exemple de la barre d'onglets qui peut s'adapter à l'utilisateur de deux façons différentes.

Mémoriser le dernier écran utilisé

La barre d'onglets permet de mettre en avant des fonctionnalités d'importance équivalente. Il est possible de mémoriser à chaque utilisation la dernière fonctionnalité utilisée pour proposer à l'utilisateur de reprendre directement au même endroit.

EXEMPLE Application Horloge

L'application Horloge de l'iPhone mémorise le dernier onglet utilisé (*Horloges, Alarmes, Chronomètre, Minuteur*) et se rouvre sur le même onglet lors du prochain lancement.

Proposer à l'utilisateur de personnaliser la barre d'onglets

Le mécanisme de personnalisation de la barre d'onglets permet à l'utilisateur d'adapter l'interface en fonction de ses préférences.

Paramètres de l'application

Dans bien des applications, les possibilités de paramétrage ne seront jamais utilisées par la plupart des utilisateurs. Il s'agit typiquement d'une fonctionnalité du troisième groupe.

C'est pourquoi l'iPhone permet aux applications d'ajouter très facilement une entrée dans le menu Réglages du téléphone pour que les réglages soient faits en dehors de l'application. L'interface, n'étant pas encombrée par ces fonctionnalités, reste très simple, mais la fonctionnalité est proposée et sera trouvée par les utilisateurs expérimentés.

Conclusion

Ce chapitre clôt la deuxième partie de ce livre destinée à l'ensemble de l'équipe de développement. Dans les prochains chapitres, destinés aux développeurs, nous plongerons au cœur du développement de l'application.

TROISIÈME PARTIE

Le développement de l'interface

Nous verrons dans cette partie comment développer l'interface d'une application iPhone.

Tout d'abord, la brique principale de construction d'une application iPhone est le contrôleur de vue qui est expliqué en détail au **chapitre 7**.

On peut alors au **chapitre 8** se pencher sur l'assemblage des contrôleurs de vue pour construire une application complète, dont les écrans s'enchaînent de manière fluide.

Le **chapitre 9** présente les vues, c'est-à-dire tous les éléments d'interface, et donne les clés pour bien comprendre comment assembler un écran à partir des composants graphiques fournis par le SDK et comment les adapter aux couleurs de votre application.

Enfin, au **chapitre 10**, on présentera les tables, qui sont les éléments les plus utilisés dans la plupart des applications, et qui permettent de faire des listes et toutes sortes d'écrans dans lesquels l'utilisateur navigue verticalement avec le doigt.

Contrôler les écrans de l'application

On décompose facilement toute application iPhone en une série d'écrans. Cette décomposition se retrouve de manière très naturelle dans le code de l'application : pour chaque écran on implémente un contrôleur qui crée la vue, charge le modèle et ajoute un peu de logique métier.

Les contrôleurs de vue sont donc les briques élémentaires de toute application iPhone : chaque contrôleur gère un écran, et peut être chaîné avec d'autres pour mettre en place des applications complexes.

La notion de contrôleur de vue est issue du design pattern MVC qui est fortement ancré dans UIKit. Dans ce chapitre, nous rappellerons le fonctionnement de ce design pattern et nous décrirons comment utiliser la classe `UIViewController` pour créer les contrôleurs d'une application.

À la fin de ce chapitre, vous aurez en main tous les éléments pour construire le contrôleur d'un écran de l'application. Dans le chapitre suivant, nous verrons comment assembler les contrôleurs pour enchaîner les écrans, puis au chapitre 9 nous verrons comment construire des vues riches et animées.

Le modèle MVC dans iPhone OS

Le design pattern MVC est un des design patterns les plus connus dans le monde du développement logiciel. Il permet de clairement séparer les données (Modèle), leur représentation graphique à l'écran (Vue) et la logique métier (Contrôleur) de l'application.

Le diagramme de séquence suivant rappelle le principe général de fonctionnement d'une application MVC.

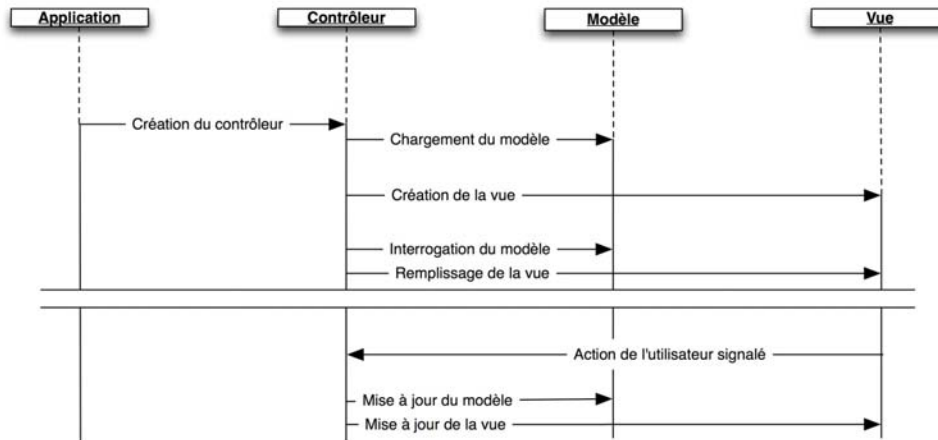


Figure 7-1 Diagramme de séquence simplifié d'une application MVC.

On retrouve dans iPhone OS les trois éléments classiques du modèle MVC.

Le modèle pour charger et stocker en mémoire les données de l'application

Toutes les applications manipulent des données métier, c'est-à-dire des informations qui représentent des objets ou des informations du monde réel – par exemple un contact, un article, un produit, etc.

Avant d'exister en mémoire, ces objets sont chargés depuis une source quelconque (fichier XML, base de données, etc.) et peuvent être éventuellement réexportés ensuite.

Le modèle représente les données que votre application manipule, ainsi que les classes permettant de les créer et de les sauver. Pour implémenter le modèle, vous pouvez créer vos propres classes ou utiliser les classes génériques du framework Foundation (`NSDictionary`, `NSArray`, etc.). Pour lire et enregistrer les données, vous utiliserez les API XML, le format de fichier `plist` ou bien le framework Core Data qui est disponible depuis la version 3.0 d'iPhone OS.

La vue pour représenter graphiquement le modèle et fournir l'interface graphique

La vue est l'interface de l'application, ce que l'utilisateur voit. C'est une représentation du modèle, il existe souvent plusieurs représentations des mêmes objets métier (une vue en liste, une vue détaillée, une vue d'édition, etc.).

La vue contient aussi des éléments qui permettent au développeur d'interagir avec l'application : des boutons, des images cliquables, des interrupteurs, etc.

On compose la vue en assemblant des objets dérivés de `UIView`, soit avec des lignes de code, soit en utilisant Interface Builder. Nous détaillerons la construction des vues dans le chapitre 9 « Développer et animer les vues ».

Le contrôleur pour lier le modèle et la vue

Lors du premier affichage, le contrôleur lance le chargement des données, et utilise les objets métiers pour remplir les éléments de la vue (mettre le nom du contact dans le champ texte prévu à cet effet par exemple). Le contrôleur est ensuite chargé de mettre à jour la vue lorsque les données changent, et de mettre à jour les données lorsque l'utilisateur interagit avec la vue.

Dans une application, il est possible d'assembler plusieurs contrôleurs de vue. Bien que ce ne soit pas une règle absolue, on écrira généralement un contrôleur pour chaque écran de l'application. Ainsi, dans une application qui utilise une barre d'onglets, on aura un contrôleur pour chaque onglet.

Les contrôleurs sont implémentés grâce à la classe `UIViewController` (ou une des classes qui en héritent).

Le contrôleur de vue standard d'iPhone OS

Lors du développement des premières applications de l'iPhone, les équipes d'Apple se sont aperçues qu'elles passaient un temps important à réimplémenter les mêmes fonctionnalités dans plusieurs applications. Ce constat les a poussées à mettre en place une classe prenant en charge les fonctionnalités de base d'un contrôleur de vue et à en faire profiter tous les développeurs d'application iPhone OS.

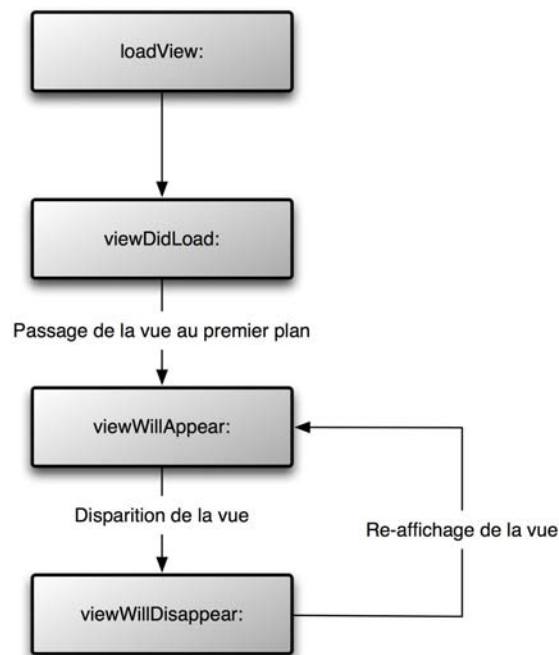
La classe `UIViewController` fournit donc une grande partie de la logique nécessaire à la gestion des vues. Elle est capable de charger la vue à partir d'un fichier NIB (Interface Builder), de retarder le chargement de la vue jusqu'à son utilisation, de libérer automatiquement de la mémoire si le système en réclame, de faire pivoter la vue

lorsque l'utilisateur bascule son téléphone en mode paysage, d'afficher des vues modales par-dessus la vue en cours, etc.

Cycle de vie d'un contrôleur de vue

On distingue trois états principaux dans lesquels le contrôleur de vue peut se trouver. Pour chaque transition entre états, le contrôleur est notifié et des méthodes spécifiques sont appelées.

Figure 7-2
Les différents états
d'un contrôleur de vue
et les transitions



Contrôleur initialisé sans vue

Les contrôleurs de vue peuvent être créés au lancement de l'application par le délégué d'application ou bien plus tard lors de l'exécution par un autre contrôleur de vue.

Le constructeur par défaut d'un contrôleur de vue est la méthode `initWithNibName:`. Vous devez la surcharger pour y ajouter votre code d'initialisation. Il peut préparer le modèle (lire des données sur le disque, lancer un chargement réseau, etc.) mais il ne crée pas la vue.

Vue chargée, non affichée

La vue n'est pas créée en même temps que le contrôleur, elle est instanciée lorsqu'on l'ajoute à une fenêtre ou à une vue parente. Il peut donc y avoir un décalage important entre le moment où le contrôleur est initialisé et le moment où la vue est chargée.

Par exemple, quand on utilise une barre d'onglets, tous les contrôleurs sont créés au lancement de l'application, mais les vues ne sont créées que lorsque l'utilisateur les sélectionne. Cela permet de charger l'application plus rapidement et limite la consommation mémoire.

Le chargement de la vue est toujours effectué par la méthode `loadView`. Cette méthode est appelée automatiquement lorsque la vue est utilisée pour la première fois.

Dans le cas où on utilise Interface Builder pour définir la vue, il ne faut pas surcharger la méthode `loadView`. L'implémentation par défaut charge les objets décrits dans le NIB dont le nom a été passé lors de l'initialisation du contrôleur de vue.

Pour créer la vue sans Interface Builder, il faut surcharger la méthode `loadView` et y insérer le code qui crée une hiérarchie de vue.

APPROFONDIR Comment fonctionne le chargement retardé de la vue ?

La classe `UIViewController` définit une propriété `view` qui est liée à un getter spécifique. Ainsi, à chaque fois que l'on fait appel à la propriété `view`, une méthode de l'objet est appelée, qui regarde si la vue a été créée (si elle est différente de `nil`) et qui appelle la méthode `loadView` s'il faut charger la vue. Une fois chargée en mémoire, la même vue est renvoyée à chaque fois qu'on appelle la propriété `view` (la méthode `loadView` n'est donc appelée que si la vue n'existe pas en mémoire).

Dans tous les cas, la méthode `viewDidLoad` est appelée après la méthode `loadView`. Elle est utile quand on a choisi de définir la vue avec un fichier NIB pour ajouter du code d'initialisation qui doit être exécuté à chaque fois que la vue est créée.

Vue chargée et affichée

Juste avant que la vue ne soit affichée, la méthode `viewWillAppear:` du contrôleur est appelée. Puis, dès que la vue est affichée, la méthode `viewDidAppear:` est appelée.

Lorsque la vue est affichée, elle envoie des événements au contrôleur pour signaler les actions de l'utilisateur. Le contrôleur réagit et met la vue et le modèle à jour en fonction des événements.

Quand la vue disparaît ou est masquée, la méthode `viewWillDisappear:` est appelée, puis la méthode `viewDidDisappear:`.

Avertissement de mémoire

Lorsque l'application reçoit un avertissement de mémoire, il est reçu par tous les contrôleurs de vue. Si la vue est chargée mais n'est pas affichée au moment où l'avertissement est reçu, elle est détruite pour libérer de la mémoire. Lorsqu'elle sera de nouveau utilisée, la vue sera recrée automatiquement (toujours par le même mécanisme et toujours en appelant `loadView` puis `viewDidLoad`).

La méthode `didReceiveMemoryWarning` est appelée pour signaler au contrôleur de vue qu'il doit libérer de la mémoire. Vous devez surcharger cette méthode pour libérer vos propres objets ; il faut absolument appeler l'implémentation par défaut pour que la vue soit également libérée si possible.

```
- (void)didReceiveMemoryWarning {
    // Libère la vue si elle n'est pas utilisée
    [super didReceiveMemoryWarning];
    // Ajouter ici la suppression des objets qui peuvent être supprimés
    (données en cache, etc.)
}
```

Si la vue a été détruite, la méthode `viewDidUnload` est également appelée. Dans le corps de cette méthode, vous devez libérer les références que vous gardez vers des éléments de la vue. En effet, si vous ne libérez pas ces éléments, ils persisteront en mémoire.

```
- (void)viewDidUnload {
    self.myButton = nil;
    self.myLabel = nil;
}
```

ESSENTIEL Prévoir le cas de la destruction automatique des vues

Gardez toujours à l'esprit que la vue peut être détruite automatiquement par le contrôleur puis recrée plus tard.

Il faut donc bien séparer le code d'initialisation du contrôleur qui est exécuté une seule fois, et le code d'initialisation de la vue qui sera réexécuté à chaque fois que la vue est rechargée. Il faut également écrire ce code de manière à ce qu'il puisse être appelé plusieurs fois, sans provoquer de fuite mémoire.

Bien qu'il soit possible d'empêcher la destruction automatique de la vue, il ne faut pas le faire car si le système n'arrive pas à libérer de mémoire, il terminera l'application.

Utilisation des contrôleurs de vue

Création d'un nouveau contrôleur de vue

Xcode permet de créer rapidement un nouveau contrôleur de vue : sélectionnez *New File* dans le menu *File* et dans la catégorie *Cocoa Touch Classes*, l'option *UIViewController subclass*.

Xcode génère un fichier d'en-tête et un fichier d'implémentation d'une nouvelle classe dérivant de `UIViewController`.

Instanciation d'un contrôleur de vue

L'initialisation du contrôleur de vue dépend de la méthode choisie pour décrire la vue.

Créer un contrôleur de vue sans fichier NIB

Pour indiquer au contrôleur de vue que l'on souhaite construire la vue manuellement, il suffit d'appeler la méthode `init`, qui ne prend aucun paramètre.

```
MyViewController *myViewController = [[MyViewController alloc] init];
```

Vous devez alors surcharger la méthode `loadView` pour implémenter la création de la vue.

```
- (void)loadView {
    UIView *myView = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320,
480)];
    UILabel *myLabel = [[UILabel alloc] initWithFrame:CGRectMake(20, 100,
280, 20)];
    myLabel.text = @"Hello World";
    myLabel.textAlignment = UITextAlignmentCenter;
    [myView addSubview:myLabel];

    self.view = myView;
    [myLabel release];
    [myView release];
}
```

Dans cet exemple, on crée une hiérarchie de vues très simple : une vue racine recouvrant tout l'écran à l'intérieur de laquelle on place un label dont le texte est centré. Nous détaillerons la manipulation des vues au chapitre 9.

Cette hiérarchie de vue est ensuite assignée à la propriété `view` du contrôleur de vue.

IMPORTANT Bien gérer la mémoire dans les contrôleurs de vue

Lorsqu'on assigne une nouvelle vue à la propriété `view`, son compteur de références est automatiquement incrémenté (en effet, la propriété est définie avec le mot-clé `retain`). Il est donc essentiel de bien décrémenter le compteur d'utilisation de la vue en appelant la méthode `release` pour éviter des fuites mémoires. Il faut faire très attention à ne pas avoir de fuite mémoire dans la méthode `loadView` car les objets manipulés peuvent rapidement prendre beaucoup de place en mémoire.

Créer un contrôleur de vue utilisant un fichier NIB

Pour utiliser un fichier NIB défini avec Interface Builder, on utilise le constructeur `initWithNibName:bundle:` et on passe le nom de ce fichier comme premier paramètre. Le deuxième paramètre à passer indique dans quel bundle chercher ce fichier, sauf quelques très rares exceptions, on indique toujours `nil` pour lui indiquer de chercher dans le bundle par défaut.

```
MyViewController *myViewController = [[MyViewController alloc]
initWithNibName:@"MyView" bundle:nil];
```

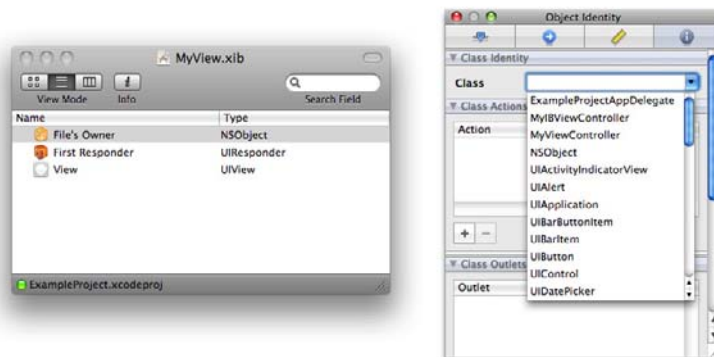
Lorsque vous utilisez un fichier NIB, vous ne devez pas surcharger la méthode `loadView`. Vous pouvez rajouter du code qui sera exécuté après chaque chargement de la vue dans la méthode `viewDidLoad:`.

Préparation du fichier XIB

Pour créer la vue d'un contrôleur avec Interface Builder, utilisez l'option *View XIB* dans la catégorie *User Interfaces* des modèles de nouveau fichier (*File > New File*).

Il faut indiquer à Interface Builder quel est l'objet qui chargera le fichier XIB. C'est le *File's Owner*. Pour cela, dans la fenêtre principale, sélectionnez la ligne *File's Owner* et dans le quatrième onglet de l'inspecteur de propriétés, choisissez votre contrôleur de vue qui doit apparaître dans la liste *Class*.

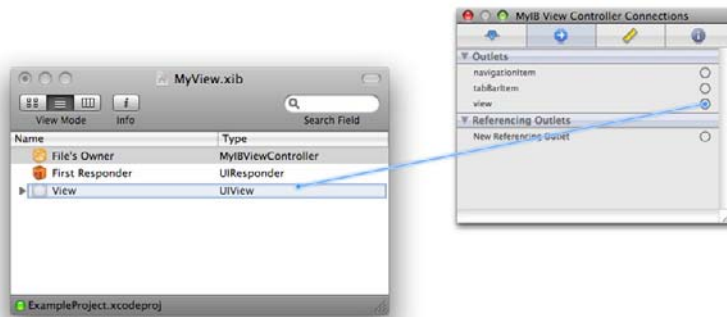
Figure 7-3
Définition du propriétaire de la
vue dans Interface Builder



Il faut ensuite lier la vue qui est contenue dans le fichier XIB à la propriété `view` du contrôleur.

Figure 7-4

Liaison de la vue du fichier XIB avec la vue du contrôleur



Une fois ce lien effectué, vous pouvez construire l'interface en glissant les objets de la bibliothèque vers la fenêtre qui représente la vue.

BEST PRACTICE Un fichier XIB pour un contrôleur

Bien qu'on puisse imaginer avoir plusieurs fichier XIB qui s'appuient sur un seul contrôleur, ou un fichier XIB utilisé par plusieurs contrôleurs, il est fortement recommandé de garder toujours un seul fichier XIB pour un seul contrôleur.

Lier les objets de la vue au contrôleur

Comme nous l'avons vu précédemment, il est possible de définir des liens entre les objets ajoutés à la vue et les propriétés du contrôleur.

Pour cela, il faut définir dans le fichier d'en-tête du contrôleur une propriété et ajouter le mot-clé `IBOutlet` pour que Interface Builder la reconnaisse.

```
@interface MyIBViewController : UIViewController {
    UILabel *myLabel;
}

@property (nonatomic, retain) IBOutlet UILabel *myLabel;

@end
```

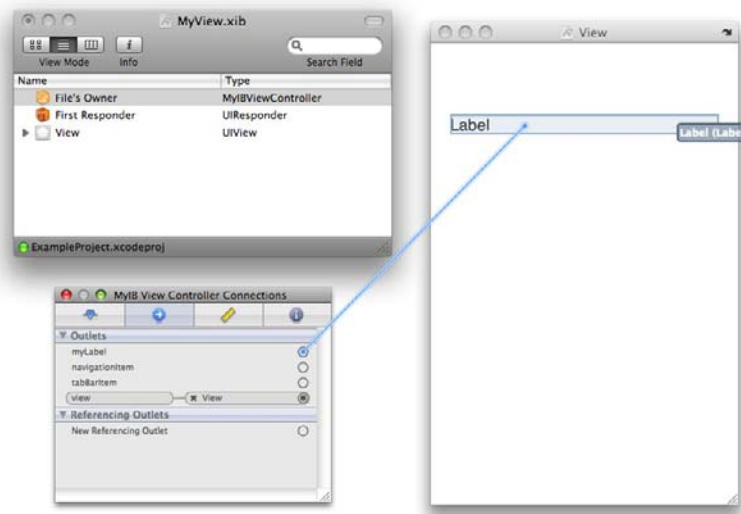
Dans l'implémentation, on demande au compilateur de générer un accesseur et un mutateur (getter et setter) pour cette propriété en utilisant le mot-clé `@synthesize`.

```
@implementation MyIBViewController  
  
@synthesize myLabel;
```

Cette propriété apparaît alors dans l'inspecteur d'Interface Builder (après avoir sélectionné la ligne *File's Owner* dans la fenêtre principale).

Figure 7-5

Création d'un lien entre le contrôleur et un champ texte et un champ texte dans la vue



Réagir au chargement et au déchargement de la vue

Il est souvent utile de pouvoir intervenir une fois la vue chargée pour finir de l'initialiser, lui donner un état initial et éventuellement charger le modèle. De la même façon, il est essentiel de réagir au déchargement de la vue pour s'assurer que toute la mémoire libérable est bien libérée.

Utilisation de la méthode `viewDidLoad`

La méthode `viewDidLoad` est appelée quand la vue a été chargée (en code ou à partir du fichier XIB). Tous les liens entre les objets de la vue et les propriétés du contrôleur ont été établis, il est possible d'ajouter d'autres éléments à la vue, ou de définir des propriétés sur les éléments de la vue.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    myLabel.text = @"Hello World";
}
```

Implémentation de la méthode `viewDidLoad`

Comme nous l'avons vu plus haut, il est essentiel d'implémenter la méthode `viewDidLoad` et de libérer toutes les références vers des objets de la vue. C'est le cas en particulier de toutes les propriétés liées à la vue (via Interface Builder ou via du code).

Le moyen le plus simple est d'assigner la valeur `nil` à la propriété, ce qui décrémentera le compteur si nécessaire (sous réserve que vous ayez bien déclaré les propriétés avec l'attribut `retain` comme vu au chapitre 2 sur la gestion de la mémoire).

Libération de la mémoire lorsque la vue est déchargée

```
- (void)viewDidUnload {
    [super viewDidUnload];
    self.myIBOutlet = nil;
}
```

Comment savoir si la vue est chargée ?

Si elle ne l'était pas déjà, la vue est chargée automatiquement lorsqu'on accède à la propriété `view` du contrôleur de vue. L'exemple suivant aurait donc pour effet de charger systématiquement la vue, ce qui n'est probablement pas ce que souhaite le développeur.

Exemple à ne pas reproduire : provoque systématiquement le chargement de la vue et l'exécution du code dans le bloc `if`.

```
if (self.view != nil) {
    // Faire quelque chose uniquement
    // si la vue est chargée
}
```

La méthode `isViewLoaded` permet de savoir si la vue est chargée, sans forcer le chargement.

Méthode correcte pour effectuer un traitement uniquement si la vue est chargée

```
if ([self isViewLoaded]) {
    // Faire quelque chose uniquement
    // si la vue est chargée
}
```

Réagir lorsque la vue est affichée puis masquée

Il est très utile de savoir à quel moment la vue va être affichée, puis de savoir quand elle est masquée. Ces événements permettent de rafraîchir les éléments du modèle affiché dans la vue, de lancer une mise à jour des données, etc. Il existe quatre méthodes pour suivre ces événements.

Si vous décidez de surcharger ces méthodes, il est indispensable d'appeler la méthode originale. Par exemple : `[super viewWillAppear:animated]`. En effet, de nombreux traitements très importants sont faits par ces méthodes dans l'implémentation de base de `UIViewController`. Ne pas appeler ces traitements entraîne des bogues difficiles à identifier et à corriger.

Affichage de la vue

Lorsque la vue va être affichée, la méthode `viewWillAppear:(BOOL)animated` est appelée. Elle permet de savoir que la vue va être affichée, et de savoir si une animation est en cours pour son affichage.

On utilise généralement cette méthode pour :

- 1 remettre la vue dans un état « initial » : charger le modèle, remplir les objets de la vue ;
- 2 dé-sélectionner une ligne de tableau qui serait encore sélectionnée (nous reverrons ce point au chapitre 10 consacré aux listes d'éléments) ;
- 3 lancer une tâche de fond pour mettre à jour la vue à intervalles réguliers.

Lorsque la vue est affichée, la méthode `viewDidAppear:(BOOL)animated` est appelée. On peut alors :

- 1 lancer une animation pour attirer l'attention de l'utilisateur sur un élément particulier ;
- 2 faire apparaître pendant quelques instants les barres de défilement pour que l'utilisateur ait conscience de leur présence.

Masquage de la vue

De la même façon, il existe deux événements appelés lorsque la vue est masquée. La méthode `viewWillDisappear:(BOOL)animated` et la méthode `viewDidDisappear:(BOOL)animated`. La première est souvent utilisée pour mettre en pause les animations et les éventuelles tâches en arrière-plan.

Gérer les événements

Nous avons vu comment le contrôleur peut agir sur la vue pour créer des éléments d'interface et changer leurs propriétés. La vue doit aussi pouvoir avertir le contrôleur lorsqu'un événement survient.

Il existe plusieurs moyens pour lier les objets de la vue au contrôleur. Le plus simple et le plus répandu est le mécanisme Cible-Action (*target-action* dans la documentation en anglais). Pour un événement donné, on indique une cible (un objet) et une action (un sélecteur). L'action sera effectuée sur la cible à chaque fois que l'événement se produit.

Créer une méthode pour traiter l'événement

Dans le contrôleur, vous devez ajouter une méthode qui sera responsable du traitement de l'événement. En fonction du type d'événement, la signature de la méthode pourra être différente ; pour l'action d'un bouton, il suffit de créer une méthode qui ne prenne aucun paramètre. Donnez-lui comme type de valeur de retour `IBAction` qui est synonyme de `void` mais permet à Interface Builder de reconnaître cette méthode comme une action.

```
- (IBAction) buttonAction;
```

Le corps de la méthode est ajouté dans l'implémentation de la classe :

```
- (IBAction) buttonAction {  
    if (myLabel.textAlignment == NSTextAlignmentLeft)  
        myLabel.textAlignment = NSTextAlignmentRight;  
    else  
        myLabel.textAlignment = NSTextAlignmentLeft;  
}
```

Lier un événement à une action

Encore une fois, il existe deux solutions pour lier un événement à une action : avec ou sans Interface Builder.

Lier un événement à une action en code

Tous les composants graphiques capables d'envoyer des événements héritent de la classe `UIControl` qui définit la méthode `addTarget:action:forControlEvents:`.

Cette méthode permet d'ajouter une cible qui doit être prévenue lorsqu'un événement se produit. Le premier paramètre est l'instance d'objet qui doit être notifiée. Il s'agit en général du contrôleur de vue. Le deuxième paramètre est la méthode à

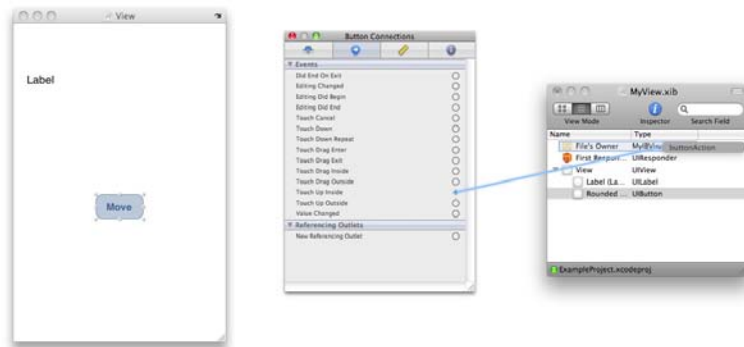
appeler quand l'événement se produit. Le dernier paramètre est le(s) événement(s) pour le(s)quel(s) on veut recevoir une notification.

```
[myButton addTarget:self action:@selector(buttonAction)
forControlEvents:UIControlEventTouchUpInside];
```

Lier un événement à une action en utilisant Interface Builder

La liste des événements générés par un composant graphique est affichée dans l'onglet des connexions d'Interface Builder. Il suffit de faire glisser un lien vers le contrôleur de vue (représenté par l'entité « File's Owner ») pour créer un lien avec l'une des méthodes du contrôleur de vue.

Figure 7-6
Création d'un lien entre un événement et une méthode du contrôleur de vue dans Interface Builder



Gérer les rotations d'écran

La classe `UIViewController` fournit une logique basique pour gérer les rotations d'écran. Vous pouvez également fournir votre propre code de rotation si vous souhaitez réaliser des effets particuliers.

Lorsque l'utilisateur bascule son iPhone, l'accéléromètre détecte un changement d'orientation et notifie le contrôleur de vue. Celui-ci appelle la méthode `shouldAutorotateToInterfaceOrientation:` en passant en paramètre la nouvelle orientation. La méthode doit renvoyer un booléen indiquant s'il faut faire pivoter la vue ou pas. Par défaut, cette méthode n'autorise que le mode portrait. La vue ne pivote donc jamais.

Tableau 7-1 Les différentes orientations de l'iPhone

Orientation	Constante
iPhone tenu verticalement	<code>UIInterfaceOrientationPortrait</code>
iPhone tenu "tête en bas"	<code>UIInterfaceOrientationPortraitUpsideDown</code>
iPhone tenu horizontalement avec le bouton home à droite	<code>UIInterfaceOrientationLandscapeLeft</code>
iPhone tenu horizontalement avec le bouton home à gauche	<code>UIInterfaceOrientationLandscapeRight</code>

Dans l'exemple suivant, on demande à la vue de basculer automatiquement quand l'utilisateur met le téléphone en mode portrait ou en mode paysage (bouton home à droite).

```
-
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)in
terfaceOrientation {
    return (interfaceOrientation == UIInterfaceOrientationPortrait) ||
(interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
}
```

APPROFONDIR Comment la vue est-elle adaptée lors d'une rotation ?

Le mécanisme de rotation d'écran va provoquer un redimensionnement de la vue. Le résultat ne sera satisfaisant que si la vue sait s'adapter à la nouvelle taille de l'écran.

Il faut pour cela définir comment la vue doit être redimensionnée, ce qui se fait via l'onglet *Size* d'Interface Builder ou via les propriétés `autoresizeSubviews` et `autoresizingMask` de la classe `UIView`.

Événements associés aux rotations d'écran

Lors d'une rotation, plusieurs événements sont appelés pour permettre à la vue de réaliser des traitements spécifiques.

Juste *avant* que la vue ne soit pivotée, est appelée la méthode `willRotateToInterfaceOrientation:duration:`. Vous pouvez utiliser cette méthode pour pousser un nouveau contrôleur de vue modal (traité dans le chapitre suivant), qui sera affiché uniquement quand l'application est dans une orientation et le faire disparaître quand l'application revient dans l'orientation normale.

Juste *après* la rotation, la méthode `didRotateFromInterfaceOrientation:` est appelée. Vous pouvez l'utiliser pour mettre à jour la vue avec de nouvelles informations, lancer un traitement particulier, etc.

APPROFONDIR Les rotations de contrôleur de vue

La méthode présentée ici est la plus simple et la plus rapide pour mettre en place des contrôleurs de vue qui supportent des rotations automatiques. Vous devriez toujours essayer d'utiliser ce mécanisme avant d'explorer des méthodes plus compliquées. En effet, les animations associées sont très complexes et difficiles à réimplémenter correctement (par exemple, la barre de navigation et la tabBar ne tournent pas avec la vue mais sont d'abord retirées, le contenu tourne, puis elles sont rajoutées dans la vue). Il existe néanmoins de nombreuses possibilités pour contrôler très précisément les animations exécutées durant la rotation d'écran et les redéfinir pour votre propre besoin. La documentation de la classe `UIViewControllerAnimatedTransitioning` les détaille.

Conclusion

Dans ce chapitre, vous avez appris comment créer des contrôleurs de vue. Souvenez-vous que pour chaque écran de votre application, il faut en implémenter un.

Tout contrôleur de vue est implémenté en dérivant la classe `UIViewController` et en surchargeant une partie de ses méthodes. Pensez à appeler l'implémentation de la classe parente, en particulier pour `viewWillAppear:`, `viewDidLoad:`, etc. (en cas de doute, la documentation précise systématiquement si cela est nécessaire ou pas).

Enfin, n'oubliez pas que votre vue peut être libérée en cas d'avertissement mémoire et que votre code doit permettre de libérer complètement la vue et de la recréer sans fuite mémoire. En effet, ce phénomène se produit souvent dans une application complexe et c'est une source importante de bogues.

8

Assembler les écrans de l'application

Nous avons vu dans le chapitre précédent comment implémenter un écran de l'application, mais la très grande majorité des applications sont construites à l'aide de plusieurs écrans qui s'enchaînent selon un des design patterns de navigation décrits au chapitre 5.

Dans ce chapitre, nous verrons comment assembler les écrans de l'application selon deux méthodes : en utilisant des contrôleurs conçus pour contenir vos contrôleurs de vue ; et en utilisant des méthodes de `UINavigationController` pour afficher des contrôleurs de vue modale.

Généralités sur les contrôleurs-conteneurs

Les deux contrôleurs-conteneurs proposés sont le contrôleur de navigation et le contrôleur d'onglets. Ils sont fournis par UIKit et diffèrent des contrôleurs de vue standard sur trois points très importants :

- 1 Ils contiennent d'autres contrôleurs et leur transmettent les événements.
- 2 Ils ne sont pas destinés à être hérités dans votre application. Vous devez les utiliser tels qu'ils sont fournis et ne pas chercher à surcharger leurs méthodes.
- 3 Ils gèrent plusieurs écrans de l'application (en s'appuyant sur vos propres contrôleurs).

CONSEIL Évitez de créer d'autres contrôleurs-conteneurs

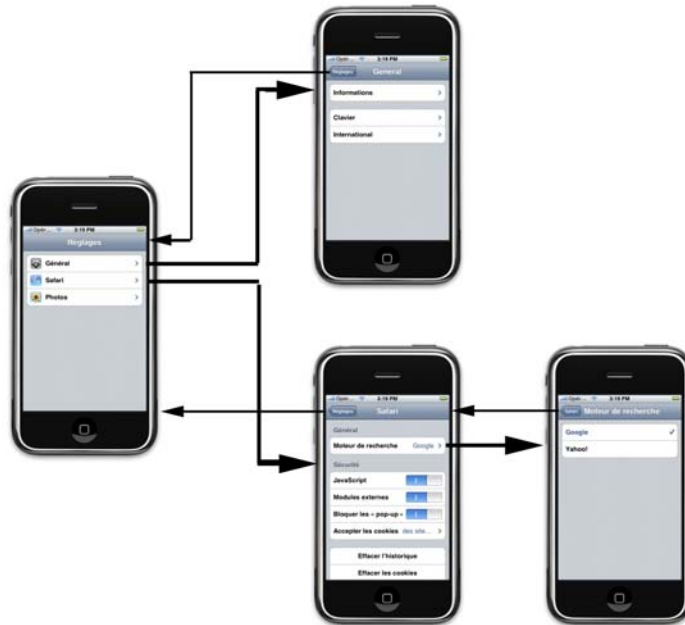
Bien qu'il soit possible de créer vos propres contrôleurs-conteneurs, il est relativement complexe de le faire correctement et impossible de garantir un bon fonctionnement avec les versions futures d'iPhone OS. Nous n'aborderons donc pas ce sujet.

Le contrôleur de navigation

La classe `UINavigationController` fournit toute la logique nécessaire pour implémenter le design pattern de navigation dans des listes hiérarchiques. Elle prend en charge l'affichage de la barre de navigation en haut de l'écran (titre et boutons), les animations entre les écrans et de nombreux petits détails indispensables pour retrouver dans votre application la même fluidité que dans les applications de l'iPhone.

Figure 8-1

Le contrôleur de navigation permet d'assembler les différents contrôleurs utilisés dans vos listes hiérarchiques.



Lorsque vos contrôleurs de vue sont ajoutés dans un contrôleur de navigation, ils sont automatiquement redimensionnés pour s'ajuster à la taille disponible sur l'écran (le contrôleur de navigation prend en compte la taille de la barre de navigation en haut, mais aussi la taille de la barre d'onglets ou de la barre de boutons qui peuvent également être présentes).

Création d'un contrôleur de navigation

Il suffit d'instancier la classe `UINavigationController` en lui fournissant un contrôleur qui sera le premier écran de la hiérarchie (et donc affiché immédiatement).

```
UINavigationController *myNavController =  
[[UINavigationController alloc]  
 initWithRootViewController:myViewController];
```

Figure 8-2

Ajout d'un contrôleur de navigation comme conteneur d'un contrôleur



Spécifier le contenu de la barre de navigation

La barre de navigation est divisée horizontalement en trois parties :

- une zone dans laquelle le bouton « Retour » vient se placer lorsqu'il existe une vue précédente ;
- une zone centrale pour afficher le titre de la vue en cours ;
- une zone à droite qui peut servir à afficher un bouton d'action (comme le bouton « + » dans l'application Contact par exemple).

Figure 8-3

Les différentes zones de la barre de navigation



Tous ces éléments sont paramétrés par l'intermédiaire d'une propriété qui est présente sur tous les contrôleurs de vue, mais qui n'est utilisée que lorsque le contrôleur est ajouté à un contrôleur de navigation. Il s'agit de la propriété `navigationItem` qui est un objet du type `UINavigationControllerItem`.

Le point important à retenir ici est que chaque contrôleur de vue indique via cette propriété comment il souhaite être représenté. Le contrôleur de navigation va lire ces informations et anime correctement les transitions pour s'assurer qu'elles aient toujours l'air naturel.

Titre du contrôleur

La propriété la plus importante de cet objet est `title` qui indique le titre du contrôleur. On peut par exemple la définir dans le constructeur du contrôleur de vue.

Définition du titre d'un contrôleur de vue

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
        self.navigationItem.title = @"Ma vue";
    }
    return self;
}
```

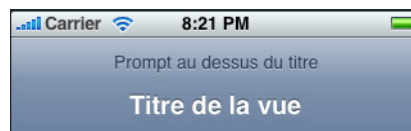
Pour afficher autre chose que du texte dans la zone de titre, il est possible de fournir une vue via la propriété `titleView`. Dans ce cas, la propriété `title` n'est plus utilisée. Nous étudierons les vues en détail dans le prochain chapitre, mais l'exemple ci-après montre comment utiliser une image comme titre.

Utilisation d'une image comme titre d'un contrôleur dans une liste de navigation

```
UIImageView *myTitleView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"mytitle.png"]];
self.navigationItem.titleView = myTitleView;
[myTitleView release];
```

Enfin, il est également possible de définir un prompt (propriété `prompt`) qui est un message affiché au-dessus du titre, pour indiquer à l'utilisateur ce qu'il doit faire dans cet écran.

Figure 8-4
Une barre de navigation avec un prompt



Boutons supplémentaires

Par défaut, et si le contrôleur en cours n'est pas le premier, le contrôleur de navigation ajoute un bouton permettant de revenir à l'écran précédent. Sinon, il n'affiche rien d'autre que le titre.

Il est possible de remplacer ce bouton et même d'afficher un bouton supplémentaire dans la partie droite de la barre. Pour cela, on renseigne les deux propriétés `leftBarButtonItem` et `rightBarButtonItem` de `navigationItem`. Ce sont tous les deux des objets `UIBarButtonItem`. Il est possible d'instancier `UIBarButtonItem` à partir d'une image, d'un texte ou bien d'une icône système.

Mise en place d'un bouton « Appareil photo » dans la barre de navigation

```
UIBarButtonItem *item = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemCamera target:self
action:@selector(buttonCameraPressed)];
self.navigationItem.rightBarButtonItem = item;
[item release];
```

BEST PRACTICE Ne pas changer le bouton de gauche

À moins que votre contrôleur soit le premier de l'application, il n'est pas du tout recommandé de mettre un autre bouton à la place qu'occupe habituellement le bouton retour. En effet, vous devriez alors proposer un autre mécanisme pour permettre à l'utilisateur de revenir en arrière dans son parcours de navigation et le résultat serait déroutant.

Définir la façon dont est représenté le contrôleur quand il n'est plus affiché

Lorsque l'utilisateur avance d'un niveau dans la navigation et qu'un autre contrôleur vient recouvrir la vue actuelle, votre contrôleur est encore représenté sous la forme d'une flèche dans la partie gauche de la barre de navigation.

Par défaut, cette flèche contient le titre de votre contrôleur (il sera tronqué automatiquement s'il était trop long), mais vous pouvez fournir votre propre `UIBarButtonItem` si vous le souhaitez, via la propriété `backBarButtonItem`.

CONSEIL Aidez l'utilisateur à retrouver son chemin

En indiquant juste « Retour » sur le bouton permettant de revenir à votre contrôleur, vous n'aidez pas l'utilisateur à se souvenir de ce qu'il trouvera dans cet écran.

Il est recommandé d'utiliser un nom court et clair qui permette à l'utilisateur de se souvenir de ce qu'il y a derrière ce bouton. En observant les applications iPhone, vous verrez que ce petit détail est très souvent utilisé : dans les mails par exemple, le nom de la boîte mail ou du dossier est indiqué.

Masquer la barre d'outils ou la barre d'onglets

Comme nous le verrons dans la suite de ce chapitre, il est possible d'utiliser un contrôleur de navigation en combinaison avec des barres d'outils ou avec une barre d'onglets.

Dans certaines applications, on peut vouloir masquer cette barre pour certains des écrans de l'application. C'est souvent le cas quand on veut afficher des photos par exemple.

La propriété `hidesBottomBarWhenPushed` du contrôleur de navigation permet d'indiquer que la barre située en bas de l'écran doit être masquée lorsque ce contrôleur est affiché.

```
self.hidesBottomBarWhenPushed = YES;
```

Pousser des écrans dans le contrôleur de navigation

Lorsque l'utilisateur sélectionne un élément dans la vue, le contrôleur courant peut demander au contrôleur de navigation d'animer la transition vers le prochain contrôleur (le prochain écran vers la droite).

On appelle pour cela la méthode `pushViewController:animated:` qui ajoute un contrôleur à un niveau inférieur dans la hiérarchie et anime une transition.

APPROFONDIR Quand utiliser le paramètre `animated`

De nombreuses méthodes du SDK prennent un paramètre `animated` qui permet d'indiquer s'il faut animer l'apparition d'un élément. La plupart du temps, on souhaite afficher une animation, mais il est parfois souhaitable d'amener l'utilisateur directement à un écran sans animation (lorsqu'on relance l'application par exemple).

Le contrôleur de navigation gère la pile des contrôleurs, il va ajouter à la pile le nouveau contrôleur passé en paramètre, l'afficher et le retenir en mémoire jusqu'à ce qu'il ne soit plus utilisé. Le retour en arrière est le plus souvent provoqué par l'utilisateur lorsqu'il utilise le bouton de gauche dans la barre de navigation.

Le contrôleur de vue qui est contenu dans le contrôleur de navigation a besoin d'une référence vers son contrôleur-conteneur pour pouvoir appeler la méthode `pushViewController:.`

Il la trouvera toujours dans sa propriété `navigationController` : en effet, à chaque fois qu'un contrôleur est ajouté dans un contrôleur de navigation, la propriété `navigationController` est définie automatiquement.

Création d'un nouveau contrôleur de vue, et ajout dans le contrôleur de navigation

```
- (IBAction) buttonAction {
    MyViewController *myViewController = [[MyViewController alloc]
initWithNibName:nil bundle:nil];
    [self.navigationController pushViewController:myViewController
animated:YES];
    [myViewController release];
}
```

On remarque que le nouveau contrôleur peut être relâché. En effet, il est automatiquement retenu jusqu'à ce qu'il disparaisse (quand l'utilisateur presse le bouton « Retour »).

Personnaliser la barre de navigation

La barre de navigation elle-même peut être personnalisée. Via la propriété `navigationBar` de l'objet `navigationController` vous pouvez accéder à l'objet `UINavigationController` qui est une vue représentant la barre.

Les propriétés les plus intéressantes sont décrites dans le tableau ci-après.

Tableau 8-1 Propriétés pour adapter le look de la barre de navigation

Propriété	Utilisation
Style de la barre (<code>barStyle</code>)	Permet de choisir le style de la barre : bleu (<code>UIBarStyleDefault</code>) ou noir (<code>UIBarStyleBlack</code>)
Transparence de la barre (<code>translucent</code>)	Permet d'indiquer que la barre doit être transparente. Dans ce cas, le contrôleur de navigation fera automatiquement en sorte que les vues de vos contrôleurs de vue passent en dessous de la barre.
Couleur de la teinte de la barre (<code>tintColor</code>)	Cette propriété permet de fournir une couleur qui sera utilisée comme teinte de la barre.

Contrôleur d'onglets

La navigation par onglets permet de créer des applications dans lesquelles plusieurs modes parallèles sont proposés à l'utilisateur.

Tout comme le contrôleur de navigation, le contrôleur d'onglets est un composant fourni par le SDK qui implémente toute la logique nécessaire à la gestion d'une barre d'onglets et qui n'est pas conçu pour être dérivé.

Ce composant gère la barre d'onglets affichée en bas de l'écran, mais aussi le mécanisme qui permet, lorsqu'il y a trop d'éléments, de faire apparaître un onglet *Autre* et de proposer la personnalisation de la barre d'onglets (voir le chapitre 5, section « Navigation par onglet » pour un rappel de ce design pattern de navigation).

Création d'un contrôleur d'onglets

Le contrôleur d'onglets est implémenté dans la classe `UITabBarController`. Son constructeur prend en paramètre une liste de contrôleurs à afficher dans les onglets.

Création et initialisation d'un contrôleur d'onglet

```
NSArray *controllers = [[NSArray alloc] initWithObjects:controller1,  
controller2, nil];  
UITabBarController *tabBarController = [[UITabBarController alloc]  
initWithControllers:controllers];  
[controllers release];
```

Les onglets sont affichés dans l'ordre de la liste passée en paramètre. S'il y a plus de cinq onglets, les quatre premiers sont affichés, et un onglet *Et aussi* est ajouté en cinquième position.

Personnalisation du titre et de l'icône des onglets

Comme pour le contrôleur de navigation, chaque contrôleur définit dans une propriété comment il doit être représenté dans la barre d'onglets, et le contrôleur d'onglets va lire cette propriété pour afficher un titre et une icône.

La propriété `tabBarItem` du type `UITabBarItem` définit deux propriétés : `title` et `image`. L'image est un PNG de 30 x 30 pixels et il doit avoir une couche alpha (la couche alpha indique où l'image est transparente).

Lorsque l'onglet n'est pas sélectionné, la barre d'onglets affiche l'icône fournie en la passant en noir et blanc. Pour cela, elle affiche un pixel gris partout où l'image n'est pas transparente. Lorsque l'onglet est sélectionné, elle affiche les mêmes pixels mais avec un masque dégradé bleu.

Définition de la propriété `tabBarItem`

```
self.tabBarItem.image = [UIImage imageNamed:@"icone.png"];  
self.tabBarItem.title = @"titre";
```

Réagir aux événements de la barre d'onglets

La barre d'onglets peut transmettre des événements lorsque l'utilisateur utilise la fonction de personnalisation.

Pour être notifié, l'objet qui construit la barre d'onglets doit fournir un délégué qui implémente le protocole `UITabBarControllerDelegate`. Comme c'est le plus souvent le délégué d'application qui crée la barre d'onglets, ce sera généralement lui qui servira directement de délégué.

Mise en place d'un délégué de la barre d'onglets

```
tabBarController.delegate = self;
```

Suivre les changements de sélection

Grâce au délégué du contrôleur d'onglets, il est possible d'être informé lorsque l'utilisateur change d'onglet, et même d'empêcher un changement d'onglet.

La méthode `shouldSelectViewController:` reçoit en paramètre le `viewController` vers lequel l'utilisateur veut aller, et attend en retour un booléen. Si la méthode renvoie `NO`, la barre d'onglets ne change pas l'onglet affiché.

La méthode `didSelectViewController:` passe en paramètre le contrôleur qui vient d'être sélectionné par l'utilisateur. Elle est appelée à chaque fois que l'utilisateur clique sur l'onglet, même si cet onglet était déjà affiché.

Réagir à la personnalisation de la barre

Il existe deux méthodes dans le délégué très intéressantes pour suivre la personnalisation de la barre.

- La première `tabBarController:willBeginCustomizingViewControllers:` permet d'être informé que l'utilisateur commence à personnaliser les onglets.
- La deuxième `tabBarController:didEndCustomizingViewControllers:changed:` permet de savoir que l'utilisateur a fini de modifier la barre d'onglets. Elle reçoit en paramètre la nouvelle liste de contrôleurs de vue et un booléen indiquant si la liste a été modifiée ou pas.

C'est généralement cette deuxième méthode qui est la plus utilisée, car elle permet de sauver l'état de la barre d'onglets pour pouvoir recharger la barre dans le même état lors du prochain redémarrage de l'application.

BEST PRACTICE Enregistrement de l'état de la barre d'onglets dans les préférences de l'utilisateur

La classe `NSUserDefaults` qui sera abordée au chapitre 11 est parfaitement adaptée pour stocker les préférences utilisateur. Elle permet de les lire et de les sauver en quelques lignes de code.

Limiter la personnalisation de la barre d'onglets

Il est possible de limiter les possibilités de personnalisation de la barre d'onglets en renseignant la propriété `customizableViewControllers` du contrôleur d'onglets.

Cette propriété contient une liste de contrôleurs qui peuvent être personnalisés (déplacés). Par défaut, c'est la liste fournie lors de l'initialisation de la barre d'onglets, mais il est possible de fournir une liste contenant moins d'éléments. Les éléments manquants seront alors figés à la place qui leur a été donnée lors de l'initialisation.

En mettant cette propriété à `nil`, on empêche toute personnalisation de la barre d'onglets (le bouton *Modifier* n'apparaîtra plus dans l'onglet *Autre*).

Combiner les contrôleurs d'onglets avec des contrôleurs de navigation

Il est possible d'utiliser des contrôleurs de navigation comme onglet dans un contrôleur d'onglets, mais l'inverse n'est pas autorisé.

Combinaison d'un contrôleur d'onglets avec un contrôleur de navigation

```
UINavigationController *myNavController = [[UINavigationController alloc]
initWithRootController:onglet3ViewController];

NSArray *controllers = [[NSArray alloc] initWithObjects:onglet1ViewController,
onglet2ViewController, myNavController, nil];

UITabBarController *tabBarController = [[UITabBarController alloc]
initWithControllers:controllers];

[controllers release];

[myNavController release];
```

Affichage d'un contrôleur en mode modal

On parle d'un affichage en mode modal quand la vue affichée vient recouvrir l'ensemble de l'écran, et qu'il faut fermer cette vue pour revenir à la vue précédente.

La classe `UIViewController` fournit une méthode qui permet d'afficher un autre contrôleur par-dessus le contrôleur actuel, en mode modal. La nouvelle vue recouvre tout l'écran, même si le contrôleur initial fait partie d'un contrôleur-conteneur et qu'il faut qu'elle soit annulée pour revenir à l'état précédent. Il est possible d'empiler plusieurs contrôleurs modaux les uns sur les autres, et également de définir une animation qui servira de transition vers la vue modale.

Pousser une nouvelle vue modale

Pour pousser une nouvelle vue modale, le contrôleur doit tout d'abord créer un nouveau contrôleur de vue puis appeler la méthode `presentModalViewController:animated:`. Le premier paramètre est le contrôleur à afficher, et le deuxième indique s'il faut déclencher une animation.

Exemple : déclenchement d'une vue modale lorsqu'un bouton est pressé

```
-(void) chooseAvatarButtonAction
{
    MyAvatarPickerController *chooseAvatarViewController =
        [[MyAvatarPickerController alloc] init];
    [self presentModalViewController:chooseAvatarViewController animated:YES];
    [chooseAvatarViewController release];
}
```

Faire disparaître une vue modale

Pour faire disparaître une vue modale, il faut appeler la méthode `dismissModalViewControllerAnimated:` sur le contrôleur qui détient la vue modale (c'est-à-dire celui qui l'a affichée en appelant `presentModalViewController`).

Dans certains cas, c'est la vue modale elle-même qui sait quand elle doit disparaître. Dans ce cas, elle peut retrouver une référence vers le contrôleur qui la détient via sa propriété `parentViewController`.

Exemple : disparition de la vue modale lorsqu'un bouton est cliqué

```
(void) buttonAction
{
    [self.parentViewController dismissModalViewControllerAnimated:YES];
}
```

Définir le mode de transition

C'est le contrôleur modal qui définit quelle est la transition qui sera utilisée pour le faire apparaître et le faire disparaître.

La propriété `modalTransitionStyle` du contrôleur peut prendre une des valeurs du tableau ci-après.

Tableau 8-2 Types de transitions possibles pour une vue modale

Transition	Effet
<code>CoverVertical</code>	La vue modale apparaît par le bas de l'écran et monte jusqu'à recouvrir tout l'écran. Elle fait le chemin inverse pour disparaître.
<code>FlipHorizontal</code>	La vue actuelle pivote sur un axe vertical pour révéler la vue modale (effet de l'application Météo lorsqu'on clique sur le bouton de paramétrage).
<code>CrossDissolve</code>	La nouvelle vue apparaît via un effet de fondu enchaîné.

Conclusion

Nous avons vu dans ce chapitre comment assembler les écrans d'une application iPhone pour construire un ensemble cohérent et qui reprenne les codes iPhone.

Les méthodes présentées devraient répondre à la très grande majorité des besoins. Si vous ressentez l'envie de faire vos propres contrôleurs-conteneurs, commencez par vérifier que votre objectif n'est pas atteignable avec les éléments fournis en standard par Apple, et envisagez de revoir la navigation de votre application : si elle n'est pas implémentable facilement, il y a de fortes chances qu'elle ne respecte pas les standards d'ergonomie.

9

Développer et animer les vues

Tous les composants graphiques proposés par UIKit pour réaliser les interfaces des applications iPhone sont des vues.

Certains sont très simples (comme un champ texte), certains réagissent aux interactions de l'utilisateur (comme un bouton) et d'autres sont extrêmement complexes (comme la vue web qui permet d'afficher des contenus HTML) ; mais ils descendent tous de la classe `UIView`.

Dans ce chapitre, nous présenterons cette classe `UIView`, ses propriétés et son fonctionnement. Nous expliquerons comment composer des vues complexes à partir de vues simples et nous présenterons enfin les composants graphiques les plus importants.

Comprendre les vues

Une vue est un objet qui dérive de `UIView`. Elle représente une sous-partie de sa vue parente.

Coordonnées des vues

Sur iPhone, les coordonnées sont toujours mesurées en pixels et avec une origine située en haut à gauche. L'axe X est l'axe horizontal et l'axe Y est l'axe vertical.

Ainsi, en supposant que l'iPhone est tenu verticalement :

- (0,0) représente le coin supérieur gauche ;
- (320, 0) le coin supérieur droit ;
- (0, 480) le coin inférieur gauche ;
- et (320, 480) le coin inférieur droit.

Centre et limites d'une vue

La zone occupée par une vue est définie par la position du centre de la vue et les limites de la vue (largeur, hauteur). Ce sont les propriétés `center` de type `CGPoint` et `bounds` de type `CGRect`.

```
UIView *view = [[UIView alloc] init];  
view.bounds = CGRectMake(0, 0, 100,50);  
view.center = CGPointMake(160,100);
```

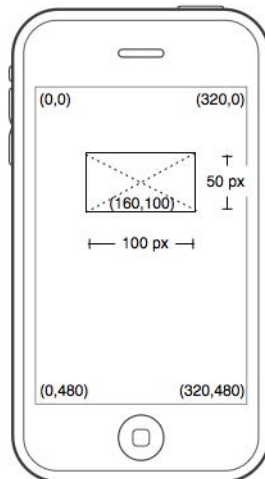
ATTENTION Origine des coordonnées

Le centre est toujours mesuré dans le système de coordonnées de la vue parente.

ATTENTION Limites de la vue

Il est possible d'exploiter l'origine du rectangle de la propriété `bounds` pour forcer la vue à ne dessiner qu'une partie d'elle-même. Dans la plupart des cas cependant, l'origine utilisée est (0,0) ce qui revient à définir la largeur et la hauteur.

Figure 9-1
Centre, taille et position
d'une vue



Frame d'une vue

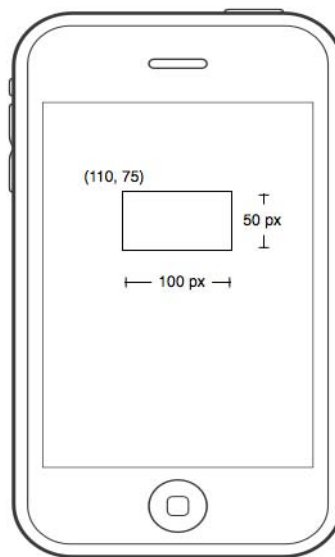
Les coordonnées d'une vue peuvent également être exprimées sous forme d'un rectangle. Dans ce cas, on donne la position du coin en haut à gauche, puis la largeur et la hauteur de la vue.

Vous pouvez accéder au rectangle représentant la vue grâce à la propriété `frame` de type `CRect`. Elle est accessible en lecture et en écriture (les propriétés `center` et `bounds` sont synchronisées automatiquement).

```
// Equivalent à l'exemple précédent.  
view.frame = CGRectMake(0, 0, 100, 50);
```

Figure 9-2

Coordonnées en utilisant la `frame` plutôt que le centre et la taille.



POINT D'ATTENTION Bonne utilisation de la propriété `frame`

La propriété `frame` étant plus naturelle à manipuler, on l'utilisera plus souvent que les propriétés `bounds` et `center`.

Il existe pourtant des cas pour lesquels la propriété `frame` n'est plus du tout équivalente aux propriétés `bounds` et `center`. En particulier, dès qu'une transformation (rotation ou zoom) est effectuée sur une vue grâce à la propriété `transform`.

Hiérarchie des vues

Les vues forment un arbre dont la racine est la fenêtre de l'application (qui est également une vue).

Chaque vue peut contenir des vues filles, et garde une référence vers sa vue parente.

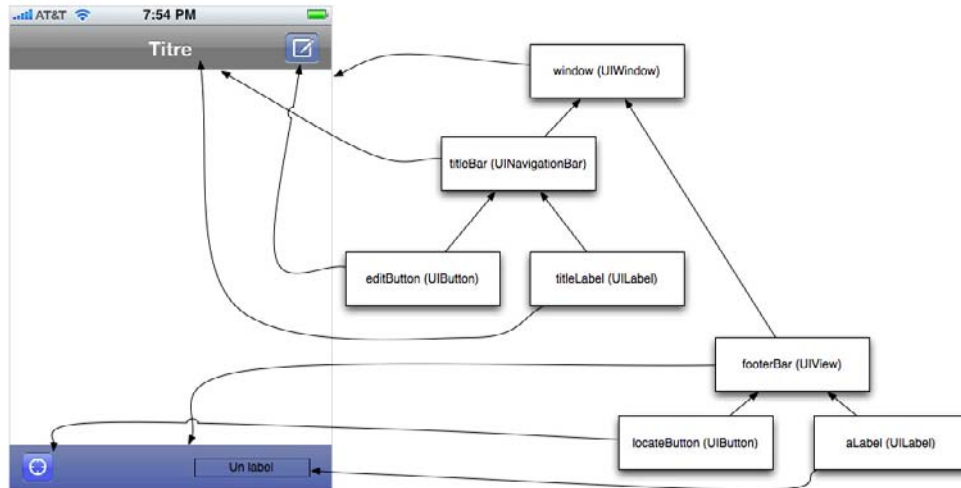


Figure 9-3 Hiérarchie de vue

Pour récupérer la référence vers la vue parente, on utilise la propriété `parentView`. Pour ajouter une sous-vue à une vue, on utilise la méthode `addSubview:`, et pour lister les sous-vues, il existe la propriété `subviews`.

Positionnement des sous-vues et redimensionnement automatique des vues

Une vue peut gérer le positionnement de ses sous-vues, et ajuster leur position quand sa taille change.

Ce travail est fait dans la méthode `layoutSubviews`. Cette méthode ne doit jamais être appelée directement. On demande à ce qu'elle soit appelée par le sous-système d'affichage des vues en appelant `setNeedsLayout`.

L'implémentation par défaut de `layoutSubviews` s'appuie sur deux propriétés qui indiquent ce qu'elle doit faire :

- la propriété `autoresizeSubviews` indique si elle doit ou pas essayer de redimensionner les sous-vues ;

- la propriété `autoresizingMask` indique quels sont les ajustements que la vue peut faire.

Cette dernière propriété s'appuie sur les valeurs suivantes, qui peuvent être combinées avec l'opérateur binaire OU :

Tableau 9-1 Valeurs pour `autoresizingMask`

Valeur	Comportement
<code>UIAutoresizingMaskFlexibleHeight</code> <code>UIAutoresizingMaskFlexibleWidth</code>	Indique que la hauteur et/ou la largeur de la vue peuvent être ajustées.
<code>UIAutoresizingMaskFlexibleMarginLeft</code> <code>UIAutoresizingMaskFlexibleMarginRight</code> <code>UIAutoresizingMaskFlexibleMarginTop</code> <code>UIAutoresizingMaskFlexibleMarginBottom</code>	Indique que certaines des marges de la vue peuvent être agrandies ou rétrécies.

Les vues élémentaires de UIKit

UIKit fournit de nombreuses vues que vous pouvez composer ensemble pour créer des interfaces iPhone riches. Nous ne passerons en revue ici que les éléments les plus essentiels.

Interface Builder est un excellent moyen de découvrir les nombreux composants et les possibilités de paramétrage. Vous devriez également installer sur votre iPhone l'application `UIColorCatalog` fournie par Apple dans les exemples de code du SDK et qui permet de voir en action sur le téléphone toutes les vues standard.

Les labels pour afficher du texte

La classe `UILabel` permet d'afficher un texte à l'écran. Le développeur peut choisir la taille du texte, sa couleur, comment l'aligner et éventuellement comment le tronquer.

Tableau 9-2 Principales propriétés de `UILabel`

Propriété	Effet
<code>text</code>	Permet de définir le texte à afficher.
<code>font</code>	Permet de choisir la police à utiliser pour afficher le texte.
<code>textColor</code>	Permet de choisir la couleur du texte.
<code>textAlignment</code>	Permet de choisir le mode d'alignement du texte : <code>UITextAlignmentLeft</code> , <code>UITextAlignmentCenter</code> , <code>UITextAlignmentRight</code> .
<code>shadowOffset</code>	Permet d'indiquer le décalage entre le texte et son ombre.
<code>shadowColor</code>	Permet d'indiquer la couleur de l'ombre.

Exemple d'utilisation de UILabel

```
UILabel *myLabel = [[UILabel alloc] initWithFrame:CGRectMake(20,20, 100, 20)];
myLabel.text = @"TestLabel";
myLabel.textColor = [UIColor blackColor];
myLabel.textAlignment = NSTextAlignmentCenter;
myLabel.font = [UIFont fontWithName:@"Helvetica" size:14];
myLabel.shadowOffset = CGSizeMake(-1.0, 0.0);
myLabel.shadowColor = [[UIColor blackColor]
                       colorWithAlphaComponent:0.3];
[mainView addSubview:myLabel];

[myLabel release];
```

ASTUCE Couleur et transparence

La méthode `colorWithAlphaComponent:` de la classe `UIColor` permet d'obtenir une couleur avec transparence à partir d'une couleur standard. La couleur `clearColor` (obtenue grâce à `[UIColor clearColor]`) est complètement transparente.

Il est ainsi très facile de préparer des interfaces très riches et qui utilisent abondamment les transparences. Votre seule limite sera alors celles des performances de l'iPhone.

Les vues d'images

La vue `UIImageView` permet d'afficher une image. Les principaux formats d'images pris en charge par l'iPhone sont le PNG, le JPEG et le GIF (la liste complète est disponible dans la documentation de `UIImage`).

CONSEIL Utilisez le format PNG

Le format PNG est le plus efficace sur l'iPhone. Les autres formats sont convertis en mémoire dans ce format avant d'être affichés.

En pratique, il est très fortement conseillé de préparer toutes vos ressources au format PNG et de ne pas utiliser d'autres formats d'images.

Pour créer une vue `UIImageView`, il faut lui passer en paramètre un objet `UIImage` qui est la représentation en mémoire de toutes les images pour UIKit.

Exemple d'utilisation de UIImageView

```
UIImageView *imageView = [[UIImageView alloc] initWithImage:[UIImage
imageNamed:@"myimage.png"]];

[mainView addSubview:imageView];

[imageView release];
```

En utilisant la propriété `contentMode` (de type `UIViewContentMode`), on peut définir comment l'image doit être ajustée par rapport à la taille de la vue.

Exemple d'utilisation de `UIImageView` en forçant la taille de la zone d'affichage

```
UIImageView *imageView = [[UIImageView alloc]
initWithFrame:CGRectMake(10, 10, 300, 460)];
imageView.image = [UIImage imageNamed:@"myimage.png"];
imageView.contentMode = UIViewContentModeScaleAspectFit;

[mainView addSubview:imageView];

[imageView release];
```

Les boutons pour déclencher des actions

La classe `UIButton` permet d'afficher une zone cliquable. Il est possible d'utiliser le style par défaut, d'utiliser un des styles prédéfinis, ou bien de fournir vos propres images pour représenter les différents états du bouton.

Nous avons déjà vu dans les chapitres traitant des contrôleurs de vues comment mettre en place une action exécutée lorsque le bouton est utilisé (via le mécanisme cible-action). On initialise un bouton à l'aide de la méthode `buttonWithType:` ; la liste des types disponibles est détaillée dans la documentation, mais le plus important est `UIButtonTypeCustom`. Dans ce cas, le bouton créé est complètement transparent.

Il est alors possible de définir l'image à utiliser pour chaque état du bouton :

- `UIControlStateNormal`,
- `UIControlStateHighlighted` (l'utilisateur a cliqué sur le bouton) et
- `UIControlStateDisabled` (le bouton est désactivé).

On définit l'image à utiliser à l'aide de la méthode :

```
setImage:(UIImage*) forState:(UIControlState)
```

Les zones de texte

La classe `UITextField` permet de définir une zone de texte dans laquelle l'utilisateur va pouvoir saisir du texte. Lorsqu'il cliquera dans cette zone, le clavier apparaîtra automatiquement. Votre application doit si nécessaire déplacer les autres éléments à l'écran pour s'assurer que le clavier n'est pas venu par-dessus la zone de texte.

Pour avoir un contrôle fin de l'édition, votre contrôleur doit s'enregistrer comme délégué de la zone de texte. Pour cela, il doit implémenter le protocole `UITextFieldDelegate`.

La méthode `-(BOOL)textFieldShouldReturn:(UITextField *)textField` de ce délégué est appelée quand l'utilisateur a fini d'éditer le texte. Pour faire disparaître le clavier, vous pouvez alors appeler la méthode `resignFirstResponder:` de la zone de texte.

Exemple d'utilisation des zones de texte

```
- (void) loadView
{
    ...
    UITextField *textField = [[UITextField alloc]
initWithFrame:CGRectMake(20, 20, 100, 20)];
    textField.delegate = self;
    [view addSubview:textField];
    [textField release];
    ...
}

- (BOOL) textFieldShouldReturn:(UITextField *) textField
{
    NSLog(@"Le texte saisi par l'utilisateur est %@", textField.text);
    [textField resignFirstResponder];
    return YES;
}
```

Affichage de contenus web dans l'application

La classe `UIWebView` s'appuie sur le moteur de rendu de Safari (WebKit) pour permettre l'affichage de tout contenu pris en charge directement dans votre application (HTML, PDF, images, etc.).

Elle peut charger ces contenus depuis une URL externe, ou directement depuis la mémoire.

Exemple d'utilisation de la classe `UIWebView`

```
UIWebView *webView = [[UIWebView alloc]
initWithFrame:CGRectMake(0,0,320,460)];

// Chargement d'une URL réseau
[webView loadRequest:[NSURLRequest requestWithURL:[NSURL
urlWithString:@"http://www.google.fr"]]];

// Ou chargement d'un contenu statique
[webView loadHTMLString:@"Hello <b>World</b>" baseURL:@""];
```

Il est également possible de définir un délégué pour que votre contrôleur soit notifié lorsque l'utilisateur navigue dans les liens (voir à ce sujet la documentation de [UIWebViewDelegate](#)).

Animation des vues

Toutes les vues sont aplaties et dessinées dans une zone de dessin `CALayer`, issue du framework `Core Animation`. Il est ainsi possible de mettre en place très facilement des animations.

Pour cela, on crée un bloc d'animation dans lequel on change les propriétés qui doivent varier pendant l'animation, puis on demande au système de lancer l'animation.

Exemple d'animation d'une vue

```
- (void)loadView {
    UIView *myView = [[UIView alloc] initWithFrame:CGRectMake(0, 20, 320, 460)];
    myView.backgroundColor = [UIColor redColor];
    self.view = myView;
    [myView release];

    UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(0, 0, 320, 20)];
    label.text = @"Un texte qui bouge";
    self.movingLabel = label;
    [label release];

    [self.view addSubview:self.movingLabel];
}

- (void)viewDidAppear:(BOOL)animated
{
    [UIView beginAnimations:@"movingLabel" context:nil];
    [UIView setAnimationDuration:10.0];
    self.movingLabel.frame = CGRectMake(0, 440, 320, 20);
    [UIView commitAnimations];
}
```

Cette technique peut être utilisée pour animer la plupart des propriétés d'une vue : sa taille, son opacité, sa couleur, etc. Durant l'animation, votre application continue à s'exécuter et l'utilisateur peut toujours interagir avec elle.

Conclusion

Ce chapitre nous a permis de découvrir les vues qui sont les éléments avec lesquels vous composerez toutes les interfaces de votre application.

Il y a bien d'autres vues standard fournies par UIKit et de nombreuses possibilités de les adapter à votre besoin. Vous les découvrirez en étudiant les applications disponibles sur l'App Store et en parcourant la documentation.

Le prochain chapitre traite d'une vue extrêmement importante sur l'iPhone : les listes d'éléments.

10

Listes d'éléments

L'une des habitudes ergonomiques les plus ancrées chez les utilisateurs d'iPhone est la navigation dans une liste avec le doigt. Ce type de navigation est omniprésent dans les applications et il est normal que le SDK fournisse un maximum d'éléments pour faciliter l'implémentation de listes d'éléments.

Les deux types de listes

Comme nous l'avons vu au chapitre 5, il existe deux sortes de listes d'éléments : les listes simples et les listes groupées. Techniquement, il y a peu de différences pour le développeur.

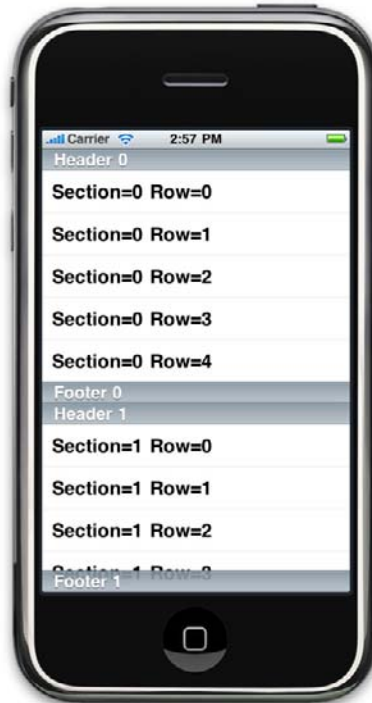
Dans les deux cas, les données à représenter sont séparées en sections puis en lignes.

Les listes simples

Les listes simples représentent des données qui occupent toute la largeur de l'écran. Les sections (si elles sont utilisées) sont représentées par des barres horizontales contenant les titres et pieds de section.

Figure 10-1

Une liste d'éléments en mode normal



Les listes groupées

Les listes groupées permettent de représenter différents groupes de données qui peuvent être hétérogènes.

Chaque section de la liste est représentée comme un bloc à bords arrondis différent.

Créer une tableView

La classe `UITableView` prend en charge l'affichage d'une liste d'éléments. On crée une instance à l'aide de l'initialisateur `initWithFrame:style:` qui prend en premier paramètre la frame dans laquelle dessiner la liste et en deuxième paramètre le style de la vue : `UITableViewStylePlain` ou `UITableViewStyleGrouped`.

Figure 10-2
Une liste d'éléments en mode groupé



Fournir des données à une tableView

Pour obtenir les données à afficher, la liste d'éléments utilise une source de données. Le développeur doit fournir un objet implémentant le protocole `UITableViewDataSource`.

Indiquer le nombre de lignes

La première méthode à implémenter est `tableView:numberOfRowsInSection:`. Elle prend en paramètre la table et le numéro de la section. Elle renvoie le nombre d'éléments contenus dans cette section. Si vous n'utilisez qu'une seule section, il s'agit du nombre d'éléments dans la liste.

```
- (NSInteger)tableView:(UITableView *)tableView  
numberOfRowsInSection:(NSInteger)section  
{  
    return 5;  
}
```

Dans le cas où il y aurait plusieurs sections, il faut indiquer le nombre de sections à l'aide de la méthode : `numberOfSectionsInTableView:` (par défaut, il y a une seule section).

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 3;
}
```

Afficher des données

Lorsque la liste d'éléments a besoin d'afficher des données, elle appelle la méthode `tableView:cellForRowAtIndexPath:` de sa source de données. Cette méthode doit renvoyer un objet `UITableViewCell` qui est une vue représentant une des cellules.

Pour réduire la consommation mémoire et améliorer la fluidité du défilement, un mécanisme de recyclage des cellules est proposé et doit impérativement être utilisé. En appelant la méthode `dequeueReusableCellWithIdentifier:` de l'objet `UITableView`, on récupère une cellule qui peut être réutilisée ou `nil` si aucune n'est disponible.

Si on a pu récupérer une cellule, on se contente de redéfinir son contenu pour refléter la ligne à afficher. Sinon, on crée une nouvelle cellule qui sera recyclée plus tard et on définit son contenu.

```
#define CELL_IDENTIFIER @"myCellIdentifier"

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CELL_IDENTIFIER];

    // Si nécessaire, on crée une nouvelle cellule
    if (cell == nil)
    {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CELL_IDENTIFIER];
        [cell autorelease];
    }

    // Dans tous les cas, on définit son contenu
    cell.textLabel.text = [NSString stringWithFormat:@"Section=%u Row=%u",
        indexPath.section, indexPath.row];

    return cell;
}
```

REMARQUE Les styles de cellule

Il existe plusieurs types de cellules prédéfinis que vous pouvez utiliser.

- `UITableViewCellStyleDefault` : permet d'afficher une cellule contenant un seul label et une image à gauche. C'est le style par défaut.
- `UITableViewCellStyleValue1` : affiche une cellule avec deux libellés, un à gauche et un à droite. Le texte de gauche est noir et aligné à gauche, tandis que le texte de droite est bleu et aligné à droite. C'est le style utilisé dans l'application Réglages.
- `UITableViewCellStyleValue2` : affiche une cellule avec deux libellés, un texte bleu dans la partie gauche de la cellule qui est aligné à droite et un texte noir aligné à gauche dans la partie droite de la cellule. C'est le style utilisé dans l'application Téléphone.
- `UITableViewCellSubtitle` : affiche une cellule avec deux libellés. Un texte noir aligné à gauche sur la première ligne et un texte gris légèrement plus petit sur la deuxième ligne. C'est le style utilisé dans l'application iPod.

Vous pouvez adapter légèrement chacun de ces styles en modifiant la police, la couleur du texte, l'alignement, etc. grâce aux propriétés `textLabel` et `detailTextLabel` qui donnent un accès direct à l'objet `UILabel` utilisé pour afficher les deux textes.

Si les styles par défaut ne suffisent pas à votre besoin, vous devez personnaliser la cellule. Nous verrons un peu plus loin dans ce chapitre quelles sont les différentes techniques possibles pour contrôler de manière beaucoup plus fine le contenu de la cellule et afficher des contenus riches (plusieurs lignes de texte, images, etc.).

Définir les en-têtes et pieds de section

Afin de fournir les textes à afficher dans les en-têtes et pieds de section, la source de données peut implémenter les méthodes :

- `tableView:titleForHeaderInSection:` et
- `tableView:titleForFooterInSection:`

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section
{
    return [NSString stringWithFormat:@"Header %i", section];
}

- (NSString *)tableView:(UITableView *)tableView
titleForFooterInSection:(NSInteger)section
{
    return [NSString stringWithFormat:@"Footer %i", section];
}
```

Réagir aux actions sur la liste

Le développeur peut fournir un délégué à la liste d'éléments. Ce dernier doit implémenter le protocole `UITableViewDelegate`.

Le délégué permet une personnalisation plus poussée de l'affichage de la liste et aussi de réagir lorsque l'utilisateur sélectionne un élément, modifie une ligne, etc.

Sélection d'un élément

Lorsqu'un élément de la liste est sélectionné par l'utilisateur, la méthode `tableView:didSelectRowAtIndexPath:` est appelée.

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSLog(@"Sélection d'un élément. Section=%u Ligne=%u",
indexPath.section, indexPath.row);
}
```

Édition dans une tableView

Le mode édition d'une liste permet à l'utilisateur de supprimer des éléments, d'en ajouter ou de les réorganiser.

Pour activer le mode édition, on appelle la méthode `setEditing:animated:` sur l'objet `tableView`. Il faut ensuite implémenter les méthodes correspondantes du délégué pour effectivement supprimer les éléments et mettre la liste à jour.

Techniques pour afficher des cellules personnalisées

Nous avons vu comment afficher du texte dans une liste d'éléments. Le plus souvent, vous voudrez afficher des contenus plus riches, contenant des images, plusieurs lignes de texte, etc.

Il existe trois techniques différentes pour fournir des cellules propres à votre application. Ces trois techniques ont chacune leurs avantages et leurs inconvénients, le choix de la technique à utiliser dépendra de votre projet.

Dans tous les cas, la hauteur de la cellule devra être indiquée à la table en définissant la propriété `rowHeight` (il est également possible d'avoir des cellules de tailles différentes grâce à la méthode `tableView:heightForRowAtIndexPath:` du délégué).

Composition de la cellule

La classe `UITableViewCell` hérite de `UIView`, et on peut donc lui ajouter une hiérarchie de vue plus complexe.

Exemple de construction d'une cellule plus complexe

```
#define CELL_IDENTIFIER @"myCellIdentifler"

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CELL_IDENTIFIER];

    UILabel *firstTextLabel;
    UILabel *secondTextLabel;
    UIImageView *imageView;

    if (cell == nil)
    {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
➤ reuseIdentifier:CELL_IDENTIFIER];
        [cell autorelease];

        firstTextLabel = [[UILabel alloc] initWithFrame:CGRectMake(50, 2, 200, 20)];
        firstTextLabel.tag = 10;
        [cell addSubview:firstTextLabel];
        [firstTextLabel release];

        secondTextLabel = [[UILabel alloc] initWithFrame:CGRectMake(50, 20, 200, 20)];
        secondTextLabel.tag = 11;
        [cell addSubview:secondTextLabel];
        [secondTextLabel release];

        imageView = [[UIImageView alloc] initWithFrame:CGRectMake(5, 5, 40, 30)];
        imageView.tag = 12;
        [cell addSubview:imageView];
        [imageView release];
    }
    else {
        firstTextLabel = (UILabel *)[cell viewWithTag:10];
        secondTextLabel = (UILabel *)[cell viewWithTag:11];
        imageView = (UIImageView *)[cell viewWithTag:12];
    }
}
```

```
firstTextLabel.text = [NSString stringWithFormat:@"Section: %u",
    ↳ indexPath.section];
secondTextLabel.text = [NSString stringWithFormat:@"Ligne: %u", indexPath.row];
imageView.backgroundColor = [UIColor redColor];

return cell;
}
```

ASTUCE Utilisation des identifiants de vue

Toutes les vues peuvent recevoir un identifiant fourni par le développeur via la propriété `tag`. On peut ensuite retrouver cette vue dans une hiérarchie en appelant la méthode `viewWithTag:`. Cette technique est utilisée dans cet exemple pour retrouver les sous-vues qui ont été ajoutées manuellement lorsqu'on recycle la cellule.

Utilisation d'Interface Builder pour concevoir les cellules

Il est possible d'utiliser Interface Builder pour concevoir les cellules. Vous devez pour cela :

- 1 Créer un nouveau fichier XIB selon le modèle « Empty XIB ».
- 2 Ajouter un objet `UITableViewCell` dans le XIB.
- 3 Indiquer que le type du File's Owner est `UIViewController`.
- 4 Relier la cellule à la propriété `view` du File's Owner.

Il est ensuite possible d'instancier la cellule en créant un contrôleur de vue que nous n'utiliserons pas.

Exemple d'utilisation d'une cellule construite avec Interface Builder

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CELL_IDENTIFIER];

    if (cell == nil)
    {
        UIViewController *cellFactoryViewController = [[UIViewController alloc]
        initWithNibName:@"Cell" bundle:nil];
        cell = (UITableViewCell*)cellFactoryViewController.view;
        [cell retain];
        [cellFactoryViewController release];
        [cell autorelease];
    }

    return cell;
}
```

Une fois la cellule créée à partir du fichier XIB, vous aurez certainement besoin d'accéder aux sous-vues pour définir leurs valeurs : c'est possible en utilisant des tags comme on l'a fait dans l'exemple précédent (les tags peuvent être définis dans Interface Builder).

Une solution plus élégante est de créer une nouvelle sous-classe de `UITableViewCell`, de lui ajouter des propriétés marquées comme `IBOutlet` et de définir cette nouvelle classe comme étant le type de l'objet cellule. Vous pourrez alors faire des liens entre les éléments de la cellule et les propriétés.

Dessiner manuellement le contenu de la cellule

La dernière solution ne repose pas sur un assemblage de vues mais sur une seule vue, dérivant de `UITableViewCell` et qui a en charge le dessin de l'ensemble de la cellule.

Pour implémenter cette solution, il faut utiliser les API de dessin 2D pour dessiner directement le texte et les images de la cellule, ce qui sort du cadre de ce livre, mais c'est de loin la solution la plus performante lorsque vous devez dessiner des cellules complexes contenant de nombreuses informations.

POUR APPROFONDIR Dessiner manuellement le contenu de la cellule

Le guide de programmation des listes d'éléments (*TableView Programming Guide*) fournit des exemples d'implémentation pour les trois techniques présentées ici.

Un exemple complet

En résumé, la mise en place d'une liste d'éléments nécessite de :

- 1 Créer une instance de `UITableView` et l'ajouter dans la vue d'un contrôleur.
- 2 Ajouter le protocole `UITableViewSource` au contrôleur et implémenter les méthodes `tableView:numberOfRowsInSection:` et `numberOfSectionsInTableView:` pour indiquer le nombre de sections et de cellules.
- 3 Implémenter la méthode `tableView:cellForRowAtIndexPath:` de la source de données pour fournir les cellules.
- 4 Éventuellement, ajouter le protocole `UITableViewDelegate` et implémenter la méthode `tableView:didSelectRowAtIndexPath:` pour déclencher une action lorsque l'utilisateur sélectionne un élément.

Le code ci-après est un exemple complet d'un contrôleur de vue permettant de reproduire l'illustration de début de chapitre. Il peut servir de base à vos expérimentations avec les listes d'éléments.

Un exemple complet de contrôleur avec une liste d'éléments : `TableViewCellController.h`

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

@interface TableViewController :
    UIViewController<UITableViewDelegate, UITableViewDataSource> {
}

@end
```

Un exemple complet de contrôleur avec une liste d'éléments : `TableViewCellController.m`

```
#import "TableViewCellController.h"

@implementation TableViewController

- (void)loadView {
    UIView *view = [[UIView alloc] initWithFrame:CGRectMake(0, 20, 320, 460)];
    self.view = view;
    [view release];

    UITableView *tableView = [[UITableView alloc]
        initWithFrame:CGRectMake(0, 0, 320, 460) style:UITableViewStylePlain];
    tableView.dataSource = self;
    tableView.delegate = self;

    [self.view addSubview:tableView];
    [tableView release];
}

- (void)dealloc {
    [super dealloc];
}

#pragma mark UITableViewDataSource

#define CELL_IDENTIFIER @"myCellIdentifier"

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CELL_IDENTIFIER];
```

```
if (cell == nil)
{
    cell = (UITableViewCell*) [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:CELL_IDENTIFIER];
    [cell autorelease];
}

cell.textLabel.text = [NSString stringWithFormat:@"Section=%u Row=%u",
    indexPath.section, indexPath.row];

return cell;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 3;
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return 5;
}

- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
{
    return [NSString stringWithFormat:@"Header %i", section];
}

- (NSString *)tableView:(UITableView *)tableView
    titleForFooterInSection:(NSInteger)section
{
    return [NSString stringWithFormat:@"Footer %i", section];
}

#pragma mark UITableViewDelegate

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSLog(@"Sélection d'un élément. Section=%i Ligne=%i", indexPath.section,
        indexPath.row);
}

@end
```

Conclusion

Dans ce chapitre, nous avons vu comment utiliser la classe `UITableView` pour créer des listes d'éléments de différents types et comment y intégrer des cellules spécifiques en fonction des besoins de votre application.

Cette classe est extrêmement importante car les listes sont souvent le moyen le plus efficace pour présenter des informations, et les utilisateurs les comprennent très bien. Bientôt, vous réimplémenterez tous les exemples de ce chapitre les yeux fermés !

QUATRIÈME PARTIE

La manipulation des données

Cette partie présente les différentes techniques à utiliser pour lire et enregistrer des données, ainsi que pour permettre à votre application de communiquer avec l'extérieur.

La sérialisation des classes de base (listes et dictionnaires) en XML ou en JSON sera présentée au **chapitre 11**, ainsi que l'utilisation du mécanisme des préférences utilisateur et les techniques de lecture d'un flux XML quelconque.

Le **chapitre 12** montre comment combiner ces techniques avec des appels réseau, et en particulier comment faire en sorte que l'application reste réactive même quand des appels réseau prennent plusieurs secondes.

Le framework CoreData, une nouveauté de l'iPhone OS 3 est présenté dans le **chapitre 13**. C'est un framework complet de mapping objet-relationnel pour l'iPhone et incontestablement le moyen le plus efficace d'enregistrer et de parcourir des volumes importants de données métier.

L'utilisation de contenus multimédias est couverte par le **chapitre 14** : lecture de sons, utilisation de la bibliothèque iPod de l'utilisateur, de la caméra ou encore lecture de vidéos.

Cette partie se termine par un **chapitre 15** sur le mécanisme des notifications qui est également une nouveauté de l'iPhone OS 3 et qui permet de rester en contact avec l'utilisateur, même quand l'application est éteinte. La mise en place des notifications est présentée, de manière détaillée, avec des exemples en PHP pour la partie serveur qui pourront facilement être adaptés dans n'importe quel langage.

11

Lire et enregistrer des données

Comme toute application informatique, vos applications iPhone ont besoin de lire des données et de pouvoir en enregistrer. Le SDK iPhone met à votre disposition un ensemble de méthodes des plus basiques (fopen, socket, etc.) aux plus évoluées (parseur XML, Core Data, etc.).

Nous nous attacherons dans ce chapitre à décrire les mécanismes les plus efficaces et les plus utilisés dans des applications iPhone.

C'est au chapitre suivant « Communiquer avec l'extérieur » qu'on montrera comment combiner ces techniques avec des accès à des serveurs distants.

Les préférences utilisateur

Cocoa met à disposition des développeurs un mécanisme permettant d'enregistrer des combinaisons clé-valeur et de les recharger très simplement. Ce mécanisme est implémenté par la classe `NSUserDefaults` et on parle des « préférences utilisateur » car il est très fortement recommandé de l'utiliser pour enregistrer les choix de l'utilisateur, l'état de l'application quand il la ferme, etc.

Obtenir une instance des préférences utilisateur

La classe `NSUserDefaults` propose une méthode statique `+standardUserDefaults` qui permet d'obtenir un pointeur vers un singleton contenant les préférences de l'utilisateur.

Cet objet se synchronise automatiquement avec un fichier de propriétés sur le disque, ce qui permet de ne pas avoir à se soucier de quand enregistrer les données.

Enregistrer une valeur dans les préférences

La classe `NSUserDefaults` permet d'associer à une clé un objet de type `NSString`, `NSDate`, `NSNumber`, `NSData`, `NSArray` ou `NSDictionary`.

Pour enregistrer un objet dans les préférences, on appelle simplement la méthode `setObject:ForKey:`.

Utilisation des préférences utilisateur pour enregistrer le nom de l'utilisateur

```
- (void) viewWillAppear:(BOOL) animated
{
    [super viewWillAppear:animated];

    NSString *username = self.usernameTextField.text;
    [[NSUserDefaults standardUserDefaults] setObject:username
                                             forKey:@"username"];
}
```

Il est également possible d'enregistrer la valeur d'un des types primitifs via les méthodes `setBool:forKey:`, `setFloat:forKey:` et `setInteger:forKey:`.

En enregistrant des listes ou des dictionnaires comme valeur d'une clé, on peut très simplement enregistrer des structures de données complexes. Vous pouvez ainsi utiliser les préférences pour enregistrer l'état d'un contrôleur de barre d'onglets et les paramètres permettant de recréer les mêmes onglets dans le même ordre.

Lire les valeurs des préférences

Pour chacun des types de données qui peuvent être enregistrés, une méthode est mise à disposition qui renvoie la valeur associée à la clé passée en paramètre.

Il est de la responsabilité du développeur de savoir quel est le type de donnée qui a été enregistré sous une clé. Si aucune donnée n'a été enregistrée, ou si la donnée ne correspond pas au type demandé, la valeur `nil` est renvoyée (pour les types primitifs, une valeur prédéterminée est renvoyée : `NO` ou `0`).

Les méthodes mises à disposition pour récupérer des objets sont : `arrayForKey:`, `dataForKey:`, `objectForKey:` et `stringForKey:`. Pour récupérer des types primitifs, il faut utiliser `boolForKey:`, `floatForKey:` et `integerForKey:`.

Lecture du nom d'utilisateur dans les préférences

```
- (void) viewDidLoad
{
    [super viewDidLoad];

    self.usernameTextField.text = [[NSUserDefaults standardUserDefaults]
    stringForKey:@"username"];
}
```

Permettre à l'utilisateur de modifier directement les préférences

Les mécanismes évoqués ici permettent de manipuler des paires clés-valeurs et de les enregistrer sur le disque. C'est le développeur qui met en place l'interface permettant d'éditer une préférence et de la modifier.

Ce mécanisme est pertinent pour de nombreuses préférences qui sont implicites : l'application enregistre les habitudes de l'utilisateur.

Dans certains cas, on voudra pourtant donner la possibilité à l'utilisateur de régler explicitement des préférences. L'iPhone propose pour cela un mécanisme qui permet d'ajouter dans l'application Réglages, une page pour votre application.

Figure 11-1
Écran de réglage des
préférences d'une application



Pour chaque page de réglage, votre application fournit un fichier de propriétés qui indique le titre de la propriété réglable et son type. L'application Réglages met en œuvre l'interface (vous n'avez aucun code à écrire) et enregistre les valeurs dans les préférences utilisateur, en s'appuyant également sur la classe `NSUserDefaults`. On y accède exactement comme pour des préférences que vous auriez enregistrées depuis votre code.

APPROFONDIR Mise en place de pages de préférences utilisateur

La mise en place de ces pages de réglages est décrite dans le guide *iPhone Application Programming Guide*.

Les fichiers de propriétés

Les fichiers de propriétés permettent d'enregistrer au format XML les types standard de l'iPhone (`NSString`, `NSDate`, `NSArray`, `NSDictionary`, etc.). En combinant les types de base avec les listes et les dictionnaires, on peut créer des structures de données complexes. Cocoa met à disposition des développeurs un ensemble de méthodes permettant de lire et d'écrire très facilement ces données.

Ce format étant très facile à éditer depuis Xcode ou même à générer depuis une application web (PHP, Java, etc.), il est souvent bien plus facile de l'utiliser plutôt que de réécrire le code nécessaire pour charger un fichier XML qui s'appuierait sur un format spécifique.

Le format plist

Un fichier plist est un fichier XML qui respecte une DTD stricte fournie par Apple. Xcode intègre un éditeur qui permet d'éditer très facilement ces fichiers. Pour créer un nouveau fichier, utiliser le menu *File > New File > Other > Property List*.

L'exemple ci-après présente un fichier de propriétés utilisé pour enregistrer une liste de contacts. Ce type de fichier pourrait être envoyé par un serveur web à une application iPhone, qui le chargerait en mémoire pour permettre à l'utilisateur de naviguer dans des contacts propres à l'application.

Figure 11-2
Exemple de fichier
plist édité dans Xcode.

Key	Type	Value
Root	Dictionary	(3 items)
URLSource	String	http://www.monapplication.com/carnetadresses.plist
DerniereModification	Date	23 déc. 09 20:30:00
Contacts	Array	(2 items)
Item 1	Dictionary	(3 items)
Nom	String	M. Dupont
Adresse	String	45 avenue de la Marquise
Telephone	String	06 123 456 42
Item 2	Dictionary	(3 items)
Nom	String	M. Durand
Adresse	String	42 rue de la Paix
Telephone	String	06 12345678

Code XML généré par Xcode correspondant au même fichier de propriétés

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>URLSource</key>
  <string>
    http://www.monapplication.com/carnetadresses.plist
  </string>
  <key>DerniereModification</key>
  <date>2009-12-23T19:30:00Z</date>
  <key>Contacts</key>
  <array>
    <dict>
      <key>Nom</key>
      <string>M. Dupont</string>
      <key>Adresse</key>
      <string>45 avenue de la Marquise</string>
      <key>Telephone</key>
      <string>06 123 456 42</string>
    </dict>
    <dict>
      <key>Nom</key>
      <string>M. Durand</string>
      <key>Adresse</key>
      <string>42 rue de la Paix</string>
      <key>Telephone</key>
      <string>06 12345678</string>
    </dict>
  </array>
</dict>
</plist>
```

On voit que le code généré par Xcode est très facilement compréhensible et pourra être adapté ou généré par une application web écrite en PHP, Java, etc.

Lire un fichier de données plist

Dans la très grande majorité des cas, la racine de votre fichier plist sera un dictionnaire ou une liste.

Dans ce cas, vous pouvez utiliser la méthode `initWithContentsOfFile:` de la classe `NSDictionary` ou `NSArray` pour lire le fichier et renvoyer une instance initialisée avec son contenu.

Exemple d'utilisation de la méthode `initWithContentsOfFile:` pour lire un fichier plist en mémoire

```
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Exemple"
ofType:@"plist"];
NSDictionary *dict = [[NSDictionary alloc]
initWithContentsOfFile:filePath];
```

ASTUCE Obtenir le chemin complet d'un fichier de ressource

Dans cet exemple, on récupère le chemin vers le fichier en utilisant la méthode `pathForResource ofType:` du bundle principal de l'application. Elle prend en paramètre le nom du fichier recherché et son extension.

C'est la méthode recommandée pour obtenir le chemin complet vers un fichier de ressource de votre application.

Vous pouvez ensuite parcourir les données normalement, mais attention, les instances renvoyées lors du chargement d'un fichier de propriétés sont toujours non mutables, elles ne peuvent pas être modifiées directement. Pour les modifier, vous devez créer un objet mutable (`NSMutableArray` ou `NSMutableDictionary`) à partir de l'objet non mutable.

Écrire un fichier de données plist

Pour enregistrer des données, les classes `NSDictionary` et `NSArray` proposent la méthode `writeToFile:atomically:` qui permet d'enregistrer le contenu de l'instance dans un fichier au format plist.

Le premier paramètre est le chemin vers le fichier. Le deuxième paramètre est un booléen qui indique si le fichier doit être écrit de manière atomique. Si ce booléen est vrai, le fichier sera écrit dans un fichier temporaire puis renommé avec le nom du

fichier de destination. Cela permet de garantir qu'en cas de plantage, le fichier existant ne sera pas corrompu.

Cette méthode renvoie un booléen indiquant si l'écriture s'est correctement déroulée ou pas. Une des causes d'échec peut être que l'arbre de données contient un objet qui n'est pas directement pris en charge par les propriétés.

ATTENTION **Savoir où enregistrer vos données**

Lorsque votre application iPhone est compilée, l'ensemble de l'application est signée avec votre clé de développeur et ne peut plus être modifiée par la suite. Ainsi, vous ne pouvez pas, depuis votre application, écraser un fichier de ressource fourni avec votre application, car cela modifierait sa signature électronique.

À la place, vous devez enregistrer la version mise à jour de ce fichier dans le répertoire des données utilisateurs. Vous pouvez obtenir le chemin vers cet emplacement en utilisant les deux appels suivants :

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                    NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

Le contenu de ce répertoire est sauvegardé par iTunes quand l'utilisateur synchronise son iPhone (et restauré si l'utilisateur restaure son iPhone à partir d'une sauvegarde).

Pour enregistrer des contenus qui ne soient pas sauvegardés, vous pouvez utiliser le répertoire temporaire :

```
NSString *tempDirectory = NSTemporaryDirectory();
```

Le format de données JSON

Le format de données JSON (*JavaScript Object Notation*) est un format très compact utilisé initialement pour sérialiser des données JavaScript. Il est maintenant utilisé pour échanger des données entre le navigateur web et le serveur dans la plupart des applications Ajax.

De la même façon que les fichiers de propriétés, le format JSON permet de sérialiser des ensembles de paires/valeurs ; ces dernières peuvent être des chaînes, des booléens, des nombres, des listes ou des dictionnaires (mais les objets date ne sont pas supportés directement et doivent être convertis en nombre ou en chaîne).

Dans le cadre des applications iPhone, ce format de fichier présente plusieurs avantages intéressants :

- 1 C'est un format très compact qui occupe moins d'espace en octets, les temps d'aller-retour client-serveur sont donc accélérés. Ce point peut sembler négligeable, mais l'expérience montre que le poids ajouté par le langage XML peut rapidement représenter plus de la moitié du volume des échanges, et la différence est très sensible surtout en EDGE ou en 3G ;

- 2 C'est un format standard, et pour lequel il existe déjà de nombreuses bibliothèques dans tous les langages ;
- 3 À l'aide de bibliothèques tierces, il est extrêmement facile de générer et de lire du JSON depuis une application iPhone.

Tirer partie de JSON dans vos applications

Bien qu'Apple apprécie manifestement ce format de données (il est utilisé pour l'envoi des notifications), il n'est pas encore supporté par les API standards et vous devez intégrer à votre projet une bibliothèque externe pour que le format JSON soit pris en charge.

Vous trouverez un framework JSON pour l'iPhone en open source (licence BSD) sur le site : <http://code.google.com/p/json-framework/>.

Les instructions d'installation du site expliquent comment l'ajouter à votre projet d'application comme une bibliothèque externe.

Un autre moyen plus simple, mais moins élégant, consiste à ajouter à votre projet l'ensemble des fichiers sources de la librairie.

Ce framework JSON propose des extensions aux objets de base Objective-C (il utilise pour cela le mécanisme des catégories) pour leur ajouter la possibilité de se sérialiser ou de se restaurer en JSON.

Lire des données JSON

Une fois le fichier d'en-tête du framework importé (`#import "JSON.h"`), il est possible, depuis n'importe quelle chaîne de caractères, d'obtenir l'objet dictionnaire ou liste correspondant :

```
NSString *jsonData = @"{\"nom\":\"Jobs\",\"adresse\":\"1 Infinite Loop\", \"num\":\"555-1234-5678\"}";  
NSDictionary *dict = [jsonData JSONValue];
```

Enregistrer des données JSON

Il est possible tout aussi simplement de sérialiser un dictionnaire ou une liste dans sa représentation JSON à l'aide de la méthode `-JSONRepresentation`.

```
NSString *jsonData = [dict JSONRepresentation];
```

Manipuler des données XML

Le SDK propose deux mécanismes pour lire et interpréter des fichiers XML. Vous pouvez choisir d'utiliser directement la bibliothèque C libxml qui est fournie avec le SDK ou vous pouvez vous appuyer sur la classe `NSXMLParser` qui propose une interface Objective-C à libxml.

Dans les deux cas, le mode de lecture des données est proche de l'API SAX. Il n'existe pas de mécanisme standard pour lire les données selon l'API DOM.

RAPPEL SAX et DOM

Il existe deux API standard pour lire des fichiers XML. Celles-ci ont été déclinées dans presque tous les langages de programmation.

L'API DOM (*Document Object Model*) lit l'ensemble du document XML et le charge en mémoire sous la forme d'un arbre. Il est ensuite possible de parcourir l'arbre en utilisant des boucles, des fonctions récursives, etc. Cette API est souvent plus facile à utiliser pour le développeur, mais elle a comme principal désavantage de charger l'intégralité des données en mémoire, ce qui n'est pas toujours possible.

L'API SAX (*Simple API for XML*) s'appuie sur des événements qui sont envoyés au fur et à mesure de la lecture du document (balise ouvrante, balise fermante, contenu texte, etc.). Elle est plus efficace en termes de mémoire et permet de commencer à traiter un document incomplet, mais elle est plus complexe à utiliser.

Nous ne détaillerons ici que l'utilisation simple de `NSXMLParser`. Sachez que des exemples de l'utilisation de libxml sont fournis dans le SDK, en particulier dans le projet exemple `XMLPerformance`, qui montre comment lire des données du réseau et les parser à la volée, sans attendre que tous les contenus soient chargés.

Création d'un parseur XML

Pour utiliser la classe `NSXMLParser`, vous devez lui fournir un délégué qui sera appelé à chaque fois qu'un événement XML se produit (quand il rencontre une balise ouvrante, une balise fermante, un contenu texte, etc.). C'est ce délégué qui reconstruira en mémoire, avec des objets métiers, les données décrites dans le XML.

Vous pouvez également définir plusieurs options indiquant comment gérer les espaces de noms et les entités.

Il est recommandé de créer une classe dédiée au parsing de chaque type de flux que gère votre application. Cette classe sera le délégué de `NSXMLParser`, et elle contiendra une méthode `-parseData:` qui sera chargée de créer l'objet parseur, de lui indiquer que le délégué est `self`, et de lancer le traitement.

L'exemple ci-après montre comment implémenter cette fonction, en indiquant au parseur qu'on ne souhaite pas gérer les espaces de noms et les entités.

Exemple d'utilisation de NSXMLParser

```
- (void) parseData:(NSData*) data
    NSXMLParser *xmlParser = [[NSXMLParser alloc] initWithData:data];
    [xmlParser setDelegate:self];
    [xmlParser setShouldProcessNamespaces:NO];
    [xmlParser setShouldReportNamespacePrefixes:NO];
    [xmlParser setShouldResolveExternalEntities:NO];

    [xmlParser parse];

    NSError *parseError = [xmlParser parseError];
    if (parseError) {
        BKLog(@"XmlParser - Error parsing data: %@", [parseError
        localizedDescription]);
    }

    [xmlParser release];
}
```

Lorsqu'on appelle la méthode `-parse:`, le traitement est lancé, et le parseur va lire le fichier, appeler les méthodes du délégué pour chaque événement et indiquer ensuite si tout c'est bien passé grâce à la méthode `-parseError`.

Gérer les événements XML

Pour chacun des événements XML que vous souhaitez traiter, vous devez implémenter une fonction dans votre délégué. Ces fonctions vous permettront de reconstruire les données métier en mémoire au fur et à mesure.

Les objets métier ou la valeur de leurs propriétés sont stockés dans des variables d'instance du parseur tant que l'objet n'a pas été entièrement traité.

Dans les paragraphes suivants, nous décrirons l'implémentation d'un parseur XML très simple, qui est destiné à lire le flux suivant pour reconstituer en mémoire une liste d'objets `Contact`.

Exemple de données XML à interpréter

```
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
  <contact id="001" type="A">
    <telephone>0123456789</telephone>
    <adresse>xxx</adresse>
  </contact>
  ...
</contacts>
```

Début du document

Lorsque le parseur rencontre le début du document XML, il appelle la méthode `parserDidStartDocument:`. Vous pouvez utiliser cet événement pour initialiser vos variables d'instance, par exemple en créant une nouvelle liste pour contenir les objets qui vont être lus ou la réinitialiser.

```
- (void)parserDidStartDocument:(NSXMLParser *)parser
{
    if (contacts == nil)
        contacts = [[NSMutableArray alloc] initWithCapacity:0];
    else
        [contacts removeAllObjects];
}
```

Début d'un élément

Le parseur appelle la méthode `parser:didStartElement:` quand il rencontre un nouvel élément XML.

Pour chaque élément rencontré, vous voudrez probablement suivre une des deux stratégies suivantes :

- Cet élément correspond à un objet métier de votre application, il faut donc créer une nouvelle instance de l'objet métier et la garder dans une variable d'instance du parseur.
- Cet élément correspond à un attribut d'un de vos objets métier ; dans ce cas, il faut préparer une variable d'instance dans laquelle on stockera le contenu de l'élément.

Cette méthode reçoit également en paramètre un dictionnaire contenant la liste et la valeur de tous les attributs de l'élément.

Traitement d'un élément par le délégué du parseur XML

```

- (void)parser:(NSXMLParser *)parser didStartElement:(NSString
*)elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName attributes:(NSDictionary
*)attributeDict
{
    // Si l'élément lu correspond à un objet métier
    if ([elementName isEqualToString:@"contact"]) {
        // On crée une instance de l'objet métier et on la stocke
        // dans une variable d'instance
        currentContact = [[Contact alloc] init];

        // On ajoute l'objet métier à la liste de tous les objets
        // lus par le parseur
        [contacts addObject:currentContact];

        // Lecture des attributs de l'élément
        currentContact.identifiant = [attributeDict valueForKey:@"id"];
        currentContact.type = [attributeDict valueForKey:@"type"];

        // Le traitement est fini pour cet élément.
        [currentContact release];
        return;
    }

    // Si le texte de l'élément contient une propriété de
    // l'objet métier, alors on prépare une variable
    // temporaire pour enregistrer son contenu
    if ([elementName isEqualToString:@"telephone"]) {
        currentProperty = [NSMutableString string];
    }
    else if ([elementName isEqualToString:@"adresse"]) {
        currentProperty = [NSMutableString string];
    }
}

```

Récupérer le contenu texte des éléments

Lorsque le parseur rencontre du texte, il appelle la méthode `parser:foundCharacters:`. Il suffit alors d'enregistrer les caractères trouvés dans la variable temporaire `currentProperty` que nous avons préparée au paragraphe précédent.

Traitement de texte par le parseur

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    if (currentProperty) {
        [currentProperty appendString:string];
    }
}
```

Repérer la fermeture d'un élément et enregistrer son contenu

Lorsque le parseur rencontre une balise fermante, il appelle la méthode `parser:didEndElement:` en lui passant en paramètre le nom de la balise.

Si cette balise contenait du texte intéressant pour nous, nous le récupérerons grâce à la variable d'instance `currentProperty`.

Implémentation de la méthode `parser:didEndElement:`

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)
elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName
{
    if ([elementName isEqualToString:@"adresse"]) {
        [currentContact setAdresse:currentProperty];
    } else if ([elementName isEqualToString:@"telephone"]) {
        [currentContact setTelephone:currentProperty];
    }
}
```

Conclusion

Dans ce chapitre, nous avons vu différentes méthodes pour lire et enregistrer des données.

Pour enregistrer les préférences utilisateur, l'état de l'application lors de la fermeture, etc., vous disposez de la très efficace classe `NSUserDefaults`.

Si vous souhaitez distribuer votre application avec un référentiel de données, le format `plist` présente l'avantage d'être très facile à éditer dans Xcode.

Pour tout échange avec un service web, essayez toujours d'utiliser le format JSON qui a l'avantage non négligeable d'être très léger.

Enfin, si vous devez lire des données XML, vous pouvez le faire à l'aide de `NSXMLParser` ou de la `libxml` directement – mais attention, c'est plus complexe pour le développeur.

12

Communiquer avec l'extérieur

La plupart des applications iPhone communiquent avec un service web, que ce soit pour récupérer des informations à présenter à l'utilisateur, mettre à jour un référentiel, ou proposer un envoi de données vers un service communautaire.

Le très haut niveau de connectivité des applications est sans aucun doute un des atouts de l'iPhone. Aussi réussir à combiner fluidité de l'application, rapidité d'apparition des données et réactivité de l'interface est-il un défi incontournable pour le développeur.

Nous verrons dans ce chapitre comment lancer des appels réseau vers des serveurs distants, récupérer et traiter la réponse, le tout sans ralentir l'interface. En termes techniques, il s'agira de connexions synchrones et asynchrones ; surtout nous parlerons des threads et de leur gestion dans une application iPhone.

RAPPEL Connexions synchrones et asynchrones

On parle de requêtes synchrones quand la méthode utilisée pour faire un appel réseau est bloquante, et attend d'avoir la réponse avant de repasser le contrôle à l'utilisateur. C'est généralement le mode le plus simple pour le développeur.

Une requête asynchrone rend immédiatement la main au développeur et utilise des méthodes callback pour transmettre des informations sur le statut de la requête et les données reçues.

Premiers appels réseau synchrones

La classe `NSURLRequest` est la base des API Cocoa pour exécuter des requêtes distantes. Elle propose, pour exécuter une requête et attendre la réponse du serveur, la méthode statique `sendSynchronousRequest:returningResponse:error:`.

Cette méthode prend en paramètre un objet `NSURLRequest` qu'on construit à partir de l'URL cible et deux pointeurs sur des pointeurs. Le premier pointeur permet d'indiquer une variable qui pointerait après l'appel vers un objet `NSURLResponse` contenant la réponse envoyée par le serveur. Le deuxième pointeur, pointerait après l'appel vers un objet `NSError` ou aura la valeur `nil` s'il n'y a pas eu d'erreur.

L'utilisation de la classe est ensuite très simple.

Exécution d'une requête réseau synchrone

```
NSURLRequest *request = [NSURLRequest requestWithURL:@"http://
www.myserver.com/xxx"];

NSURLResponse *response;
NSError *error;

NSData *data = [NSURLConnection sendSynchronousRequest:request
returningResponse:&response error:&error];
if (data != nil)
{
    NSLog(@"Requête HTTP réussie: %@", url);
}
else {
    if (error != nil)
        NSLog(@"Echec lors de la requête HTTP: %@ (%@)", url,
            [error localizedDescription]);
    else
        NSLog(@"Echec lors de la requête HTTP: %@", url);
}

// Les données sont stockées dans data
```

Les données récupérées peuvent ensuite être facilement converties en chaîne de caractères et utilisées par exemple comme source pour construire un dictionnaire à partir de données JSON. Dans l'exemple suivant, on reconstruit une chaîne de caractères à partir des données et en sachant que l'encodage utilisé est l'UTF8.

Conversion des données reçues en chaîne de caractères puis en objet grâce à JSON

```
NSString *jsonString = [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];

dictionary = [jsonString JSONObject];

[jsonString release];
```

Modifier le comportement d'une requête synchrone

La méthode `sendSynchronousRequest:returningResponse:error:` ne propose que les options de paramétrage les plus utiles.

Authentification

Il est possible d'indiquer un nom d'utilisateur et un mot de passe en l'ajoutant directement dans l'URL de la requête : `http://username:password@serveur`.

Gestion des redirections

Les redirections renvoyées par le serveur (301, 302) sont toujours suivies, et le client renverra dans le bloc `data` les données de la dernière réponse.

Définir le délai d'attente d'une requête

Il est possible de spécifier votre propre délai d'attente (*timeout*) lors de la création de l'objet `NSURLRequest`. Si le serveur n'a pas répondu dans l'intervalle indiqué, la connexion sera interrompue et une erreur renvoyée.

```
NSURLRequest *request = [NSURLRequest
requestWithURL:@"http://www.myserver.com/xxx"
cachePolicy:NSURLRequestUseProtocolCachePolicy
timeoutInterval:10];
```

Réaliser des traitements en arrière-plan

La plus grosse limitation de la méthode que nous venons de voir est que l'exécution de l'application est bloquée tant que le serveur n'a pas répondu à la requête (c'est le principe d'une requête synchrone).

Comprendre le thread principal

Dans une application iPhone, comme dans la plupart des applications de type client lourd, il existe un thread principal qui est chargé de recevoir les événements et de les traiter en permanence. C'est dans ce thread que s'exécute la boucle de traitement des événements et donc tous les appels aux méthodes de vos contrôleurs de vue, les méthodes de vos vues, et la plupart des fonctions de callback de votre application. La documentation en anglais parle de *main thread* (le thread principal).

Lorsqu'on exécute un appel réseau synchrone dans le thread principal, l'application ne peut plus traiter les événements. L'utilisateur a l'impression (et c'est le cas en effet) que l'interface est figée, totalement non réactive : il ne peut plus cliquer sur les boutons (ces derniers ne réagissent plus et ne changent pas d'état), il ne peut pas faire défiler les listes, changer d'onglets, etc.

ATTENTION Ne jamais exécuter de traitements longs sur le thread principal

Pour que l'interface reste en permanence réactive et que l'utilisateur continue à avoir un sentiment de fluidité, il est absolument essentiel de ne *jamais lancer de traitements longs sur le thread principal*. Cette remarque est particulièrement vraie pour tous les traitements réseau, mais elle s'applique aussi à des chargements de gros fichiers depuis le disque, à des calculs compliqués, etc.

Il existe heureusement plusieurs méthodes très faciles à utiliser, qui permettent de lancer un traitement en arrière-plan puis de reprendre la main dans le thread principal.

Lancer un traitement en arrière-plan

Un traitement en arrière-plan est un traitement qui s'exécute sur un autre thread que le thread principal. Pour lancer un traitement en arrière-plan, il suffit d'utiliser la méthode `-performSelectorInBackground:withObject:` de `NSObject`.

Cette méthode prend en paramètre un sélecteur (un pointeur sur fonction) et un objet à lui passer en paramètre. Elle crée un nouveau thread et lance l'exécution de la méthode désignée par le sélecteur. L'exécution du thread principal continue *en parallèle* de l'exécution de cette fonction, l'application peut donc rester réactive.

Un moyen simple de lancer une requête réseau est donc d'isoler le code qui exécute la requête dans une méthode et de demander l'exécution de cette méthode en arrière-plan.

Dans un contrôleur de vue, on pourrait donc trouver le code suivant qui déclenche un rafraîchissement de la vue.

Lancement d'un traitement en arrière-plan

```
- (IBAction) refreshButtonAction
{
    NSURL *url = [NSURL URLWithString:@"..."];
    [self performSelectorInBackground:@selector(refreshDataWithURL:)
        withObject:url];
}
```

Particularités des traitements en arrière-plan

Le traitement en arrière-plan va être exécuté sur un nouveau thread, ce qui entraîne quelques spécificités.

Mise en place d'un nouveau pool d'autorelease

Tout d'abord, le pool d'autorelease du thread principal n'est plus accessible. Pour pouvoir utiliser des objets autoreleased (ce que vous voudrez presque certainement faire), il faut créer un nouveau pool et le détruire lorsque les traitements sont finis.

Mise en place d'un pool d'autorelease pour une méthode qui est exécutée en arrière-plan

```
- (void) refreshDataWithURL:(NSString*) url
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    // ...

    [pool release];
}
```

Interactions avec l'interface

Le code qui s'exécute en arrière-plan va s'exécuter en même temps que le thread principal. Ainsi, les méthodes du thread principal peuvent créer, détruire ou déplacer des vues et surtout les dessiner pendant que votre code s'exécute.

Si vous avez besoin de modifier un des objets de l'interface : une vue, les données d'une table ou de manière générale toute variable qui pourrait être utilisée par le thread principal, il faut vous assurer que cela ne viendra pas interférer avec les opérations toujours en cours. Le meilleur moyen pour cela est de repasser le contrôle au thread principal à l'aide de la méthode :

```
performSelectorOnMainThread:withObject:waitUntilDone:
```


Cette méthode prend en paramètre un sélecteur qui sera appelé depuis le thread principal et qui peut recevoir un paramètre. Enfin, il est possible d'indiquer si l'on souhaite attendre que le code ait été exécuté ou pas avant de continuer l'exécution.

Le thread d'arrière-plan va alors attendre que le thread principal soit libre pour lui demander d'exécuter votre méthode. Les problèmes de synchronisation sont évités. C'est *le seul et unique moyen* que vous devez utiliser pour interagir avec l'interface.

ATTENTION Ne jamais modifier l'interface depuis un thread d'arrière-plan

Vous ne devez jamais modifier l'interface (la hiérarchie des vues, les contrôleurs de vues, les données utilisées par une table, etc.) depuis un thread d'arrière-plan.

Si vous le faites, vous obtiendrez très certainement des bogues graphiques dans l'interface, très difficiles à reproduire et à diagnostiquer.

L'exemple ci-après présente une méthode destinée à être exécutée en arrière-plan. Elle crée un nouveau pool d'autorelease, exécute une requête synchrone, et rappelle une autre méthode sur le thread principal qui est chargée de mettre à jour l'interface avec les données obtenues.

Exécution d'une requête réseau en arrière-plan et envoi des informations reçues vers le thread principal.

```
- (void) refreshDataWithUrl:(NSString*) url
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSURLRequest *request = ...;
    NSData *data;
    // [ ... ]
    // Exécution de la requête réseau - Voir exemple plus haut

    [self performSelectorOnMainThread:@selector(updateWithNewData:)
        withObject:data waitUntilDone:NO];

    [pool release];
}
```

BEST PRACTICE Allumer l'indicateur d'activité réseau

Lors de vos traitements réseau, vous pouvez faire appel à la propriété `networkIndicatorVisible` de la classe `UIApplication` pour faire tourner l'indicateur d'activité réseau pendant toute la durée de chargement.

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;  
[UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
```

C'est généralement une bonne idée de mettre en marche l'indicateur pour que l'utilisateur comprenne que l'application est en train de travailler en arrière-plan.

Attention cependant, comme pour tout autre accès à l'interface, il faut exécuter ces modifications depuis le thread principal et non pas dans un thread d'arrière-plan.

Connexions réseau asynchrones

Les requêtes asynchrones donnent au développeur un contrôle beaucoup plus fin sur la préparation, l'exécution de la requête et la réception des données.

Vous voudrez par exemple les utiliser quand :

- Le volume de données à charger est important et que vous avez besoin d'un indicateur de progression du téléchargement.
- Les règles par défaut de gestion des cookies et du cache ne conviennent pas à votre besoin.
- Vous voulez lancer et suivre de nombreuses connexions en parallèle.
- Vous voulez pouvoir interrompre une connexion avant qu'elle se termine.

Préparation d'une requête asynchrone

Pour exécuter une requête asynchrone, il faut, comme pour les requêtes synchrones, préparer un objet `NSURLRequest`. La classe qui exécutera ensuite le chargement est `NSURLConnection`. Celle-ci est initialisée avec l'objet `NSURLRequest` et un délégué.

À chaque fois que l'état de la connexion évolue, l'instance de `NSURLConnection` le notifie au délégué. Le plus souvent, le délégué sera la classe qui déclenche la connexion.

Préparation et lancement d'une connexion asynchrone

```
data = [[NSMutableData alloc] init];  
NSURLConnection *urlConnection = [[NSURLConnection alloc]  
    initWithRequest:urlRequest delegate:self];  
if (urlConnection) {  
    // La connexion va démarrer  
}
```

```
else {  
    // La connexion n'a pas pu être initialisée.  
    [data release];  
}
```

Une fois la méthode d'initialisation appelée, la connexion est lancée et le délégué va commencer à recevoir des informations. Dans l'exemple précédent, on initialise une variable d'instance (`data`) dans laquelle on stockera les données au fur et à mesure.

Il est important de noter qu'ici comme dans la plupart des cas où des fonctions de callback sont appelées, les méthodes du délégué sont appelées sur le thread principal. Il est donc possible de modifier directement l'interface.

REMARQUE Le délégué de `NSURLConnection` implémente un protocole informel

Le délégué d'un objet `NSURLConnection` n'a pas besoin d'implémenter un protocole explicitement. Il s'agit d'un protocole informel, c'est-à-dire qu'il suffit de créer les méthodes avec les noms attendus pour qu'elles soient appelées.

Cette pratique est de moins en moins répandue dans les API Cocoa. On lui préfère en général l'utilisation d'un protocole explicite.

Établissement de la connexion

Une fois la connexion établie, la méthode `connection:didReceiveResponse:` du délégué est appelée avec la connexion en cours et un objet `NSURLResponse`.

Si jamais le serveur a demandé une redirection, cette méthode sera appelée plusieurs fois. On réinitialise à chaque fois l'objet qui contient les données reçues pour qu'il ne garde que les données de la dernière réponse.

Réception d'une réponse à une connexion asynchrone

```
- (void)connection:(NSURLConnection *)connection  
didReceiveResponse:(NSURLResponse *)response  
{  
    [data setLength:0];  
}
```

Il est également possible à ce moment de connaître la taille totale de la réponse à l'aide de la propriété `expectedContentLength` de l'objet `response`.

Réception de données

À chaque fois que des données sont reçues, la méthode du délégué `connection:didReceiveData:` est appelée. Elle doit enregistrer les données reçues.

Réception de données dans une connexion asynchrone

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)someData
{
    [data appendData:someData];
}
```

La taille totale des données attendues étant connue, il est possible de mettre à jour dans l'interface un indicateur de progression du téléchargement.

Fin de connexion

La fin de la connexion est signalée au délégué par l'intermédiaire de la méthode `connection:didFailWithError:` en cas d'erreur ou, en cas de succès, de la méthode `connection:didFinishLoading:`.

Méthode appelée en cas d'échec d'une connexion réseau asynchrone

```
- (void)connection:(NSURLConnection *)connection
didFailWithError:(NSError *)error
{
    [connection release];
    [data release];

    NSLog(@"Echec de la connexion - %@ %@",
          [error localizedDescription],
          [[error userInfo] objectForKey:NSErrorFailingURLStringKey]);
}
```

Méthode appelée quand une connexion réseau asynchrone se termine normalement

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Succès de la requête. %d octets reçus",[data length]);

    [connection release];

    // Traitement sur les données reçues

    [data release];
}
```

Conclusion

Nous avons vu dans ce chapitre comment exécuter des requêtes réseau en mode synchrone ou asynchrone, et comment exécuter des traitements en arrière-plan.

Avec ces éléments, vous avez toutes les connaissances nécessaires pour implémenter des applications complexes communiquant avec des services web.

CONSEIL Mise en place de caches de données

Les classes `NSURLConnection` et `NSURLRequest` tentent de mettre en cache les données reçues. Néanmoins, sur l'iPhone ce cache est extrêmement limité : les données sont uniquement gardées en mémoire (pas écrites sur le disque) et la taille du cache est très limitée.

Il est fortement recommandé d'implémenter votre propre cache pour tous les éléments un peu lourds qui proviennent du Web. C'est en particulier vrai pour toutes les images distantes que vous voudrez intégrer dans vos applications.

13

Persistance d'objets avec CoreData

Core Data est un framework apparu avec Mac OS 10.4 sur le Mac et intégré à Cocoa Touch pour l'iPhone depuis l'OS 3.0. C'est un framework de mapping objet-relationnel (*Object-Relational Mapping* anglais ou ORM) qui permet de stocker dans une base de données un graphe d'objets.

Avant la démocratisation de ces outils, il était courant de passer un temps important dans chaque application pour écrire des objets en charge de la persistance des données. Le plus souvent en appliquant le design pattern DAO (*Data Access Objects*) on écrivait du code qui exécutait les requêtes SQL pour charger et enregistrer des objets.

Les frameworks ORM permettent aux développeurs de manipuler directement des objets métier en mémoire, de les lier entre eux et de sauver l'état complet du graphe (c'est-à-dire l'état des objets, mais aussi l'état des relations entre les objets) sans écrire une seule ligne de code SQL.

APPROFONDIR Les frameworks ORM

Difficile de s'en passer une fois qu'on y a goûté : il existe de nombreux frameworks de mapping objet-relationnel pour les langages modernes. Le plus connu est probablement Hibernate qui a conquis en quelques années le monde des développeurs Java et a fortement inspiré la norme EJB3.

En PHP, Propel et Doctrine, intégrés notamment au framework Symfony, remplissent le même rôle et Hibernate propose une implémentation .NET.

Dans ce chapitre nous présenterons le principe de fonctionnement d'un framework ORM, puis nous présenterons les bases de Core Data.

Introduction à l'ORM

Pour la très grande majorité des projets qui manipulent des objets stockés dans une base de données, les frameworks de mapping objet-relationnel sont une solution extrêmement élégante et efficace à un certain nombre de problèmes plus ou moins complexes.

Du monde objet au monde relationnel

Les frameworks ORM prennent en charge la conversion d'un modèle d'objets (les objets de l'application en mémoire) vers un modèle relationnel (une base de données composée de tables et de clés étrangères).

Bien qu'il soit relativement simple de stocker dans une base des objets simples, le problème devient beaucoup plus compliqué quand on souhaite enregistrer des relations n-n ou des listes contenant des objets de types différents. Pour ces deux cas, il faut créer des tables supplémentaires afin d'exprimer en langage relationnel la richesse des relations permises par le monde objet.

Cette complexité est complètement masquée par le framework ORM. Le développeur n'a plus besoin de se soucier des tables supplémentaires qui vont être créées et maintenues automatiquement par le framework.

Gestion des relations

Dans le monde objet, les relations sont concrétisées par des références que les objets gardent les uns sur les autres. Un objet `Voiture` va garder une référence vers un objet `Conducteur` ; un objet `Éditeur` va garder une liste de références vers des `Livres`.

Un des rôles joués par le framework ORM est le chargement automatique des objets liés à un objet. Ainsi, une fois l'objet `Voiture` chargé en mémoire, il suffira d'appeler la propriété `voiture.conducteur` pour obtenir une référence vers le conducteur. Il suffit d'utiliser cette même propriété pour affecter un nouveau conducteur et le framework va prendre en compte ce changement et le répercutera dans la base quand on lui demandera d'enregistrer les modifications faites au graphe des objets.

Dans le cas de relation bidirectionnelle, le framework prend en charge la mise à jour de l'autre extrémité de la relation. Ainsi, quand on affecte un nouveau conducteur à une voiture, la propriété `voiture.Courante` de l'instance de conducteur est mise à jour automatiquement.

Performances

Le framework ORM est un outil très complexe, souvent plus encore que le code qui aurait été écrit manuellement pour faire persister des objets en base.

Il est ainsi capable de ne pas charger automatiquement toutes les relations : il ne charge les objets liés que quand on y fait référence. C'est totalement transparent pour le développeur et cela permettra souvent un gain très important en performances.

Le framework peut également ne pas charger toutes les propriétés, ce qui est intéressant quand certaines propriétés prennent beaucoup de place en mémoire mais ne sont pas toujours utilisées.

Il permet d'exécuter une requête, de récupérer une liste d'objets, mais de faire en sorte que les objets ne soient réellement chargés depuis la base que quand ils sont utilisés. Cela permet encore une économie très importante d'accès à la base et une forte amélioration des performances.

Notion de contexte ou de session

Dans tous les frameworks ORM, on retrouve la notion de contexte. Les objets étant capables d'aller chercher des informations en base pour compléter leurs relations ou leurs propriétés, ils ont besoin de garder une référence vers cette base de données.

De plus, puisqu'on ne manipule pas un seul objet mais bien un graphe d'objets, il faut un mécanisme permettant de délimiter le graphe d'objets : c'est la session dans la terminologie Hibernate ou le contexte dans la terminologie Core Data.

Tous les objets chargés depuis la base de données sont associés à ce contexte. Lorsqu'une modification est faite à l'un d'eux, il notifie le contexte qui sait alors que cet objet a été modifié et enregistre quelle propriété devra être sauvegardée. Quand un objet a besoin d'aller chercher des objets liés, il interroge le contexte qui renvoie directement les instances existantes en mémoire, si ces objets étaient déjà chargés, ou qui fait appel à la base pour les charger si ce n'était pas le cas.

Mise en place de l'environnement Core Data

Pour fonctionner, Core Data a besoin de trois éléments distincts :

- 1 un modèle qui contient la description des objets qui vont être manipulés et les relations qui peuvent exister entre eux : `NSManagedObjectModel` ;
- 2 un entrepôt de stockage de données (*Store* en anglais), implémenté par la classe `NSPersistentStoreCoordinator` ;

3 un contexte d'objet que l'on crée à partir des deux éléments précédents : `NSManagedObjectContext`.

Si vous avez sélectionné un modèle de projet avec Core Data, ces trois étapes sont déjà faites dans le code généré.

RAPPEL Ne pas oublier d'ajouter Core Data à votre projet

Core Data est un framework supplémentaire que vous devez ajouter à votre projet. Si vous n'avez pas sélectionné un modèle de projet avec Core Data, il aura été ajouté automatiquement.

Sinon, il suffit de cliquer avec le bouton droit sur la cible de votre projet (dans le groupe *Target* de Xcode) puis *Add > Existing Framework*. Dans la fenêtre qui s'ouvre, cliquez sur le plus en bas à gauche et sélectionnez *Core Data*.

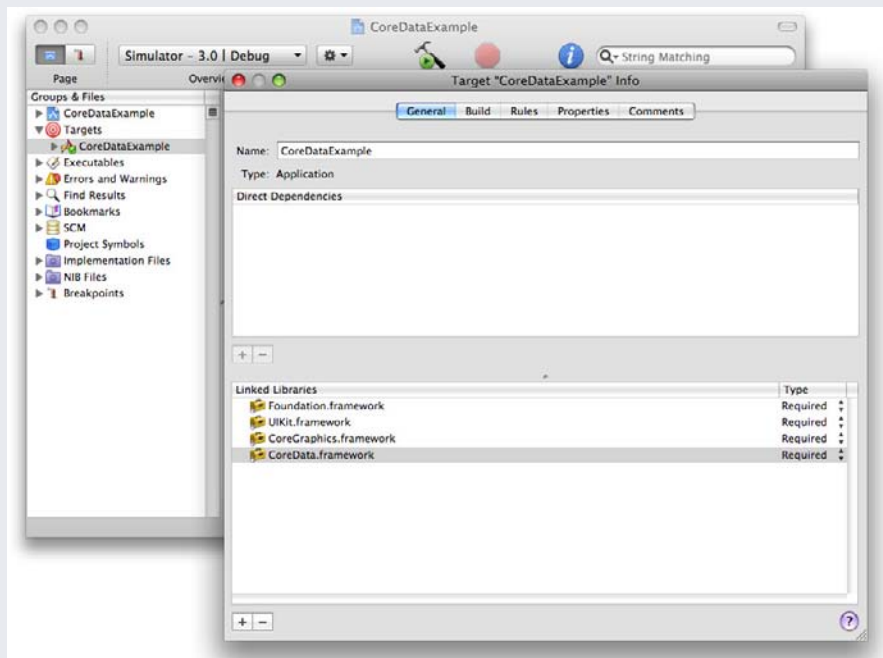


Figure 13-1 Ajout du framework Core Data au projet

Chargement de la description du modèle

La description du modèle contient la liste de toutes les entités (tous les objets persistants) de l'application, de leurs attributs, des relations entre eux et des paramètres et contraintes que vous leur aurez ajoutés.

Nous verrons un peu plus loin comment se fait la définition du modèle dans Xcode. Sachez simplement que le modèle est enregistré sous forme d'un ensemble de fichiers qui sont interprétés à l'exécution par Core Data.

La classe `NSManagedObjectModel` propose la méthode statique `mergedModelFromBundles` qui renvoie une nouvelle instance initialisée avec toutes les définitions de modèle trouvées dans le bundle.

Ainsi, pour charger en mémoire toutes les définitions de modèle du bundle principal, on peut utiliser le code suivant, qui ne charge le modèle qu'une seule fois (il est gardé dans une variable d'instance) :

```
- (NSManagedObjectModel *)managedObjectModel {
    if (managedObjectModel != nil) {
        return managedObjectModel;
    }
    managedObjectModel = [[NSManagedObjectModel
mergedModelFromBundles:nil] retain];
    return managedObjectModel;
}
```

Mise en place de l'entrepôt de stockage des données

La classe `NSPersistentStoreCoordinator` est responsable de tous les accès vers l'entrepôt de stockage. Pour la plate-forme iPhone, l'entrepôt est toujours une base de données SQLite ; sur la plate-forme Mac OS X il est possible d'enregistrer également les données au format XML.

On initialise une instance en lui passant en paramètre l'objet `NSManagedObjectModel` créé un peu plus tôt, puis on lui indique à quel endroit stocker les données.

De la même façon que pour le modèle, on ne crée qu'une seule instance de cet objet dans l'application qui est gardée dans une variable d'instance.

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (persistentStoreCoordinator != nil) {
        return persistentStoreCoordinator;
    }

    NSURL *storeUrl = [NSURL fileURLWithPath: [[self
applicationDocumentsDirectory] stringByAppendingPathComponent:
@"CoreDataExample.sqlite"]];
```

```

NSError *error;
persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel: [self managedObjectModel]];
if (![persistentStoreCoordinator
addPersistentStoreWithType:NSSQLiteStoreType configuration:nil
URL:storeUrl options:nil error:&error]) {
    // Une erreur est survenue.
}

return persistentStoreCoordinator;
}

// Récupération du chemin vers le répertoire Documents de l'application
// Cf. Chapitre 11
- (NSString *)applicationDocumentsDirectory {
    NSArray *paths =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *basePath = ([paths count] > 0) ? [paths objectAtIndex:0] :
nil;
    return basePath;
}

```

Création du contexte

Le contexte permet de maintenir le lien entre les objets et la base de données. On parle d'objets managés, car ils appartiennent tous au contexte et qu'ils sont tous gérés par le contexte.

C'est également grâce à ce contexte qu'on pourra ensuite exécuter des requêtes sur la base de données.

On initialise le contexte à l'aide de `alloc/init` puis on utilise la méthode `setPersistentStoreCoordinator:` pour lui permettre d'accéder au modèle et aux données. Ici encore, on ne crée qu'une seule instance de ce contexte qui est conservée dans une variable d'instance.

```

- (NSManagedObjectContext *) managedObjectContext {

    if (managedObjectContext != nil) {
        return managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator =
        [self persistentStoreCoordinator];

```

```
if (coordinator != nil) {
    managedObjectContext = [[NSManagedObjectContext alloc] initWith:
        [managedObjectContext setPersistentStoreCoordinator:
coordinator];
}
return managedObjectContext;
}
```

Description du modèle

La description du modèle peut se faire en code, mais il est très fortement recommandé d'utiliser l'outil graphique intégré à Xcode qui reprend un vocabulaire proche des diagrammes de classe UML.

ASTUCE Séparer le modèle en plusieurs fichiers de description

Votre modèle peut être composé d'un ou plusieurs fichiers de description. Si vous utilisez la méthode présentée plus haut pour initialiser `NSManagedObjectContext`, tous les modèles seront fusionnés lors du chargement de l'application.

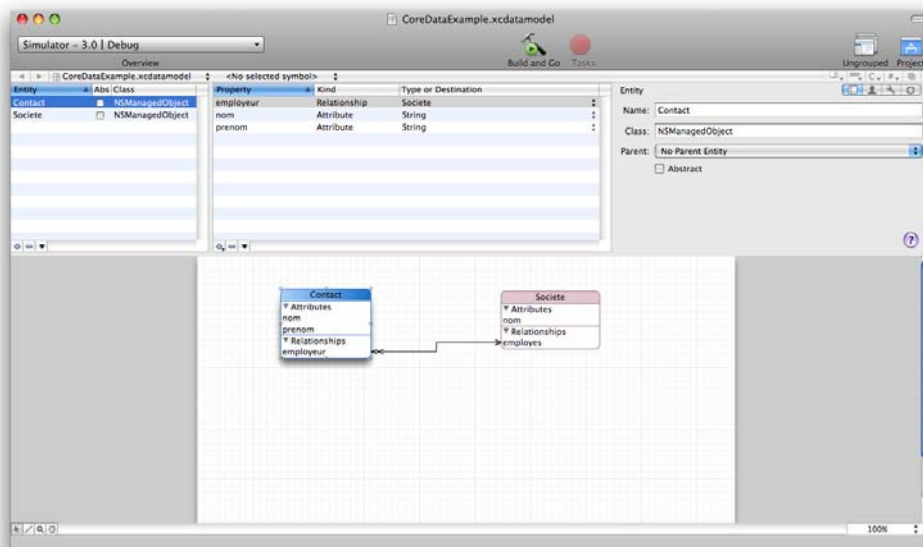


Figure 13-2 L'éditeur de modèle dans Xcode

Création d'un nouveau modèle

Si vous avez créé votre projet en utilisant un modèle basé sur Core Data, un fichier de définition de modèle vide a été créé, qui porte le nom de votre projet.

Vous pouvez toujours en ajouter un à l'aide du menu *New File... > Ressource > Core Data Model*.

L'interface est découpée en trois zones supérieures et en une zone inférieure.

Dans la partie supérieure de l'écran, on trouve trois panneaux permettant de gauche à droite :

- 1 de lister toutes les entités du modèle, d'en ajouter et d'en supprimer ;
- 2 de lister tous les attributs et relations de l'entité actuellement sélectionnée ;
- 3 d'éditer les propriétés supplémentaires de l'entité, l'attribut ou la relation sélectionnée.

Dans la partie inférieure, on trouve une représentation graphique du modèle : entités, attributs et relations.

Édition du modèle

L'édition du modèle se fait de manière très intuitive dans l'interface graphique proposée par Xcode.

Vous pouvez simplement créer de nouvelles entités, définir le nom, le type et la valeur par défaut de leurs attributs, ajouter des contraintes particulières sur les attributs (comme une plage de valeurs autorisées, ou une expression régulière qui doit être vraie).

La définition des relations se fait tout aussi simplement à l'aide de l'outil *Line* disponible en bas à gauche qui permet de relier deux objets à l'aide d'un trait. On définit ensuite, dans la partie supérieure de l'éditeur, les propriétés de cette relation : uni- ou bidirectionnelle, 1 à N, N à N, etc.

APPROFONDIR Utilisation de Xcode pour concevoir le modèle

Le guide *Creating a Managed Object Model with Xcode* (disponible dans la documentation du SDK iPhone) présente en détail l'interface de conception de modèle de Xcode.

Si vous souhaitez plus d'informations sur la signification des différentes options, il vaut mieux vous référer au guide de référence Core Data : *Core Data Programming Guide*.

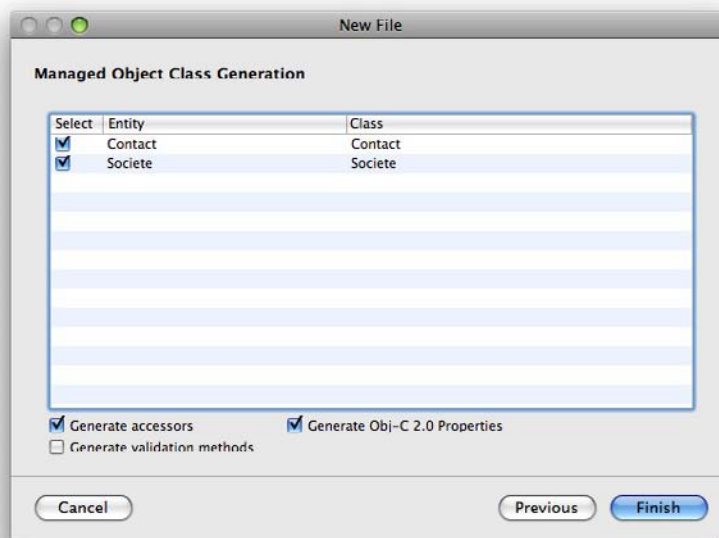
Création des classes du modèle

Il n'est pas indispensable de créer vos propres classes pour le modèle. Par défaut, la classe `NSManagedObject` est instanciée pour représenter vos entités et elle est capable de gérer les messages correspondant aux attributs que vous avez déclarés dans le modèle ainsi que la validation de ces derniers.

Néanmoins, si vous souhaitez pouvoir utiliser la complétion dans Xcode et avoir la possibilité d'ajouter des méthodes supplémentaires à vos objets métiers, il est utile de créer explicitement une classe pour les représenter.

Xcode permet de faire cela automatiquement. Il faut tout d'abord sélectionner le modèle dans les ressources du projet (panneau de gauche de la fenêtre principale Xcode) puis à l'aide du menu *New File... > Cocoa Touch Class > Managed Object Class*. Un assistant se lance, qui va rechercher dans le modèle toutes les entités et les propose dans une liste. Il suffit ensuite de sélectionner toutes les entités pour lesquelles vous voulez que la classe soit générée automatiquement.

Figure 13-3
Sélection des entités
pour lesquelles générer
une classe



Le code généré est très simple. On remarque qu'il n'y a aucun code spécifique à CoreData dans la classe, si ce n'est la référence à la classe parente : `NSManagedObjectModel`.

Fichier d'interface généré par Xcode pour l'entité Contact

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

@interface Contact : NSObject
{
}

@property (nonatomic, retain) NSString * nom;
@property (nonatomic, retain) NSString * prenom;
@property (nonatomic, retain) Societe * employeur;

@end
```

Fichier d'implémentation généré par Xcode pour l'entité Contact

```
#import "Contact.h"

@implementation Contact

@dynamic nom;
@dynamic prenom;
@dynamic employeur;

@end
```

Manipulation d'objets gérés par le contexte

Les objets maintenus persistants par Xcode (les entités) sont des instances d'objet Objective-C qui dérivent toutes de `NSObject`.

On utilise les objets de manière tout à fait habituelle, mais leur création et leur enregistrement sont naturellement un peu particuliers.

Création d'une nouvelle instance

La création d'une nouvelle instance se fait à l'aide de la méthode statique `insertNewObjectForEntityForName:inManagedObjectContext:` de la classe `NSEntityDescription`.

Cette classe prend en premier paramètre le nom de l'entité, et en deuxième paramètre le contexte. Elle renvoie une instance initialisée avec les valeurs par défaut définies dans le modèle.

```
Contact *newContact = [NSEntityDescription
    insertNewObjectForEntityForName:@"Contact"
    inManagedObjectContext:context];
```

Il est important de noter que cette instance est ajoutée au pool de libération automatique et que l'appelant doit la retenir s'il souhaite conserver une référence vers cet objet.

Enregistrement des objets du contexte

La sauvegarde de tous les objets du contexte se fait en appelant la méthode `save` sur le contexte. Tous les objets qui ont été ajoutés au contexte, modifiés ou supprimés sont alors sauvegardés en base.

```
NSError *error;
if (![context save:&error])
{
    NSLog(@"Erreur lors de l'enregistrement des données: %@ - %@", error,
        [error userInfo]);
}
```

Si une erreur survient lors de l'enregistrement (par exemple parce que les attributs ne respectent pas les contraintes de validation), la variable `error` contiendra une description complète de l'erreur.

Exécution d'une requête pour obtenir des objets

Pour exécuter une requête, il faut créer une instance de l'objet `NSFetchRequest` dans laquelle on définit les critères de recherche.

Il existe trois critères principaux pour une requête :

- 1 l'entité que l'on cherche à obtenir ;
- 2 un prédicat sur cette entité ;
- 3 un (ou plusieurs) critère de tri.

Recherche d'une entité dans la base

La recherche la plus simple permet de récupérer tous les objets d'une entité en particulier. Il suffit pour cela d'indiquer uniquement l'entité que l'on recherche.

```
NSFetchRequest *fetchRequest = [[[NSFetchRequest alloc] init] autorelease];
[fetchRequest setEntity:[NSEntityDescription entityForName:@"Contact" inManagedObjectContext:managedObjectContext]];

NSArray *contacts = [managedObjectContext executeFetchRequest:fetchRequest error:&error];
```

On remarque l'utilisation de la méthode `entityForName:inManagedObjectContext:` de la méthode `NSEntityDescription` qui renvoie la description d'une entité à partir de son nom.

Recherche basée sur un prédicat

Un prédicat est une condition qui doit être vérifiée pour que l'objet soit sélectionné et renvoyé. En Objective-C, c'est une instance de l'objet `NSPredicate`.

```
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"prenom = 'John'"];
[fetchRequest setPredicate:predicate];

NSArray *johns = [managedObjectContext executeFetchRequest:fetchRequest error:&error];
```

APPROFONDIR Maîtriser les prédicats

La syntaxe exhaustive des prédicats est décrite dans le guide *Predicate Programming Guide* qui est fourni avec le SDK.

Définir l'ordre des objets renvoyés

Des critères de tri peuvent être ajoutés à la requête pour définir dans quel ordre les objets doivent être renvoyés. C'est la propriété `sortDescriptors` de l'objet `NSFetchRequest`. Elle contient une liste d'objet `NSSortDescriptor`. Chacun définit un tri sur une propriété.

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"nom" ascending:YES];

NSArray *sortDescriptors = [NSArray arrayWithObject:sortDescriptor];

[fetchRequest setSortDescriptors:sortDescriptors];
[sortDescriptor release];

NSArray *sortedContacts = [managedObjectContext
executeFetchRequest:fetchRequest error:&error];
```

Aller plus loin avec les requêtes

L'objet `NSFetchRequest` permet de définir de nombreux paramètres qui ont un impact essentiel sur les performances de votre application. Il est ainsi possible de demander à ce que tous les objets ne soient pas chargés en mémoire dès la requête, mais plutôt quand on y accédera ; ou encore d'indiquer que les entités liées ne doivent pas être chargées immédiatement.

La documentation de `NSFetchRequest` décrit plus en détail les possibilités de paramétrage des requêtes.

Supprimer un objet

Il est possible de supprimer un objet à l'aide de la méthode `deleteObject:` du contexte.

```
[managedContext deleteObject:contact];
```

L'objet n'est réellement supprimé qu'après un appel à la méthode `save`.

Conclusion

Dans ce chapitre, nous avons couvert les bases de Core Data. C'est un framework extrêmement riche et puissant qui devrait faire l'objet de toute votre attention si vos applications manipulent des volumes de données quelque peu importants.

14

Manipuler des données multimédias

L'iPhone est plébiscitée notamment comme une formidable plate-forme multimédia. Ses utilisateurs ont l'habitude d'écouter de la musique, de prendre des photos et regarder des vidéos.

Nous montrerons dans ce chapitre comment lire des sons au sein de votre application, comment lancer la lecture de vidéos, comment accéder à la bibliothèque iPod de l'utilisateur et enfin comment utiliser l'appareil et la caméra vidéo de l'iPhone.

Intégrer le son au cœur de vos applications

Le son peut apporter une dimension supplémentaire à votre application. Utilisé avec parcimonie et sous réserve que vous preniez un grand soin à l'enregistrement et à la préparation de vos fichiers audio, il permet de surprendre l'utilisateur via un canal homme-machine encore sous-utilisé.

Les formats audio pris en charge par l'iPhone

L'iPhone est capable de lire et décoder différents formats audio non compressés et compressés. Le tableau ci-après résume les principaux formats reconnus par l'iPhone.

Format	Compression	Description
MP3	Oui	Le format MPEG1-Audio layer 3 est un format de compression audio avec perte. C'était à l'origine le format audio du format vidéo MPEG1. Rendu célèbre dans les années 1990, ce n'est plus aujourd'hui le format le plus efficace pour compresser des fichiers audio.
AAC	Oui	<i>Advanced Audio Coding</i> est un algorithme de compression avec perte inventé pour offrir une meilleure qualité que le MP3 à débit égal.
ALAC	Oui	<i>Apple Lossless Audio Codec</i> est un format de compression audio sans perte propriétaire Apple. Il permet de compresser des fichiers audio sans perdre d'information (donc sans aucune dégradation du son).
IMA4	Oui	Aussi connu sous le nom ADPCM, ce format permet une compression avec perte des fichiers audio. Son principal intérêt est qu'il peut être facilement décodé par le CPU de l'iPhone, ce qui en fait le format de choix si vous devez lire plusieurs fichiers audio en même temps.
Linear PCM	Non	C'est le format de fichiers audio non compressé le plus répandu. On le retrouve dans les fichiers WAV ou AIFF.

ATTENTION Les limites du hardware audio de l'iPhone

À l'exception du format IMA4, tous les formats compressés sont décodés par le hardware de l'iPhone. Un seul fichier peut être lu à la fois ; au-delà, c'est le CPU qui doit faire le décodage, avec de fortes répercussions sur votre application en termes de performances.

Si vous souhaitez lire plusieurs fichiers audio en même temps ou lire un fichier audio pendant que l'utilisateur écoute l'iPod, il est fortement conseillé d'utiliser le format IMA4 ou le format PCM.

Convertir les fichiers audio pour l'iPhone

Tous les ordinateurs Mac disposent de `afconvert`, un outil en ligne de commande qui permet de convertir des fichiers audio d'un format à l'autre.

Pour convertir un fichier audio au format Linear PCM dans un fichier CAF :

```
afconvert -f caff -d LEI16 <fichier source> <fichier destination>
```

Pour convertir au format AAC :

```
afconvert -f aac <fichier source> <fichier destination>
```

Enfin, pour convertir un fichier au format IMA4 :

```
afconvert -f caff -d ima4 <fichier source> <fichier destination>
```

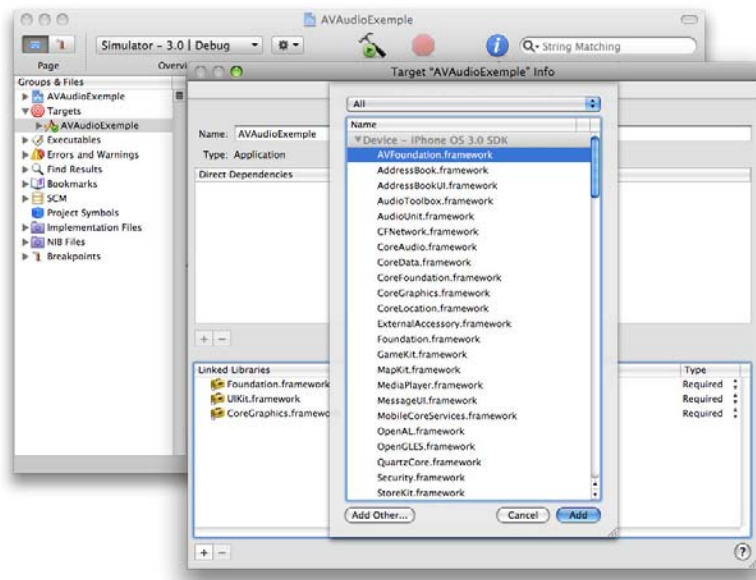
Lorsque `afconvert` est lancé sans aucun argument, il affiche la liste des formats pris en charge.

Lancer la lecture de sons dans votre application

Il existe plusieurs méthodes pour lire des sons dans votre application. La méthode présentée ici s'appuie sur la classe `AVAudioPlayer` qui offre une interface objet aux API de plus bas niveau comme Core Audio.

Cet objet fait partie du framework `AVFoundation` qui doit être ajouté à votre projet. Pour cela, sélectionnez la cible de votre projet (dans le groupe *Targets* de Xcode) puis sélectionnez *Add > Existing Framework*. Dans la fenêtre qui s'ouvre, cliquez sur le **+** en bas à gauche puis sélectionnez `AVFoundation`..

Figure 14-1
Ajout du framework
`AVFoundation` dans le projet



Vous devez ensuite importer son fichier d'en-tête :

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

On initialise `AVAudioPlayer` avec l'URL d'un fichier audio. Il est possible de régler le volume de lecture (valeur entre 0 et 1), ce volume permet d'ajuster le son du fichier qui va être lu, mais ne change pas le réglage du volume de l'iPhone (il n'est pas possible de changer le volume de l'iPhone depuis une application).

```
NSString *soundPath = [[NSBundle mainBundle] pathForResource:@"alerte"
ofType:@"wav"];
AVAudioPlayer *audioPlayer = [[AVAudioPlayer alloc]
initWithContentsOfURL:[NSURL URLWithString:soundPath]
error:nil];
[audioPlayer play];
[audioPlayer setVolume:1];
```

On peut fournir un délégué pour être informé lorsque la lecture se termine ou lorsqu'elle est interrompue (si un appel entrant est reçu par exemple).

La principale méthode du délégué est `audioPlayerDidFinishPlaying:successfully:` qui est appelée lorsque la lecture est terminée.

REMARQUE Volume de l'iPhone

Bien qu'il ne soit pas possible depuis une application de définir le son de l'iPhone, vous pouvez ajouter dans vos vues le contrôle `MPVolumeView` qui est une barre de réglage horizontale liée au volume système. Si l'utilisateur bouge le curseur de ce contrôle, le volume système est modifié. S'il utilise les boutons de réglage du volume de son iPhone, le curseur bouge pour refléter les modifications.

Lecture de vidéos

La lecture de vidéos sur l'iPhone se fait à l'aide du composant `MPMoviePlayerController` qui fait partie du framework `MediaPlayer`. Ce composant spécifique permet de lire des vidéos depuis les ressources de votre application ou depuis une URL externe.

Lorsque l'application démarre la lecture, le lecteur vient recouvrir l'ensemble de l'application et joue la vidéo. Il est possible de définir quels contrôles doivent être affichés (pause, avance rapide, etc.). Il n'est par contre pas possible de lire des vidéos dans une zone plus petite : la vidéo doit être jouée en mode plein écran.

Formats de vidéos pris en charge

Le composant `MPMoviePlayerController` est capable de lire toutes les vidéos prises en charge par l'iPhone. Le format le plus utilisé est le H264, on peut le trouver dans des fichiers `.mov`, `.mp4` et `.mpv`.

ASTUCE Déterminer le format d'un fichier vidéo

Il est possible de trouver très facilement le format d'une vidéo en l'ouvrant dans le lecteur QuickTime et en affichant l'inspecteur de séquence (menu *Fenêtre > Afficher l'inspecteur de séquence*).

Intégrer le lecteur vidéo dans une application

La première étape consiste à ajouter le framework `MediaPlayer` dans votre projet et à importer le fichier d'en-tête spécifique :

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

Il faut ensuite créer une instance de l'objet `MediaPlayer` et l'initialiser avec l'URL de la ressource vidéo à lire.

```
NSURL *movieURL = [NSURL URLWithString:@"http://www.mywebsite.com/  
mymovie.mp4"];  
moviePlayer = [[MPMoviePlayerController alloc]  
initWithContentURL:movieURL];
```

Une fois l'objet `MPMoviePlayerController` créé, il commence immédiatement à charger la vidéo. Il est possible d'appeler immédiatement la méthode `play` pour lancer la lecture (ce qui entraîne l'affichage du lecteur vidéo en plein écran).

```
[moviePlayer play];
```

S'abonner aux notifications pour suivre le déroulement de la lecture

Pour suivre le déroulement de la lecture, la classe `MPMoviePlayerController` ne propose pas de délégué, mais s'appuie sur le mécanisme des notifications qui est très répandu dans le monde Cocoa.

L'application doit s'enregistrer auprès du centre de notification pour recevoir cet événement. On indique :

- l'objet qui souhaite recevoir les événements (en général `self`) ;
- le nom du sélecteur à appeler ;

- le nom de la notification qu'on souhaite recevoir ;
- et enfin l'objet pour lequel on veut recevoir les notifications.

```
[[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(moviePlaybackDidFinish:)
 name:MPMoviePlayerPlaybackDidFinishNotification
 object:moviePlayer];
```

Le sélecteur peut (par exemple) libérer la mémoire du lecteur vidéo lorsque la lecture est terminée.

```
- (void) moviePlaybackDidFinish:(NSNotification*)notification
{
    [moviePlayer release];
    moviePlayer = nil;
}
```

Les autres notifications qui peuvent être transmises sont :

- `MPMoviePlayerContentPreloadDidFinishNotification` qui indique que le pré-chargement est terminé (il est donc possible d'appeler la méthode `play` et la vidéo démarrera sans attente) ;
- et `MPMoviePlayerScalingModeDidChangeNotification` qui est appelée lorsque l'utilisateur change le zoom de la vidéo.

Personnaliser le lecteur vidéo

Il est possible de personnaliser le lecteur vidéo en choisissant la couleur de fond (qui par défaut est le noir) et les contrôles à afficher à l'écran.

La couleur de fond peut être définie avec la propriété `backgroundColor`. Elle est affichée autour de la vidéo lorsque celle-ci ne remplit pas tout l'écran.

```
moviePlayer.backgroundColor = [UIColor whiteColor];
```

Les contrôles affichés à l'écran par défaut permettent à l'utilisateur de contrôler la lecture de la vidéo (lecture/pause, barre de temps) et le volume. Il est possible de limiter l'interface au réglage du volume ou de supprimer complètement l'interface. C'est ce qui est fait généralement quand une vidéo est jouée au démarrage d'une application.

La propriété `movieControlMode` peut donc prendre un des trois états suivants :

- `MPMovieControlModeDefault`,
- `MPMovieControlModeVolumeOnly` ou
- `MPMovieControlModeHidden`.

```
moviePlayer.movieControlMode = MPMovieControlModeHidden;
```

Aller plus loin avec les vidéos iPhone

Le lecteur vidéo est un composant extrêmement puissant qui a bénéficié de plusieurs améliorations lors du lancement de l'iPhone OS 3.0. Les deux plus importantes sont la possibilité de lire des flux vidéo en streaming, ce qui permet de transmettre des vidéos *live* à l'iPhone et la possibilité d'ajouter des éléments d'interface par-dessus la vidéo.

Proposer des vidéos live

La préparation des flux vidéo pour iPhone est un sujet complexe qui fait l'objet de plusieurs documentations Apple.

Le guide *HTTP Live Streaming Overview* explique comment fournir des flux vidéo en HTTP. Il est possible de protéger les contenus en les chiffrant et en utilisant une connexion HTTPS.

Ajouter des éléments par-dessus la vidéo

Avec iPhone OS 3.0, Apple a documenté une méthode pour enrichir les vidéos, qui était déjà connue des développeurs, mais restait non officielle (et donc soumise à des rejets lors de la soumission à l'App Store).

Il est désormais possible d'enrichir l'interface de lecture des vidéos en ajoutant des vues par-dessus, ce qui permet d'ajouter vos propres zones de texte, des boutons, etc., ou même de dessiner directement sur la vidéo.

Cette technique est présentée dans le projet exemple `MoviePlayer` qui fait partie du SDK.

Accéder à la bibliothèque musicale de l'iPhone

Autre nouveauté apportée par l'iPhone OS 3.0, l'accès à la bibliothèque de l'utilisateur, fonctionnalité très demandée par les développeurs.

Elle permet d'obtenir la liste de toutes les chansons de l'utilisateur et de contrôler le lecteur audio.

ATTENTION Seuls les contenus audio sont accessibles

Dans sa version actuelle, l'API d'accès à la bibliothèque iPod permet de lire tous les contenus audio (chansons, podcasts, livres audio), mais elle ne permet pas d'accéder aux contenus vidéo.

Comme pour les autres exemples de ce chapitre, vous devez commencer par ajouter à votre projet le framework MediaPlayer et importer son fichier d'en-tête.

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

ATTENTION La bibliothèque iPod n'est pas accessible dans le simulateur

Toutes les fonctionnalités de l'API permettant l'accès à la bibliothèque iPod doivent être testées sur un vrai terminal, car le simulateur ne permet pas de simuler l'application iPod.

Parcourir la bibliothèque musicale de l'iPhone

Le framework MediaPlayer propose deux approches pour accéder à la bibliothèque de l'utilisateur.

La première approche permet d'afficher une fenêtre dans laquelle l'utilisateur choisit les morceaux qui l'intéressent. Cette fenêtre reprend le *look&feel* de l'application iPod.

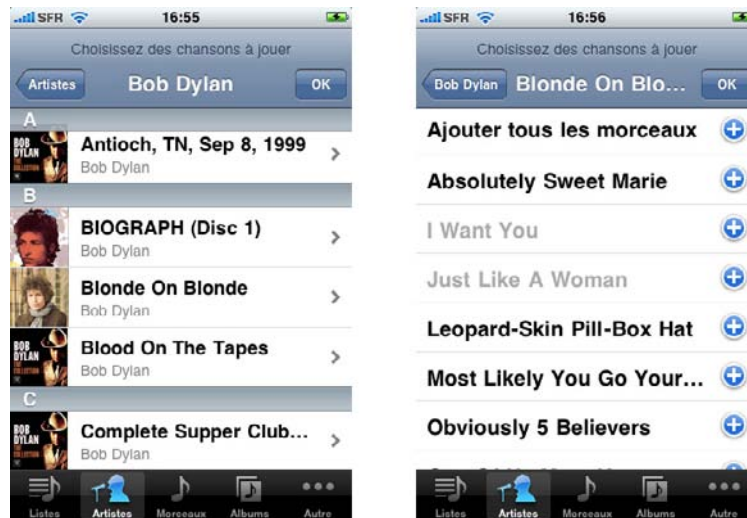
La deuxième approche permet de lancer depuis le code de votre application des recherches dans la base de données iPod.

Demander à l'utilisateur de choisir de la musique

La classe `MPMediaPickerController` est un contrôleur de vue modale. On lui fournit un délégué qui est appelé quand l'utilisateur a fini de sélectionner des morceaux.

```
MPMediaPickerController *mediaPickerController =  
    [[MPMediaPickerController alloc]  
     initWithMediaTypes: MPMediaTypeAnyAudio];  
  
[mediaPickerController setDelegate: self];  
[mediaPickerController setAllowsPickingMultipleItems: YES];  
mediaPickerController.prompt = @"Choisissez des chansons à jouer";  
  
[self presentViewController: mediaPickerController animated: YES];  
[mediaPickerController release];
```

Figure 14–2
L'interface du contrôleur de vue
MPMediaPickerController



Le délégué (qui est en général le contrôleur qui a affiché la vue modale) doit implémenter deux méthodes. La première est appelée en cas de succès, avec un objet `MPMediaItemCollection`. Cet objet contient une liste d'objets `MPMediaItem` qui décrivent chacun une piste audio.

```

- (void) mediaPicker: (MPMediaPickerController *) mediaPicker
  didPickMediaItems: (MPMediaItemCollection *) collection {

    for (MPMediaItem *item in collection.items)
    {
        NSLog(@"Item: %@",
              [item valueForKeyProperty:MPMediaItemPropertyTitle]);
    }

    [self dismissModalViewControllerAnimated: YES];
}

```

POUR APPROFONDIR Les propriétés des médias

Chaque objet `MPMediaItem` contient une liste de propriétés qui permettent de le décrire de manière exhaustive : artiste, nom de la chanson, album, numéro de la piste, etc. Vous trouverez la liste exhaustive des propriétés disponibles dans la documentation de la classe `MPMediaItem`.

La deuxième méthode du délégué est appelée dans le cas où l'utilisateur ferme l'interface sans choisir de musique. On se contente alors de faire disparaître la vue modale.

```
- (void) mediaPlayerDidCancel: (MPMediaPickerController *) mediaPlayer {  
    [self dismissModalViewControllerAnimated: YES];  
}
```

Interroger directement la bibliothèque iPod

Il est également possible d'interroger directement la bibliothèque iPod à l'aide de la classe `MPMediaQuery`.

Cette classe dispose d'une propriété `items` qui par défaut renvoie l'intégralité de la bibliothèque iPod de l'utilisateur. Il est possible d'ajouter des prédicats pour filtrer la liste selon les propriétés des contenus.

```
- (void) searchAction  
{  
    MPMediaQuery *query = [[MPMediaQuery alloc] init];  
  
    NSString *search = textField.text;  
    [query addFilterPredicate:  
        [MPMediaPropertyPredicate  
            predicateWithValue:search  
            forProperty:MPMediaItemPropertyArtist]];  
  
    for (MPMediaItem *item in [query items])  
    {  
        NSLog(@"Item: %@",  
            [item valueForKey:MPMediaItemPropertyTitle]);  
    }  
    [query release];  
}
```

Contrôler l'iPod depuis l'application

Vous pouvez choisir de créer un lecteur de musique propre à votre application. Dans ce cas, la lecture en cours dans l'iPod est interrompue et votre application prend la main sur la musique.

Vous pouvez également choisir de piloter le lecteur iPod de l'utilisateur. Dans ce cas, la lecture en cours n'est pas interrompue, vous pouvez interroger le lecteur de musique pour savoir quelle est la chanson en cours (la chanson que l'utilisateur était en train d'écouter avant qu'il lance votre application), manipuler le lecteur et sa liste

de lecture. Quand l'utilisateur quittera votre application, la musique que vous avez ajoutée à la liste de lecture continuera à être lue.

REMARQUE La bibliothèque iPod est étanche

Bien que vous puissiez parcourir toutes les propriétés de la bibliothèque iPod et de l'utilisateur, et même manipuler la liste de lecture de l'iPod, il est important de remarquer que la bibliothèque reste complètement étanche.

Il n'est pas possible depuis une application de récupérer le flux audio des morceaux de l'utilisateur ou d'ajouter des chansons dans sa bibliothèque.

Quel que soit le type de lecteur que vous aurez retenu, la classe `MPMusicPlayerController` permet de contrôler le lecteur de musique. Cette classe propose deux méthodes statiques pour accéder aux lecteurs :

- `applicationMusicPlayer` et
- `iPodMusicPlayer`.

On contrôle très simplement la lecture à l'aide des méthodes `play`, `pause`, `skipToNextItem`, etc. Les propriétés `repeatMode`, `shuffleMode` et `volume` permettent de configurer le lecteur.

La liste de lecture peut être définie à l'aide des méthodes `setQueueWithQuery:` et `setQueueWithItemCollection:`. Dans les deux cas, on remplace la liste de lecture en cours.

```
MPMusicPlayerController* musicPlayer =  
    [MPMusicPlayerController iPodMusicPlayer];  
[musicPlayer setQueueWithQuery:query];  
[musicPlayer play];
```

Comme le lecteur vidéo, le lecteur de l'iPod utilise le mécanisme des notifications pour signaler que son état a changé (l'utilisateur a mis la musique en pause ou le lecteur passe à la chanson suivante).

Vous devez indiquer au lecteur que vous souhaitez recevoir les notifications à l'aide de la méthode `beginGeneratingPlaybackNotifications`. La liste de toutes les notifications est fournie dans la documentation de la classe `MPMusicPlayerController`.

Tirer partie des photos et vidéos de l'utilisateur

La classe `UIImagePickerControllerController` est un contrôleur de vue qui permet de tirer partie de l'album photo intégré dans tous les iPhone et iPod Touch, de l'appareil

photo qui est présent sur tous les iPhone et de la caméra vidéo qui est présente sur l'iPhone 3GS. Il est possible de contrôler précisément quels contrôles seront affichés à l'utilisateur pour lui permettre éventuellement de recadrer sa photo ou de sélectionner une sous-partie de sa vidéo.

L'application qui tire partie de cette classe doit définir un délégué qui sera appelé quand l'utilisateur sélectionnera une photo ou une vidéo.

Vérifier ce que peut permet le matériel

Tous les terminaux n'embarquant pas le même matériel, il est important de pouvoir vérifier dans votre application ce que permet le terminal.

La méthode de classe `isSourceTypeAvailable:` permet de savoir si une source de contenus est accessible.

```
if ([UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera])
{
    // Un appareil photo est disponible.
}
```

Type de source	Description
<code>UIImagePickerControllerSourceTypeCamera</code>	La caméra est disponible uniquement sur les iPhone.
<code>UIImagePickerControllerSourceTypePhotoLibrary</code>	La collection de photos contient toutes les photos que l'utilisateur a synchronisées avec son terminal. Il est disponible sur tous les iPhone et iPod Touch.
<code>UIImagePickerControllerSourceTypeSavedPhotosAlbum</code>	L'album photo contient toutes les images que l'utilisateur a enregistrées sur son terminal, soit depuis la caméra, soit en faisant une capture d'écran. L'album photo est disponible sur tous les iPhone et iPod Touch.

La méthode de classe `availableMediaTypesForSourceType:` permet de connaître les types de médias disponibles pour une source. Il existe deux types de médias : `kUTTypeImage` pour les images fixes, `kUTTypeMovie` pour les films.

Pour savoir si la caméra vidéo est disponible, il faut donc vérifier que la source `UIImagePickerControllerSourceTypeCamera` est disponible et qu'elle prend en charge le type de média `kUTTypeMovie`.

Paramétrer l'interface de prise de vue

On crée une instance de `UIImagePickerController` à l'aide de la méthode `init`. Puis on définit la source que l'on souhaite utiliser.

En indiquant `UIImagePickerControllerSourceTypeCamera`, on provoque l'affichage d'une vue caméra qui permet à l'utilisateur de prendre une photo ; avec les deux autres sources, l'utilisateur choisit parmi les photos qui existent déjà sur son téléphone.

Si la propriété `allowsImageEditing` est vraie, l'utilisateur pourra recadrer sa photo ou ajuster sa séquence vidéo.

```
UIImagePickerController *imagePickerController =
    [[UIImagePickerController alloc] init];

imagePickerController.sourceType =
    UIImagePickerControllerSourceTypeCamera;
imagePickerController.delegate = self;
imagePickerController.allowsImageEditing = YES;

[self presentViewController:imagePickerController
    animated:YES];
[imagePickerController release];
```

Récupérer le média de l'utilisateur

Le délégué est généralement le contrôleur de vue qui a fait appel à `UIImagePickerController`.

Il doit implémenter le protocole `UIImagePickerControllerDelegate`.

La première méthode à implémenter est celle qui est appelée si l'utilisateur annule la prise de vue. Dans ce cas, l'application doit faire disparaître le contrôleur modal.

```
- (void)imagePickerControllerDidCancel:(UIImagePickerController
*)picker
{
    [self dismissModalViewControllerAnimated:YES];
}
```

La deuxième méthode du délégué est plus intéressante. Elle reçoit en paramètre un dictionnaire qui contient des informations sur le média sélectionné par l'utilisateur.

S'il a sélectionné une photo, l'image (sous la forme d'un objet `UIImage`) est disponible dans la clé `UIImagePickerControllerEditedImage` du dictionnaire, et l'image originale dans la clé `UIImagePickerControllerOriginalImage`. Les objets `UIImage` peu-

vent être directement exploités dans l'application, on peut par exemple les afficher dans une vue `UIImageView`.

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    imageView.image = [info
        objectForKey:UIImagePickerControllerEditedImage];
    [self dismissModalViewControllerAnimated:YES];
}
```

ATTENTION La résolution des photos est bien supérieure à l'écran

En fonction du terminal utilisé, l'image peut faire jusqu'à 3 mégapixels, ce qui est bien plus que nécessaire pour afficher une image à l'écran.

Il est donc très fortement recommandé de manipuler uniquement des images réduites dans votre application, surtout si vous devez en conserver plusieurs en mémoire.

Si l'utilisateur a enregistré une nouvelle vidéo, le dictionnaire contient une URL vers cette vidéo dans la clé : `UIImagePickerControllerMediaURL`.

Conclusion

Nous venons de voir comment tirer partie des API multimédias de l'iPhone. Pour chaque API, nous avons décrit les éléments les plus importants, mais chacune regorge d'options et de fonctionnalités qui peuvent être essentielles à votre application.

Référez-vous à la documentation de chaque classe pour obtenir une description exhaustive des possibilités offertes par l'iPhone aux développeurs.

Utiliser les API de notifications

Le service de notification poussée d'Apple (*Apple Push Notification Service* ou APNS pour les intimes) vous permet de garder le contact avec vos utilisateurs, même quand votre application n'est pas lancée. C'est une des avancées les plus attendues apportée par iPhone OS 3.0, et elle permet d'envisager de nouveaux types d'applications : messagerie instantanée, systèmes d'alertes (flash info, un but est marqué, le cours d'une action dépasse un seuil, etc.).

Dans ce chapitre, nous présenterons le principe de fonctionnement du service de notification, nous montrerons comment l'utiliser au sein d'une application iPhone, et comment envoyer des notifications à l'aide du langage PHP pour la partie serveur.

Principe de fonctionnement d'APNS

Le service de notification d'Apple repose sur une connexion permanente entre les serveurs d'Apple et l'ensemble des iPhone en circulation. Cette connexion n'est active que si l'utilisateur n'a pas désactivé les notifications dans les réglages et s'il dispose d'une connexion de données.

REMARQUE Notification et iPod Touch

Les iPod Touch bénéficient aussi de ce service, mais uniquement lorsque leur connexion réseau est active (c'est-à-dire qu'ils doivent être à portée d'une borne Wi-fi et que leur écran doit être allumé).

Qu'est-ce qu'une notification ?

Une notification est un moyen de signaler à l'utilisateur qu'une information est disponible dans votre application. Elle peut prendre la forme d'une boîte de dialogue, d'un son, d'une pastille (*badge* en anglais) sur l'icône de l'application ou d'une combinaison de ces éléments.

Pré-requis pour l'utilisation du service de notification

Pour pouvoir utiliser le service de notification, votre application doit avoir son propre identifiant d'application (AppID) et le nom d'application renseigné dans le AppID doit correspondre exactement à l'identifiant de package (*Bundle Identifier*) dans le fichier de propriété de votre projet (*Info.plist*).

D'autre part, vous devez avoir activé le service de notification pour cet identifiant d'application. Il faut régénérer et retélécharger le fichier de provisionnement après avoir activé les notifications ; en effet, au moment de s'inscrire au service, l'iPhone vérifie dans le fichier de provisionnement que les notifications sont activées.

ASTUCE Pour vérifier que tout est prêt

Ouvrez le fichier de provisionnement de votre application avec un éditeur de texte et cherchez la section *Entitlements*.

```
<key>Entitlements</key>
<dict>
  <key>application-identifiant</key>
  <string>7E4N8Z534B.com.masociete.monapp</string>
  <key>aps-environment</key>
  <string>development</string>
  <!-- [...] -->
</dict>
```

La chaîne de caractères correspondant à la clé `application-identifiant` doit reprendre exactement la chaîne du *Bundle Identifier* défini dans votre fichier *Info.plist*. Le préfixe unique (les lettres et les chiffres) est ajouté automatiquement par Xcode et ne doit pas apparaître dans le fichier *Info.plist*.

La clé `aps-environment` doit apparaître aussi. Elle indique que votre application est autorisée à utiliser le service APNS en développement.

Si vous rencontrez des difficultés, n'hésitez pas à effacer le fichier de provisionnement de votre iPhone et de votre ordinateur et à le réinstaller après l'avoir retéléchargé.

Les notifications en quatre étapes

L'utilisation du service de notification se déroule en quatre grandes étapes :

- 1 Lorsqu'une application prévoit d'utiliser les notifications, elle doit s'enregistrer auprès du système qui, après confirmation de l'utilisateur, lui donne un jeton unique pour ce terminal et cette application (*deviceToken*).
- 2 L'application transmet ce jeton à un serveur qui appartient au développeur de l'application.
- 3 Lorsque le serveur de l'application veut envoyer une notification, il envoie un message aux serveurs Apple via un canal sécurisé en fournissant le jeton.
- 4 La notification est reçue par l'iPhone et affichée à l'utilisateur.

Nous détaillerons ces quatre étapes dans ce chapitre.

Étape 1 : inscription au service de notification

L'inscription au service de notification se fait depuis le code de votre application en appelant la méthode `registerForRemoteNotificationTypes:`, de la classe `UIApplication`.

ATTENTION Pas de notification dans le simulateur

Le service de notification n'est pas utilisable dans le simulateur, et toute tentative d'enregistrement renverra systématiquement une erreur. Ne soyez pas surpris !

Vous indiquez en paramètre les types de notifications que vous souhaitez recevoir. Si c'est la première fois que l'application s'inscrit, l'iPhone demande une confirmation à l'utilisateur.

L'inscription est une opération relativement longue qui nécessite un échange avec le service APNS. Aussi, vous n'obtenez pas de réponse immédiatement et le processus d'inscription est asynchrone.

Lorsqu'une réponse est obtenue (ou qu'une erreur est détectée) des méthodes du délégué de votre application sont appelées.

La méthode `didRegisterForRemoteNotificationsWithDeviceToken:` est appelée lorsque l'inscription s'est déroulée correctement. Elle reçoit un paramètre du type `NSData` qui contient le jeton de notification.

La méthode `didFailToRegisterForRemoteNotificationsWithError:` reçoit en paramètre un objet `NSError` qui donne la raison de l'erreur (l'utilisateur a refusé, l'application s'exécute dans le simulateur, le réseau n'est pas disponible, etc.).

L'ensemble de ces opérations est généralement fait directement dans le délégué de l'opération. Le code ci-après montre comment déclencher l'inscription et les deux méthodes déléguées qui doivent être implémentées.

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // [ Initialisation de l'application ... ]
    [window addSubview:myMainViewController.view];
    [window makeKeyAndVisible];

    // Inscription aux notifications
    [application registerForRemoteNotificationTypes:
     UIRemoteNotificationTypeAlert
     |UIRemoteNotificationTypeBadge
     |UIRemoteNotificationTypeSound];
}

- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken
{
    NSLog(@"Enregistrement auprès de APNS réussi.");
    // ...
}

- (void)application:(UIApplication *)application
didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
{
    NSLog(@"Erreur lors de l'enregistrement APNS : %@", [error
    localizedDescription]);
}
```

BEST PRACTICE Répéter l'inscription à chaque lancement de l'application

Il est recommandé par Apple de répéter l'inscription à chaque lancement de l'application. Vous éviterez ainsi des problèmes de synchronisation entre le parc de terminaux et la base de données des inscrits sur vos serveurs.

De plus, si l'utilisateur restaure sa sauvegarde sur un autre iPhone, l'identifiant changera.

Étape 2 : Transmettre le jeton APNS à votre serveur

Une fois que votre application a obtenu le jeton pour le service de notification, elle doit le transmettre à votre serveur.

Il n'y a aucune règle sur la manière de mettre en place cet échange. Vous pouvez par exemple utiliser un simple appel HTTP et passer en paramètre une chaîne de caractères avec le jeton.

Sur votre serveur, il faudra enregistrer dans une base de données ce jeton et éventuellement d'autres paramètres utiles à votre application (l'identifiant de l'utilisateur, la liste des alertes auxquelles il est inscrit, etc.).

L'exemple ci-après permet de convertir le jeton en une chaîne de caractères hexadécimaux et de lancer une connexion réseau en tâche de fond.

```
NSString const *baseUrl = @"http://www.monserveur.com";

// Cette méthode est appelée par le délégué d'application
// quand le jeton est reçu.
- (void) registerDeviceToken:(NSData*) deviceToken
{
    [self performSelectorInBackground:@selector(doRegisterDeviceToken:)
    withObject:deviceToken];
}

// Cette méthode exécute la requête HTTP en tâche de fond.
- (void) doRegisterDeviceToken:(NSData*) deviceToken
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSString *deviceTokenString = [self serializeDeviceToken:deviceToken];

    NSLog(@"Enregistrement du jeton: %@", deviceTokenString);
    NSURL *url = [NSURL URLWithString:[NSString stringWithFormat:@"%s%/
    register.php?devicetoken=%s", baseUrl, deviceTokenString]];
    [self performHTTPRequestForUrl:url];

    [pool release];
}

// Cette méthode permet de convertir le jeton en hexadécimal
+ (NSString*) serializeDeviceToken:(NSData*) deviceToken
{
    NSMutableString *str = [NSMutableString stringWithCapacity:64];
    int length = [deviceToken length];
    char const* bytes = [deviceToken bytes];
```

```

for (int i = 0; i < length; i++)
{
    [str appendFormat:@"%02.2hhX", bytes[i]];
}

return str;
}

```

Étape 3 : Envoyer les notifications à votre application

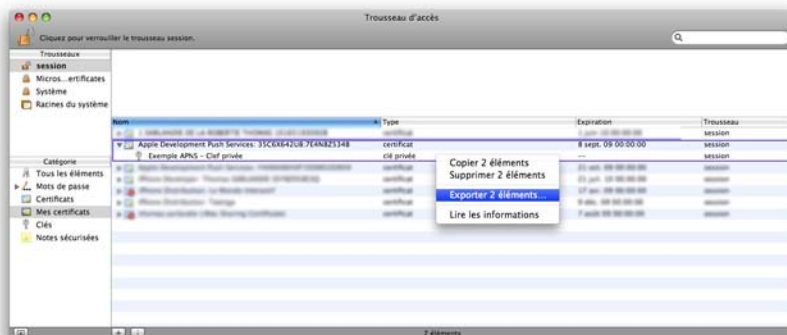
L'envoi de notification se fait au sein d'une connexion TCP ouverte entre vos serveurs et les serveurs d'Apple. Cette connexion est cryptée grâce au protocole SSL et un échange de certificats permet de valider l'identité des deux extrémités.

Obtenir un certificat SSL pour le service APNS

Vous générez le certificat SSL pour l'APNS de la même façon que vous avez généré un certificat de développeur.

- 1 Vous devez tout d'abord vous rendre dans l'*iPhone Program Portal* et accéder à la page de détails du *AppId* de votre application. Le service APNS devrait déjà être activé, il vous reste à cliquer sur le bouton *Configure* pour le configurer en environnement de développement.
- 2 En suivant les instructions du portail, vous générez une demande de certificat sur votre machine de développement et vous l'envoyez au serveur Apple.
- 3 Après quelques secondes, le certificat est disponible au téléchargement, il faut l'enregistrer sur votre machine et l'installer dans votre trousseau de clés.

Figure 15-1
Export du certificat et de la clé privée depuis le trousseau d'accès



- 4 Ouvrez ensuite l'application Trousseau d'accès, sélectionnez le certificat *et* la clé
- 5 Puis avec l'outil *Fichier > Exporter des éléments*, faites un export au format P12. Le trousseau d'accès vous propose de saisir un mot de passe pour protéger le fichier exporté, laissez un mot de passe vide.

Nous utilisons ensuite l'outil en ligne de commande `openssl` pour convertir la clé dans un format utilisable directement par du code PHP.

```
## Extraction du certificat vers le fichier cert.pem
$ openssl pkcs12 -clcerts -nokeys -out cert.pem -in Certificat.p12
# L'outil vous demande le mot de passe d'import du fichier, appuyez sur
Entrée.

## Extraction de la clé privée vers le fichier key.pem
$ openssl pkcs12 -nocerts -out key.pem -in Certificat.p12
# openssl vous demande encore le mot de passe d'import
# puis il vous demande de saisir un mot de passe pour protéger la clé.
Vous devez saisir un mot de passe d'au moins 4 caractères.

## On supprime ensuite ce mot de passe sur la clé
$ openssl rsa -in key.pem -out key.unencrypted.pem
# Vous devez ressaisir le mot de passe choisi à l'étape précédente

## On crée un fichier unique regroupant le certificat et la clé
$ cat cert.pem key.unencrypted.pem > ck.pem
```

Vous pouvez vérifier que toutes les étapes jusqu'ici ont été réalisées correctement en utilisant la commande `openssl` pour ouvrir une connexion aux serveurs Apple.

```
openssl s_client -connect gateway.sandbox.push.apple.com:2195 -cert
ck.pem
```

Openssl affiche sur la sortie standard le détail de l'échange de clés et de la vérification des certificats.

Si la commande `openssl` ne vous permet pas de taper du texte et qu'elle rend directement la main, il y a un problème avec votre certificat. Reprenez les étapes une par une.

Si tout s'est bien passé, vous devriez pouvoir taper du texte qui est envoyé au serveur Apple lorsque vous appuyez sur entrée. Cela doit provoquer la fermeture de la connexion (en effet, il est peu probable que vous ayez respecté le protocole attendu par Apple).

```
$ openssl s_client -connect gateway.sandbox.push.apple.com:2195 -cert ck.pem
CONNECTED(00000004)

[...]
```



```

MRmYhpAYxvArJZxVkX63USZ1iLy3k4bfbD0LGUvFzMJP9W2jGwauFWE=
-----END CERTIFICATE-----
subject=/C=US/ST=CALIFORNIA/L=Cupertino /O=Apple Inc/OU=ISSE/
CN=gateway.sandbox.push.apple.com
issuer=/C=US/O=Entrust.net/OU=www.entrust.net/CPS incorp. by ref. (limits
liab.)/OU=(c) 1999 Entrust.net Limited/CN=Entrust.net Secure Server
Certification Authority
---
No client certificate CA names sent
---
SSL handshake has read 1525 bytes and written 2003 bytes
---
New, TLSv1/SSLv3, Cipher is AES256-SHA
Server public key is 1024 bit
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol : TLSv1
    Cipher   : AES256-SHA
    Session-ID:
    Session-ID-ctx:
    Master-Key:
8DA2920F19A1D3D43DE3251AF5D07273FEB84515BDB6C9A242C670D99FBCCBB0D2981535FBD4D7F
37D4683283EA08DDF
    Key-Arg : None
    Start Time: 1249843791
    Timeout : 300 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---
Hi Guys!
closed

```

Bravo, vous avez réussi à regrouper la clé privée et le certificat dans le fichier `ck.pem`. Vous pouvez ensuite utiliser directement ce fichier sur vos serveurs. C'est sans aucun doute l'étape la plus difficile dans la mise en place du service de notification !

BEST PRACTICE Prendre grand soin de ses certificats

Vos certificats doivent être protégés avec soin. Si un tiers mal intentionné met la main dessus, il aura la possibilité d'envoyer des notifications en masse à vos utilisateurs. C'est votre image de marque qui est en jeu !

Envoyer une notification depuis le serveur

La connexion sécurisée avec les serveurs Apple vous permet d'envoyer un grand nombre de notifications à la suite. Le moyen le plus efficace pour envoyer un grand nombre de notifications en peu de temps est de n'ouvrir qu'une seule connexion vers APNS et de pousser toutes les notifications dans cette même connexion.

Préparer le message de notification

Chaque notification est décrite par un dictionnaire encodé en JSON. Certaines clés sont imposées par le service, et vous pouvez ajouter vos propres données, mais l'ensemble, une fois encodé, ne doit pas dépasser 256 octets.

L'exemple suivant montre comment préparer un message de notification qui présente un message dans une boîte de dialogue, joue le son par défaut et affiche la valeur 1 dans la pastille de l'application.

```
$body = array();
$body['aps'] = array();
$body['aps']['message'] = "Vous avez reçu une notification!";
$body['aps']['badge'] = 1;
$body['aps']['sound'] = "default";
$payload = json_encode($body);
```

Vous pouvez demander la lecture de vos propres fichiers sons en indiquant le nom d'un fichier audio contenu dans votre package d'application.

Envoi du message

L'envoi du message se fait avec un format binaire spécifique APNS. On concatène quelques caractères spéciaux, le jeton du terminal, et la notification.

```
// Le jeton du terminal
$deviceToken = "";

// Création d'un contexte SSL
$cctx = stream_context_create();
// Ouverture du fichier PEM contenant clé privée et certificat
stream_context_set_option($cctx, 'ssl', 'local_cert', './ck.pem');

// Ouverture d'une connexion réseau vers les serveurs de développement
Apple
$fp = stream_socket_client('ssl://gateway.sandbox.push.apple.com:2195',
    $err, $errstr, 60, STREAM_CLIENT_CONNECT, $cctx);
```

```
if (!$fp) {
    print "Erreur de connexion: $err $errstr\n";
    return;
}
else {
    print "Connection OK\n";
}

// Construction des données à envoyer
$msg = chr(0) . pack("n",32)
    . pack('H*', str_replace(' ', '', $deviceToken))
    . pack("n",strlen($payload)) . $payload;

print "Message: " . $payload . "\n";

// Envoi des données
fwrite($fp, $msg);

// Fermeture de la connexion
fclose($fp);
```

CONSEIL On ne réinvente pas la roue !

L'exemple présenté ici a été simplifié à l'extrême pour tenir sur ces pages. Il permet juste de valider le bon fonctionnement de toute la chaîne.

Vous trouverez dans les forums Apple des exemples de code bien plus complets et déjà vérifiés permettant de maintenir la connexion ouverte vers les serveurs APNS, d'envoyer plusieurs messages et ce pour tous les langages modernes.

Étape 4 : Recevoir les notifications

Deux cas de figure peuvent se présenter : soit votre application est fermée lorsque la notification est reçue ; soit votre application est déjà ouverte.

Réception des notifications quand l'application est fermée

Lorsque votre application est fermée, les notifications sont signalées à l'utilisateur par l'ajout d'une pastille, la lecture d'un son et éventuellement l'affichage d'un message.

Si vous avez demandé l'affichage d'un message, l'utilisateur a la possibilité de cliquer sur le bouton « Voir » pour ouvrir votre application.

Si vous le souhaitez, vous pouvez implémenter dans votre délégué d'application, la méthode `application:didFinishLaunchingWithOptions:` qui sera appelée à la place de `application:didFinishLaunching:`. Elle reçoit un paramètre qui est un dictionnaire contenant la notification. Il vous est ainsi possible de récupérer des informations que vous aurez ajoutées dans la notification et de les utiliser pour ouvrir votre application dans le contexte de la notification.

Si vous n'implémentez pas cette méthode, votre application est lancée normalement.

Réception des notifications lorsque l'application est ouverte

Lorsque l'iPhone reçoit une notification pour votre application et que celle-ci est déjà ouverte, il appelle la méthode `application:didReceiveRemoteNotification:` de votre délégué d'application en lui passant en paramètre un dictionnaire contenant la notification.

L'implémentation par défaut de cette méthode ne fait rien. C'est à vous de l'implémenter si vous souhaitez gérer les notifications quand votre application s'exécute.

Détecter les désinscriptions et les erreurs

Il peut arriver que les messages n'arrivent pas jusqu'à l'utilisateur. C'est en particulier le cas s'il désinstalle l'application ou si son iPhone est hors d'usage (perdu ou cassé).

Pour éviter que vos serveurs ne continuent à envoyer des notifications à des utilisateurs qui ne les reçoivent plus, Apple a mis en place un mécanisme (*the Feedback Service*) permettant de récupérer la liste des jetons vers lesquels il ne faut plus envoyer de message.

Ce mécanisme s'appuie sur une connexion SSL vers un port différent (2196) mais avec le même certificat. À chaque fois que vous vous connecterez, le serveur Apple renverra la liste de tous les jetons qui sont en erreur et videra cette liste. Vous devez alors désinscrire ces utilisateurs.

Si vous ne vous connectez pas régulièrement à ce service, Apple peut révoquer le certificat que vous utilisez pour envoyer les notifications, ce qui aurait comme effet de ne plus du tout pouvoir envoyer de notifications.

Conclusion

Dans ce chapitre, nous avons présenté ce qui est sans aucun doute l'une des nouveautés les plus intéressantes de l'iPhone OS 3.0.

Les principales difficultés qui peuvent survenir lors de la mise en place de ce service ont été évoquées mais attention, la mise en place d'un service de notifications peut être extrêmement consommatrice de ressources pour vos serveurs. Vous devez ainsi réfléchir très sérieusement à l'architecture de votre plate-forme, car si votre application a du succès, la fréquence d'envoi des notifications peut rapidement grimper à plusieurs milliers par jour.

CINQUIÈME PARTIE

La publication des applications

Cette dernière partie contient un seul chapitre, consacré à la publication des applications sur l'App Store. Il est destiné à toute l'équipe projet qui y trouvera la liste des éléments à préparer avant de soumettre leur application, ainsi que des conseils pour favoriser les chances qu'elle soit validée du premier coup.

Enfin, nous y présentons quelques outils permettant de suivre le succès de l'application et d'en préparer la prochaine version.

16

Publier sur l'App Store

La publication sur l'App Store est la dernière étape pour vous... mais la première apparition publique pour votre application.

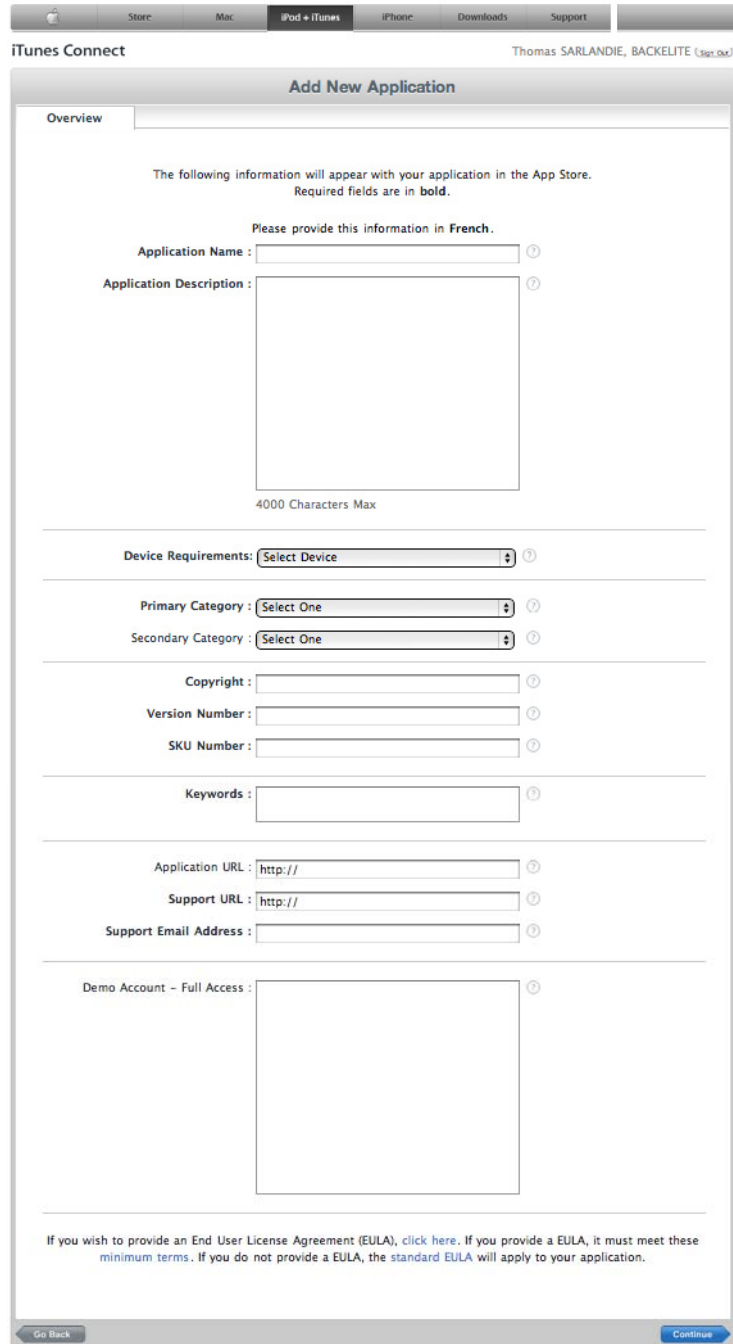
C'est le moment de préparer les éléments marketing qui feront le premier lien entre vos futurs utilisateurs et votre application ; c'est aussi le moment de s'assurer que tout a bien été fait. En effet, une fois la soumission effectuée, il n'est plus possible de retoucher à votre application – sauf à recommencer l'ensemble du processus de validation.

Dans ce chapitre, nous aborderons tous les aspects de la publication, depuis la description détaillée de l'ensemble des éléments à fournir lors de la soumission, jusqu'à la liste des vérifications à effectuer avant l'envoi de l'application à Apple. Et sans omettre, bien sûr, de précieux conseils de lancement directement issus d'une expérience riche en publications.

Préparer les éléments marketing en vue de la publication

C'est au moment de la soumission que vous fournissez tous les éléments qui représentent votre application dans l'App Store sur iPhone et dans iTunes. Du nom de l'application aux captures d'écran, en passant par l'icône de votre application, il faut porter autant d'attention à la préparation de ces éléments qu'à votre application elle-même.

Figure 16-1
L'écran d'ajout d'une
application dans iTunes
Connect



Store Mac iPod + iTunes iPhone Downloads Support

iTunes Connect Thomas SARLANDIE, BACKELITE (Sign Out)

Add New Application

Overview

The following information will appear with your application in the App Store.
Required fields are in **bold**.

Please provide this information in **French**.

Application Name : ⓘ

Application Description : ⓘ

4000 Characters Max

Device Requirements: ⓘ

Primary Category : ⓘ

Secondary Category : ⓘ

Copyright : ⓘ

Version Number : ⓘ

SKU Number : ⓘ

Keywords : ⓘ

Application URL : ⓘ

Support URL : ⓘ

Support Email Address : ⓘ

Demo Account - Full Access : ⓘ

If you wish to provide an End User License Agreement (EULA), [click here](#). If you provide a EULA, it must meet these [minimum terms](#). If you do not provide a EULA, the [standard EULA](#) will apply to your application.

Go Back Continue

Nom de société et langue principale

La première fois que vous soumettez une application via iTunes Connect, vous devrez choisir un nom (celui de votre société ou le vôtre) qui apparaîtra sous le nom de votre application dans l'App Store et dans iTunes. Choisissez bien ce nom car vous ne pourrez plus le modifier par la suite, ce sera le même pour toutes vos applications.

Vous devez également choisir une langue principale qui est la langue dans laquelle vous fournirez les informations sur vos applications. Pour un éditeur français distribuant des applications francophones, ce sera certainement le français.

Nous verrons plus tard qu'il est possible de fournir des informations dans d'autres langues. Si ces informations n'ont pas été fournies pour la langue de l'utilisateur, il verra les informations de la langue par défaut. Un éditeur distribuant des applications pour le monde entier voudra donc probablement choisir l'anglais comme langue par défaut.

Le nom de l'application

C'est le nom qui apparaîtra dans l'App Store et dans iTunes. Sa taille est limitée à 255 caractères. Il ne doit pas contenir le numéro de version ni de marque dont vous ne détenez pas les droits (il est donc interdit d'inclure le mot « iPhone » dans le nom de votre application).

Ce nom peut être différent du nom qui apparaîtra sur le menu de l'iPhone. Il est donc possible d'utiliser un peu plus de place. Néanmoins, il vaut mieux garder un nom très proche pour permettre à vos utilisateurs de retrouver facilement l'application dans leurs icônes et partager votre application avec des amis.

RAPPEL Choisir le nom qui apparaîtra dans le menu de l'iPhone

Le nom qui apparaît dans l'iPhone est celui défini dans le fichier `Info.plist` de votre application.

Description de l'application

La description de l'application est un texte libre à saisir. Il est limité à 4000 caractères, mais Apple recommande de ne pas dépasser 700 caractères pour que le texte reste facilement lisible sur l'iPhone.

Vous ne pouvez saisir que du texte, sans formatage (les tags HTML sont automatiquement retirés).

Terminaux ciblés

Vous pouvez choisir de limiter votre application à un certain type de terminaux :

- iPhone uniquement ;
- iPhone & iPodTouch de deuxième génération ;
- tous les iPhone et iPod Touch.

La première catégorie s'appliquera pour toutes les applications qui ont besoin des composants présents uniquement dans l'iPhone, comme le GPS ou l'appareil photo.

La deuxième catégorie permet de ne cibler que les terminaux équipés d'un micro (l'iPod Touch de deuxième génération n'a pas de micro, mais peut être utilisé avec le casque micro).

La troisième catégorie permet de cibler tous les terminaux.

SKU : référence de l'application

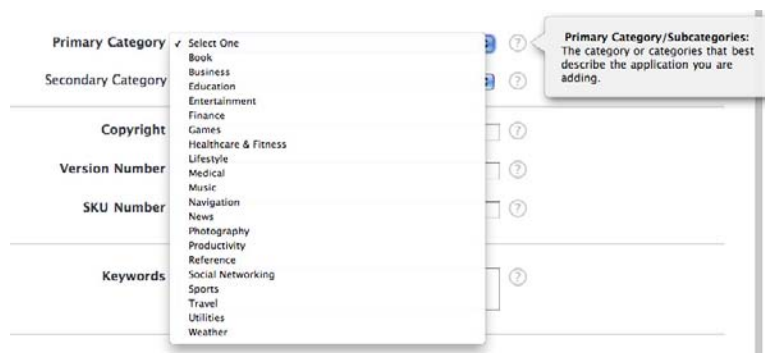
Vous devez fournir un identifiant numérique unique qui servira à identifier cette application dans les rapports financiers fournis par Apple. Il ne sera pas vu des utilisateurs. Les nombres 1 ou 42 feront parfaitement l'affaire.

Catégorie de l'application

Vous devez choisir à quelle catégorie appartient votre application. Les utilisateurs peuvent en effet rechercher les applications par catégorie dans l'App Store.

Figure 16-2

Vous pouvez choisir une ou deux catégories pour votre application.



Il est possible de choisir une catégorie secondaire, l'application apparaîtra lorsqu'un utilisateur lancera une recherche sur cette catégorie, mais elle n'apparaîtra pas dans les pages de cette catégorie.

Numéro de version de l'application

Le numéro de version est une chaîne de caractères du type 1.0 ou 1.0.0.

Détenteur des copyrights

Vous devez indiquer le nom de la personne ou de la société qui détient les droits sur l'application, ainsi que l'année de création de l'application (c'est le mécanisme anglo-saxon de droits d'auteur).

Exemple : 2009, Worldwide Company

Mots-clés

Il est possible de fournir une liste de mots-clés qui décrivent votre application. Cette liste est utilisée par le moteur de recherche de l'App Store.

Informations de contact

Vous devez indiquer une URL qui pointe vers une page web dédiée à l'application ou à la marque de l'application.

Vous devez également indiquer l'URL d'une page fournissant une assistance ou une documentation pour votre application. Ce peut être la même URL.

Enfin, Apple vous demande de fournir une adresse email à laquelle ils pourront vous contacter pour obtenir de l'aide sur l'application (en particulier dans le cas où ils recevraient beaucoup de demandes sur leur support technique). Cette adresse email n'est jamais communiquée aux utilisateurs de votre application.

Informations de démonstration (Demo account)

Si votre application permet à un utilisateur de se connecter avec un compte, vous devez fournir à Apple un compte de test qui permettra aux équipes en charge de la revue des applications de tester toutes les fonctionnalités de l'application.

Contrat de licence

Vous pouvez fournir votre propre contrat de licence avec votre application. Si vous le faites, votre contrat de licence doit être compatible avec celui d'Apple et contenir certaines clauses imposées par Apple. Vous trouverez plus d'informations à ce sujet dans le guide développeur iTunes Connect, téléchargeable depuis iTunes Connect.

Niveau de contrôle parental de l'application (Ratings)

Un niveau de contrôle parental vous est automatiquement proposé sur la base de quelques questions du type « Est-ce que votre application contient de la violence ? Pas du tout/Un peu/Souvent ».

CONSEIL Soyez plus que rigoureux en répondant à ces questions

Depuis la mise en place du système de contrôle parental pour toutes les applications (avec l'arrivée de l'iPhone OS 3.0), de nombreuses applications ont été rejetées parce que les développeurs avaient indiqué un niveau de contrôle parental très bas alors qu'Apple a réussi à trouver dans l'application des contenus « offensants ».

En particulier, il semble que toutes les applications qui reprennent des flux d'actualités ou des contenus web sont systématiquement rejetées si elles ne sont pas au moins en 9+ ou 12+.

Pour de nombreux développeurs, il vaut mieux viser haut et indiquer 12+ ou 17+ plutôt que de risquer un rejet de l'application qui nécessite un nouveau cycle complet de validation.

Pays de distribution

Vous pouvez choisir les App Store dans lesquels vous souhaitez vendre votre application. Il est ainsi possible de limiter une application à un ou quelques pays.

CONSEIL Visez large

Le fait que votre application soit uniquement en français ne signifie pas qu'elle n'intéressera personne à l'étranger. Il y a de nombreux Français expatriés et encore plus de francophones dans le monde.

N'oubliez pas que le compte iTunes est rattaché à un pays en fonction de l'adresse de résidence de l'utilisateur et du pays de sa banque, pas en fonction de sa langue ou de sa culture.

Éléments graphiques

Durant le processus de soumission, vous devrez fournir plusieurs éléments graphiques.

Icône de l'application

L'icône de votre application doit être fournie en haute résolution. Le format attendu par Apple est une image au format TIFF ou PNG, de 512 x 512 pixels avec une résolution de 72 dpi.

Cette image vient en plus de l'image déjà fournie dans le package de votre application (qui, elle, doit faire 57 x 57 en PNG 24 bits) ; elle est utilisée dans l'iTunes App Store (où une image de 57 x 57 ne serait pas suffisamment détaillée).

RAPPEL La propriété `UIPrerenderedIcon`

Par défaut, l'iPhone et l'App Store ajoutent automatiquement les bords arrondis et l'effet de brillant sur les icônes. Si vous souhaitez fournir une image qui ne soit pas retouchée, vous pouvez ajouter la propriété `UIPrerenderedIcon` avec la valeur booléenne vraie dans le fichier `Info.plist` de votre application.

Captures d'écran de l'application

Vous pouvez fournir de 1 à 5 captures d'écran pour votre application. Apple demande à ce que la barre de statut soit retirée des captures d'écran.

À moins que vos applications soient des jeux ou des applications plein écran, sans barre de statut, vos captures devraient donc faire 320 x 460 pixels (mode portrait) ou 480 x 300 pixels (mode paysage).

ASTUCE Prendre des captures d'écran avec l'iPhone

Il est extrêmement facile de faire des captures d'écran d'une application iPhone. Il suffit d'appuyer simultanément sur le bouton Home et le bouton Power de l'iPhone. Les captures sont enregistrées dans les photos de l'iPhone et peuvent être transférées avec iTunes.

Date de disponibilité de l'application

Votre application ne sera pas visible sur l'App Store avant cette date. Il est donc possible (et très fortement recommandé) de soumettre une application très en avance, et de la laisser cachée jusqu'à votre vraie date de lancement. C'est d'ailleurs le seul moyen de maîtriser la date à laquelle elle apparaîtra sur l'App Store.

L'application ne sera de toute façon pas visible tant qu'elle n'aura pas été approuvée. Vous pouvez indiquer la date de la soumission comme date de disponibilité pour que l'application soit disponible dès qu'elle aura été validée.

CONSEIL La modification de cette date est possible à tout moment

Cette date est modifiable à tout moment. Vous pouvez donc indiquer une date lointaine, et attendre que l'application soit validée pour ramener cette date à une échéance plus proche.

Il est ainsi possible d'accompagner la publication sur l'App Store d'une campagne de communication (bannières, articles dans les blogs, etc.).

Prix de l'application

Vous ne définissez pas directement le prix de l'application mais une catégorie. Chaque catégorie a une valeur dans tous les App Store. Ainsi, les applications de la catégorie 1 sont vendues 0,99 \$ aux États-Unis et 0,79 € en Europe.

Au moment de remplir ces informations, vous verrez le reversement net (après déduction des taxes applicables dans chaque pays) qui vous sera versé en fonction du prix de vente.

RAPPEL Anticiper la mise en place du contrat de vente Apple

Pour vendre des applications sur l'App Store, il faut avoir signé et renvoyé à Apple un contrat spécifique (en plus des documents fournis lors de l'inscription au programme développeur) ainsi qu'un RIB pour permettre à Apple de vous verser vos gains.

Vous devez entreprendre ces démarches bien avant la soumission car elles peuvent facilement prendre deux ou trois semaines ; même une fois validée par Apple, votre application n'apparaîtra pas dans l'App Store si ces éléments n'ont pas été reçus et vérifiés par Apple.

Localisation de votre application

Si vous le souhaitez, fournissez les éléments marketing en plusieurs langues. Il est ainsi possible de fournir des éléments spécifiques pour chaque langue.

Les éléments suivants peuvent être renseignés à nouveau pour chaque langue :

- 1 le nom de l'application ;
- 2 la description de l'application ;
- 3 les mots-clés associés à l'application ;
- 4 les deux URL de l'application ainsi que l'adresse email de support ;
- 5 les cinq captures d'écran.

REMARQUE Localisation de l'application

Bien que nous n'ayons pas abordé ce sujet, la localisation d'application est grandement facilitée par les API Cocoa. Elles permettent de fournir une seule application contenant plusieurs langues, et que la langue soit choisie automatiquement en fonction des préférences de l'utilisateur.

Éléments techniques et dernières vérifications

L'application est transmise à Apple sous la forme d'un fichier ZIP contenant le package de votre application, signé avec une clé de distribution.

Fournir l'application à Apple

La préparation de la clé de distribution est très similaire à la création d'une clé développeur que nous avons vue dans le premier chapitre de ce livre et est abondamment documentée dans le portail développeur.

Vous devez créer un profil de compilation dédié à la distribution et signer l'application avec la clé de distribution en utilisant un fichier de provisionnement dédié. L'application ainsi compilée ne peut pas être installée sur un iPhone tant qu'elle n'est pas passée par Apple.

Dernières vérifications techniques

Avant d'appuyer sur le bouton pour compiler l'application qui sera ensuite téléchargée par des millions de personnes et vous rendra célèbre, il est utile de prendre le temps de faire quelques vérifications.

Mode de compilation et niveau de log

L'application soumise à Apple devrait toujours être compilée en mode *Release* ce qui signifie que les symboles de débogage sont supprimés et que l'application sera ainsi plus légère.

Si vous utilisez la console pour afficher des informations (via la fonction `NSLog` ou autre) vous devez absolument réduire au maximum les informations affichées sur la console. En effet, elles ne seront jamais vues par les utilisateurs, et ne vous serviront plus à déboguer l'application ; par contre, elles ont un impact très important sur les performances. Il faut en particulier éviter les boucles qui impriment une ligne pour chaque passage dans la boucle, les messages de débogage dans les parseurs XML, etc.

Vérifier le contenu du package applicatif

Vous avez abondamment testé l'application et vous êtes donc certain qu'elle contient tous les éléments nécessaires à son bon fonctionnement. Il est important de vérifier aussi qu'elle n'en contient pas trop.

Avez-vous pensé à supprimer toutes les ressources qui ne sont plus utilisées ? Êtes-vous sûr que les fichiers de votre gestionnaire de source (les répertoires `.svn` par exemple) n'ont pas été compressés avec votre application ?

Il est très facile d'explorer le contenu de l'application (*ctrl-clic* sur l'application puis *Afficher le contenu du paquet*), prenez le temps de vérifier qu'il n'y a pas de superflu.

Respect de la charte graphique Apple

Apple est extrêmement vigilant quant au respect des règles du guide ergonomique et à la bonne utilisation des éléments standard.

Faites en particulier très attention lorsque vous utilisez des icônes ou des boutons fournis par le SDK à bien les utiliser dans le contexte prévu par Apple. Le non-respect des règles ergonomiques entraînerait le rejet systématique de l'application.

Messages d'erreurs réseau

Avez-vous testé votre application dans un tunnel ? Que se passe-t-il si votre application n'obtient pas de réponse réseau ? Avez-vous géré proprement ces types d'erreurs ?

Les équipes Apple vont vérifier ce cas et si vous voulez éviter que l'application ne soit rejetée, il faut absolument prendre le temps d'ajouter des messages d'erreur expliquant à l'utilisateur ce qui ne va pas en lui indiquant éventuellement qu'avoir une connexion réseau est un pré-requis pour utiliser l'application.

ASTUCE Tests de l'application sans réseau

Il n'est pas toujours facile de trouver un tunnel dans lequel aucun réseau ne passe. Heureusement, vous pouvez tester le fonctionnement sans réseau de votre application en activant simplement le mode avion dans les préférences.

Conserver le fichier de symbole

Lors de la compilation, Xcode produit un package applicatif (le `.app`) et un fichier de symbole (avec l'extension `.app.dSYM`).

Vous devez conserver précieusement le fichier de symbole correspondant à la version de l'application que vous soumettez sur l'App Store. En effet, à l'aide de ce fichier, vous pourrez analyser les rapports d'erreurs remontés par les utilisateurs et faire correspondre les adresses mémoires aux symboles (variables et fonctions).

Après la soumission

Une fois l'application soumise, vous devez attendre la validation d'Apple. Si vous avez fait des erreurs, tout n'est pas perdu.

REMARQUE Délais de validation

Les délais de validation ont beaucoup varié depuis le lancement de l'App Store. En fonction des périodes, il peut être nécessaire d'attendre de 1 à 4 semaines pour avoir un retour d'Apple.

Le site tente de mesurer le délai moyen de validation en surveillant les applications publiées chaque jour et en comparant la date d'apparition avec la date de disponibilité (qui correspond en général à la date de soumission). Apple fournit également quelques indicateurs dans le portail du programme développeur (en particulier le pourcentage d'applications validées en moins de deux semaines).

Modification des éléments marketing

Tous les éléments marketing peuvent être édités à l'aide du bouton *Edit Information* qui apparaît sous l'icône de l'application dans iTunes Connect.

Vous pouvez donc modifier le nom, la description de l'application, ses captures d'écran, etc. Attention, il n'est pas possible de modifier le classement de l'application (3+, 17+, etc.) sans repasser par une soumission de l'application.

Modification de l'application

Si vous découvrez un bogue après la soumission de votre application, vous pouvez décider de la rejeter (*Reject Binary*) et d'en soumettre une nouvelle version. Sachez cependant que dans ce cas, vous repartirez du bas de la file d'attente des applications à valider.

En cas de rejet

Si votre application est rejetée, vous recevrez un email vous indiquant les points bloquants qui ont empêché votre application d'être validée. Vous devez alors soumettre un nouveau binaire en ayant pris en compte les retours Apple.

Sachez que les retours ne sont pas nécessairement exhaustifs et qu'il est possible que l'application soit encore rejetée pour d'autres raisons. Les équipes de validation ont une liste de vérifications à faire, et elles s'arrêtent en général au premier point bloquant.

Votre application est publiée

Bravo ! C'est le début de l'aventure App Store, et vous allez pouvoir commencer à suivre votre application dans le classement de sa catégorie, dans le classement général, lire les commentaires des utilisateurs, etc.

Suivre les progrès de votre application

Une fois l'application publiée, Apple met à votre disposition différents outils vous permettant de suivre au quotidien son succès.

Statistiques de téléchargement

Vous trouverez dans iTunes Connect les statistiques de téléchargement de votre application. Elles sont fournies par jour, par semaine et par mois. Les rapports quotidiens et hebdomadaires sont effacés automatiquement après une courte période, vous devez donc vous connecter régulièrement et les enregistrer pour ne pas perdre le détail journalier des ventes.

Les commentaires

Les commentaires de l'application sont un élément capital pour comprendre ce qu'aiment vos utilisateurs, identifier des problèmes d'ergonomie ou des bogues techniques et globalement améliorer l'application.

Vous pouvez lire les commentaires depuis l'App Store ou depuis iTunes, mais vous serez limité aux commentaires postés sur votre App Store.

Il existe quelques outils pour pouvoir récupérer tous les commentaires de votre application, comme le site <http://www.moopf.com/appstorereviews/> qui récupère automatiquement tous les commentaires sur tous les App Store.

Les rapports de crash

Une des fonctionnalités les plus intéressantes apportées aux développeurs est la synthèse des rapports de crash reçus par Apple. À chaque fois que l'application est victime d'un crash, un rapport est enregistré dans l'iPhone qui est ensuite synchronisé avec iTunes. Si l'utilisateur l'accepte, ces rapports sont envoyés de façon anonyme à Apple qui les compare pour identifier les problèmes les plus fréquents.

Vous pouvez ensuite télécharger directement dans iTunes Connect un rapport décrivant les crashes les plus fréquents. À vous de chercher ensuite à les corriger, mais cette information est d'une grande valeur : elle permet d'estimer le nombre de personnes affectées par le problème, la fréquence d'apparition du bogue et de concentrer votre temps sur les problèmes qui touchent le plus de monde.

Figure 16–3
iTunes collecte automatiquement les rapports générés lors d'un crash de votre application.



Quelques conseils de lancement

Le classement de l'application est souvent le moyen le plus efficace de la promouvoir. Pour que votre application apparaisse dans le classement et que l'effet « boule de neige » l'emmène tout en haut, il faut organiser le lancement.

Plus le nombre de téléchargements sera important durant les deux premières semaines, plus vous aurez de chance de voir votre application apparaître dans les 50 ou 25 applications les plus populaires, où elle sera alors téléchargée par des gens qui ne vous connaissent pas.

Utilisez vos canaux existants pour communiquer sur l'application

Si vous avez déjà un site web avec une audience établie, vous pouvez l'utiliser pour faire connaître votre application. Une page décrivant l'application, ses fonctionnalités, quelques captures d'écran et surtout un lien de téléchargement vers l'App Store vous aidera à faire connaître l'application auprès de votre audience.

Si vous disposez d'une newsletter, vous pourrez très facilement attirer un grand nombre d'utilisateurs potentiels vers votre application.

CONSEIL Utilisez les éléments marketing Apple

Vous trouverez à l'adresse <http://developer.apple.com/iphone/marketing/>, un ensemble de ressources marketing pour aider les développeurs et en particulier le logo « Disponible sur l'App Store » que vous pouvez utiliser après avoir renvoyé un accord spécifique à Apple par fax.

Communiquez auprès des blogs et des sites spécialisés

Il existe de nombreux blogs consacrés à l'iPhone et au monde Apple. Toutes les applications ne les intéressent pas, mais ils sont souvent prêts à publier une note pour soutenir les projets les plus intéressants. Organisez une bêta-privée et contactez-les avant la publication pour leur proposer de tester votre application.

Encore plus pertinent, les sites spécialisés dans le domaine métier de l'application sont généralement intéressés par ce type d'applications qui sont perçues comme étant très innovantes et donnent une image dynamique de l'ensemble du secteur.

Utilisez le bouche-à-oreille

Le bouche-à-oreille peut aussi faire beaucoup pour une application. Si tous les collaborateurs de votre société installent l'application et en parlent autour d'eux, cela aidera fortement à augmenter le classement dans les premiers jours. Les premiers commentaires sont aussi très importants pour encourager de nouveaux téléchargements.

Vous pouvez également faire appel à vos utilisateurs pour diffuser votre application en prévoyant une fonctionnalité « Partager avec un ami » qui permet d'envoyer le lien de téléchargement de l'application par courriel.

Utilisez les réseaux sociaux

Les réseaux sociaux permettent de distribuer très largement le lien de téléchargement de votre application. Ils peuvent aussi être intégrés dans l'application avec des fonctionnalités du type « Partager sur Facebook » qui afficheront le nom de votre application et un lien de téléchargement sur les pages de vos utilisateurs.

Préparez une vidéo de démonstration

Une vidéo de démonstration de l'application peut fortement accroître l'efficacité des mesures précédentes. En effet, que ce soit pour votre site web, les blogs ou le bouche à oreille, la nouvelle sera mieux reprise si elle est accompagnée d'une vidéo courte et marquante. Vous devez jouer sur les mêmes mécaniques que le marketing viral pour assurer le succès de la vidéo et donc la visibilité de votre application.

ATTENTION La qualité de la vidéo doit être au rendez-vous !

Il n'y a rien de pire qu'une vidéo de qualité moyenne pour mettre en avant une application, car vous risquez d'en donner une image « amateur ».

Si vous ne savez pas ou n'avez pas les moyens de faire une vidéo, mieux vaut, probablement, s'abstenir.

N'oubliez pas l'auto-promotion

Si vous prévoyez de développer plusieurs applications, il est essentiel de prévoir une fonctionnalité d'auto-promotion dans la première application pour pouvoir diriger vos utilisateurs vers votre nouvelle application dès qu'elle sera disponible.

Ainsi à chaque nouvelle application bénéficiez-vous d'un réseau d'utilisateurs fidèles, qui contribueront au rayonnement de vos nouveaux produits.

Conclusion

Ce chapitre conclut ce livre consacré à la conception, à l'ergonomie, au développement et à la publication d'applications iPhone.

Développer dans l'environnement iPhone est une activité extrêmement gratifiante : quel plus grand plaisir en effet que de voir ses amis et sa famille utiliser les applications conçues et développées avec passion !

J'espère que ce livre aura su vous donner les bonnes clés pour entamer votre apprentissage, que vos applications deviendront bientôt indispensables à des milliers d'utilisateurs et, surtout, que vous prendrez autant de plaisir que nous à les créer et à les développer.

Index

- %@ 37
- @end 29
- @implementation 30
- @interface 29
- @property 47
- @selector 35
- @synchronize 36
- @synthesize 46
- A**
- AAC 202
- Ad Hoc 14
- addSubview
 - , 142
- afconvert 202
- ALAC 202
- album photo 211
- alloc 31, 40
- animation 147
- APNS 215
- App Store 22, 229
- AppID 18
- application
 - didFinishLaunching 225
 - didFinishLaunchingWithOptions 225
 - didReceiveRemoteNotification 225
- application utilitaire 91
- aps-environment 216
- assign 48
- audioPlayerDidFinishPlaying
 - successfully 204
- autorelease 44

- autoresizingMask 143
- AVAudioPlayer 203
- avertissement de mémoire 116
- AVFoundation 203
- B**
- badge 216
- barre d'onglets 95
- barre d'outils 91
- barre de navigation 93, 129, 133
- base de données 187
- bibliothèque musicale 207
- boucle de gestion des événements 54
- boucle de traitement des événements 180
- bounds 140
- C**
- cache de donnée 186
- caméra vidéo 212
- capture d'écran 235
- catégorie 232
- center 140
- centre de notification 205
- certificat de développeur 15
- certificat SSL 220
- chaîne de caractères 36
- charte graphique 79
- cible-action 123
- clavier 145
- clé de distribution 237
- commentaire 240
- comptage de références 39
- concevoir le modèle 194
- console 64
- contrat de vente 236
- contrôle parental 234
- contrôleur d'onglet 133

contrôleur de navigation 128
coordonnées 139
copyright 233
Core Animation 147
Core Audio 203
Core Data 187
couleur 144

D

Data Access Object 187
dealloc 40
délai d'attente 179
délégation de contrôle 56
délégué de l'application 54, 56
deleteObject
 199
Description de l'application 231
description du modèle 190
design 80
design pattern
 délégation de contrôle 56
 MVC 111
design pattern d'interface 85
deviceToken 217
didFailToRegisterForRemoteNotificationsWith
 Error
 218
didRegisterForRemoteNotificationsWithDevice
 Token
 217
Disponible sur l'App Store 241
documentation 12
dsym 238

E

écran de démarrage 56
édition du modèle 194
entité 190, 195
entrepôt de stockage des données 191
énumération 38
erreur réseau 238

F

feedback service 225
fenêtre 50

fichier audio 201
fichier de propriétés 166
fichier de ressource 168
fichier NIB 118
fichier plist 166
File's Owner 63
format audio 202
formats de vidéos supportés 205
Foundation 36
frame 141
framework Foundation 36

G

graphe d'objets 189
guide ergonomique 85

H

H264 205
héritage 31
hiérarchie des vues 142

I

IBAction 123
IBOutlet 61, 119
icône 55, 134, 234
id 28
identifiant d'application 18
identifiant de vue 156
IMA4 202
indicateur d'activité réseau 183
indicateur de position vertical 87
Info.plist 53, 55
initialisateur 31
insertNewObjectForEntityForName
 inManagedObjectContext
 196
Interface Builder 59, 118, 156
iPod 208
iTunes Connect 10

J

JavaScript Object Notation 169
JSON 169

L

langue principale 231

- layoutSubviews 142
 - lecture de vidéos 204
 - libération retardée 43
 - Linear PCM 202
 - lire des sons 203
 - liste d'éléments 87
 - liste groupée 150
 - liste hiérarchique 92
 - liste simple 149
 - localisation 236
- M**
- main thread 180
 - main() 52
 - mapping objet-relationnel 187
 - maquette graphique 80
 - MediaPlayer 204
 - message 27
 - message de notification 223
 - modèle 112, 190
 - MP3 202
 - MPMediaItem 209
 - MPMediaItemCollection 209
 - MPMediaPickerController 208
 - MPMediaQuery 210
 - MPMoviePlayerController 204, 205
 - MPVolumeView 204
 - mutable 38
 - MVC 111
- N**
- networkIndicatorVisible 183
 - nil 28
 - nom de l'application 55, 231
 - Nom de société 231
 - notification 215
 - NSArray 38
 - writeToFile
 - atomically 168
 - NSAutoreleasePool 44
 - NSDictionary 39
 - NSEntityDescription 196
 - NSError 178
 - NSFetchRequest 197
 - NSLog 64
 - NSManagedObjectModel 191
 - NSNotificationCenter 206
 - NSObject 26
 - performSelector
 - withObject 35
 - respondsToSelector 36
 - NSPersistentStoreCoordinator 191
 - NSSortDescriptor 198
 - NSString 36
 - spécificateur de formatage 37
 - stringWithFormat 37
 - NSURLRequest 178
 - connection
 - didFailWithError 185
 - didFinishLoading 185
 - didReceiveData 185
 - didReceiveResponse 184
 - sendSynchronousRequest
 - returningResponse error 178
 - NSURLResponse 178
 - NSUserDefaults 163
 - NSXMLParser 171, 173
 - parser
 - didEndElement 175
 - didStartElement 173
 - foundCharacters 174
- O**
- Object-Relational Mapping 187
 - objet mutable 38

objet persistant 190

openssl 221

Organizer 20

P

parentView 142

pastille 216

pathForResource ofType: 168

performSelectorInBackground 180

performSelectorOnMainThread
181

personnalisation de la barre d'onglets 134

photo 211

plist 166

pointeur sur fonction 35

pool d'autorelease 181

prédicat 198

préférences utilisateur 163

prix 236

profil de provisionnement 20

programme développeur 6

propriété 33

protocole 32

protocole informel 184

publication 23, 82

Q

QuickTime 205

R

raccourcis clavier 52

recherche d'une entité 198

référence faible 48

registerForRemoteNotificationTypes
217

Réglages 165

release 40

requête asynchrone 183

requête réseau synchrone 178

resignFirstResponder
146

retain 42, 46

versement 236

rotation d'écran 124

S

section 149

sélecteur 35

service de notification 215

setPersistentStoreCoordinator
192

signature de méthode 28

simulateur 51

SKU 232

son 201

spécificateur de formatage 37

spécifications 78

SQL 187

statistiques 240

story boarding 78

streaming 207

style de cellule 153

UITableViewCellStyleDefault 153

UITableViewCellStyleValue1 153

UITableViewCellStyleValue2 153

UITableViewCellStyleSubtitle 153

supprimer un objet 199

symbole 238

T

taille de l'écran 69

target-action 123

téléchargement 240

thread 36

thread principal 180

transform 141

transparence 144

U

UIApplication 54, 217

UIApplicationDelegate

applicationDidBecomeActive
58

applicationDidFinishLaunching
54, 57

applicationDidReceiveMemoryWarning
58

applicationWillResignActive
58

- applicationWillTerminate
57
 - UIApplicationMain 54
 - UIBarButtonItem 131
 - UIButton 145
 - UIImage 144
 - UIImagePickerController 211
 - UIImagePickerControllerDelegate 213
 - UIImageView 144
 - UILabel 143
 - UINavigationController 128
 - pushViewController
animated
132
 - UINavigationControllerItem 129
 - UIPrerenderedIcon 235
 - UITabBarController 134
 - UITabBarControllerDelegate 135
 - UITableView 150
 - dequeueReusableCellWithIdentifier 152
 - initWithFrame
style
150
 - rowHeight 154
 - UITableViewDataSource 151
 - UITableViewCell 152
 - UITableViewDelegate 154
 - tableView
 - didSelectRowAtIndexPath
154
 - heightForRowAtIndexPath
154
 - UITableViewSource
 - numberOfSectionsInTableView
152
 - tableView
 - cellForRowAtIndexPath
152
 - numberOfRowsInSection
151
 - titleForFooterInSection
153
 - titleForHeaderInSection
153
 - UITextField 145
 - UITextFieldDelegate 145
 - UIView
 - frame 69
 - UIViewController 125
 - didReceiveMemoryWarning 116
 - dismissModalViewControllerAnimated
137
 - hidesBottomBarWhenPushed 132
 - loadView 115
 - modalTransitionStyle 138
 - navigationItem 129
 - parentViewController 137
 - presentModalViewController
animated
137
 - shouldAutorotateToInterfaceOrientation
124
 - viewDidAppear
115, 122
 - viewDidDisappear
115, 122
 - viewDidLoad 115, 120
 - viewDidUnload 121
116
 - viewWillAppear
115, 122
 - viewWillDisappear
115, 122
 - UIWebView 146
 - UIWindow 50
- V**
- validation 239
 - variable d'instance 30
 - version 233
 - vidéo 212
 - vidéo en streaming 207
 - viewWithTag
156
 - volume de l'iPhone 204
 - volume de lecture 204
 - volume système 204
 - vue modale 137

W

WebKit 146

X

Xcode 49

XML 171

Programmation iPhone OS 3



T. Sarlandie

Thomas Sarlandie est co-fondateur et directeur technique de Backelite, l'agence mobile leader en ergonomie et développement d'applications iPhone. Il a été l'un des pionniers du développement iPhone en France et a réalisé avec son équipe des dizaines d'applications iPhone dont *lemonde.fr*, pour un total de plus de 2 millions de téléchargements en un an. Il a souhaité partager sa passion et cette expérience unique en France à travers ce livre.

La réussite d'une application iPhone repose sur sa conception et sa réalisation : elle exige un savoir-faire en ergonomie mobile et la maîtrise de l'ensemble des contraintes spécifiques à la plate-forme.

La référence du développeur iPhone professionnel : de la conception à la publication sur l'App Store

De la conception de l'application – encadrée par de strictes règles d'ergonomie – jusqu'à son déploiement, cet ouvrage détaille les bonnes pratiques garantissant la qualité de vos développements iPhone : gestion de projet et architecture MVC, ergonomie mobile et design patterns d'interface. Les fondamentaux du développement iPhone sont détaillés, de l'Objective-C et sa gestion spécifique de la mémoire aux contrôleurs de vue, en passant par la mise en place des vues et des TableView.

Écrit par le directeur technique de l'une des premières agences spécialisées dans le développement sur plate-forme mobile et iPhone, l'ouvrage traite en profondeur d'aspects avancés tels que l'accès aux services web (JSON, XML), la gestion de flux audio et vidéo, la persistance avec le framework CoreData et l'utilisation du service de notifications Apple. Enfin, il fournit de précieux conseils pour publier sur l'App Store et y gagner en notoriété.

Couvre les nouveautés de la version 3 de l'iPhone OS.

Au sommaire

Développer pour iPhone • Pré-requis matériels • Inscription et sites développeur • iTunes Connect • **Le SDK iPhone et la documentation Apple** • Diffusion en test (mode Ad Hoc) • Certificat • **Objective-C** • Classes, variables d'instance et méthodes • Héritage • Protocoles et propriétés • La notation point • Les threads • **Bibliothèque standard : le framework Foundation** • Chaînes de caractères • Listes • Dictionnaires • Comptage de références • Gestion de la mémoire • **Xcode** • Fichiers créés par Xcode • Délégué de l'application • Fin de lancement • Clôture • **Interface Builder** • Créer vues et labels • Paramètres de signature • Compiler et lancer • **Méthodologie de développement** • Étapes du projet • Tri des fonctionnalités clés • Story-boarding et spécifications • Tests et optimisation • **Ergonomie iPhone et design patterns d'interface** • Les métaphores • Temps et fréquence d'utilisation • Degré de concentration • Manipuler des données • Listes d'éléments et listes groupées • Sections et index • **Contrôler les écrans** • Le modèle MVC • Données, vue et contrôleur • Cycle de vie d'un contrôleur • Instanciation avec et sans fichier NIB • Gestion d'événements • Rotations d'écran • Navigation et onglets • Mode modal • **Développer et animer les vues** • Frame • Hiérarchie, positionnement et redimensionnement • Vues de UIKit • Labels • Boutons • Zones de texte • Afficher des contenus web • **Listes d'éléments** • TableView • Cellules personnalisées • **Lire et enregistrer des données** • Préférences utilisateur • Fichiers de propriétés plist • Format de données JSON • Manipuler des données XML • **Communiquer avec l'extérieur : appels réseau synchrones et asynchrones** • Authentification et redirections • **Traitements en arrière-plan** • Multithreading • Persistance d'objets avec CoreData • L'ORM : de l'objet au relationnel • Manipulation d'objets gérés par le contexte • Recherche dans la base • Suppression • **Données multimédias** • Son • Formats audio et conversion • Formats vidéo pris en charge • Vidéos live • Accès aux photos et vidéos utilisateur • **API de notifications (APNS)** • Inscription au service • Jeton • Envoi et réception des notifications • Certificat SSL • Désinscription et erreurs • **Publication sur l'App Store** • Éléments marketing • Date de disponibilité et prix • Localisation • Vérifications techniques • Éléments graphiques Apple • Statistiques de téléchargement • Les commentaires • Communication et promotion.

À qui s'adresse cet ouvrage ?

- Aux professionnels de la conception web et mobile qui souhaitent être présents sur le marché des services portés sur iPhone ;
- À tous les particuliers & fans d'iPhone qui souhaitent concevoir, publier ou vendre une application sur l'App Store.