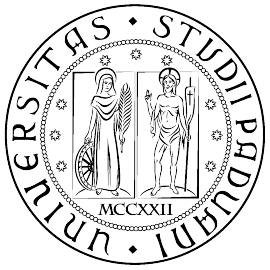

Università degli Studi di Padova

FACOLTÀ DI INGEGNERIA
Corso di Laurea Triennale in Ingegneria Informatica

Tesi di Laurea Triennale

Realizzazione e Sviluppo di Tools per la Gestione Integrata delle Funzionalità su iPhone

Candidato:
Orietti Luca
Matricola 578872

Relatore:
Ing. Federico Filira

Anno Accademico 2012–2013

Indice

Capitolo 1

Obiettivo della Tesi

Questa tesi si pone l'obiettivo di analizzare le fasi necessarie alla realizzazione di una applicazione per Smart Phone, nel nostro caso per iPhone, andando ad esaminare anche i passi preliminari che sono necessari per sviluppare la nostra applicazione.
Inizieremo quindi spiegando come acquisire tutti gli strumenti necessari per iniziare a lavorare su ambiente Mac, dalla creazione dell'account Developer nel sito della Apple alla descrizione degli strumenti, ambiente di sviluppo, simulatori, etc, che dovremo utilizzare per riuscire a realizzare le applicazioni su questo ambiente.
Successivamente andremo a realizzare una prima applicazione, molto semplice, composta da due pagine collegate fra di loro e che utilizza uno dei due metodi di salvataggio dati che verranno analizzati successivamente nella trattazione della tesi. Una volta visti i primi rudimenti per realizzare una semplice applicazione passeremo ad analizzare le varie funzioni che serviranno alle nostre applicazioni andando a vedere anche a livello di codice come devono essere realizzate.
A questo punto una volta in possesso di tutto il necessario andremo a realizzare le applicazione vere e proprie di cui, una prima che permetterà di leggere e salvare al proprio interno, tramite l'uso di un database, codici a barre e QR mentre una seconda permetterà di effettuare il pagamento di biglietti o abbonamenti di un'azienda di trasporto pubblico tramite l'invio di un semplice SMS.
Infine andremo a prendere in considerazione quelli che potrebbero essere possibili sviluppi da apportare nel futuro in modo da rendere le nostre applicazioni più complete e quindi maggiormente spendibili in ambito commerciale.

Capitolo 2

Sviluppo di Applicazioni in Ambiente Mac

Prima di iniziare a sviluppare la nostra applicazione per iPhone, bisogna prima dotarsi degli strumenti necessari affinché ciò sia possibile.
Come prima cosa quindi dobbiamo creare un account come sviluppatori al sito dell'Apple (Mac Dev Center), qui ci verrà richiesto di fornire alcuni dei nostri dati come il nostro nome, cognome e residenza e un indirizzo e-mail valido che servirà per attivare il nostro account. Oltre a questo ci viene anche chiesto su quali piattaforme abbiamo intenzione di sviluppare le nostre applicazioni (Mac OS oppure iOS) e che tipo di software svilupperemo.

Una volta finita la registrazione e attivato il nostro account ci sarà possibile scaricare l'ambiente di sviluppo Xcode, se non lo abbiamo già disponibile dal momento dell'acquisto del sistema operativo, che nel pacchetto di installazione comprende anche tutte le SDK [1] necessarie a sviluppare i nostri programmi futuri.
Oltre a tutto queste cose che possiamo definire "materiali", per la realizzazione di applicazioni o programmi in ambiente Mac o iOS, abbiamo bisogno di avere delle basi sul linguaggio di programmazione Objective-C.

2.1 Developer Account

Il primo passo, come già accennato prima, per la realizzazione dei nostri applicativi è quello di creare un account presso il sito della Apple riservato agli sviluppatori. Bisogna quindi recarsi presso l'indirizzo *developer.apple.com*, nel nostro caso entrare nella sezione dedicata agli sviluppatori per iOS quindi cliccando sul link *iOS Dev Center*. A questo punto ci viene data la possibilità di loggarci al sito, se non lo abbiamo già fatto, o di registrarci. Visto

[1] Software Development Kit o pacchetto per lo sviluppo delle applicazioni in italiano

che è la mia prima volta in questo sito, inizio con il registrarmi, quindi clicchiamo sull'apposito link e vediamo che informazioni ci vengono richieste.
Si apre subito una schermata dove possiamo facilmente notare che la registrazione si compone fondamentalmente di cinque passi:

- il primo passo riguarda la scelta dell'Apple ID, come dice il nome sembra essere un identificativo della Apple che serve ad identificare in maniera univoca gli utenti, infatti ci si presentano due opzioni, la prima è quella di creare un ID da zero mentre la seconda ci consente di utilizzare un Apple ID già creato in un altro sito della famiglia Apple come iTunes, Apple Online Store, etc. Scegliamo di creare un nuovo account e proseguiamo;

- al secondo passo si inizia con l'inserimento dei dati personali, la prima cosa che ci viene chiesto è un indirizzo e-mail valido che fungerà sia da indirizzo mail vero e proprio sia da login per accedere successivamente al sito, insieme a questo ci viene chiesto di fornire anche una password.
  nel secondo blocco di informazioni che ci vengono richieste ci viene chiesto di impostare la nostra data di nascita e una domanda con relativa risposta nel caso in cui perdessimo la nostra password e volessimo recuperarla.
  l'ultimo blocco sono le informazioni personali vere e proprie, qui ci viene richiesto il nostro nome, in che stato viviamo e il nostro indirizzo e il nostro numero di telefono.
  Tutti i campi sono considerati obbligatori e vanno quindi compilati;

- al terzo passo si tratta semplicemente di decidere quali programmi andremmo a sviluppare e su che piattaforma. La mia scelta è ovviamente ricaduta sulla piattaforma iOS e sullo sviluppo di programmi di Utility. Sinceramente non so se la scelta di opzioni diverse comporti un trattamento effettivamente diverso o se queste informazioni vengano semplicemente usate per una ricerca più veloce di certi utenti o sviluppatori da parte della società che poi andrà ad utilizzare le applicazioni prodotte dai primi;

- al quarto step, ci viene semplicemente chiesto di accettare alcune licenze legali come quella della privacy, accettiamo tutto ed andiamo avanti;

- a questo punto siamo arrivati alla fine; come ultima cosa da fare non ci rimane che attendere la mail di conferma di registrazione avvenuta ed inserire il codice che si trova all'interno della mail nell'ultima pagina di registrazione in modo da attivare in modo permanente il nostro account.

Una volta arrivati alla fine di questa procedura siamo pronti per entrare come sviluppatori nel sito e poter scaricare la versione gratuita dell'ambiente di sviluppo Xcode per poter iniziare a creare le nostre applicazioni.

2.2 Xcode

Xcode è un ambiente di sviluppo integrato (Integrated Development Environment, IDE) sviluppato da Apple per agevolare lo sviluppo di software per Mac OS X e iOS. È fornito gratuitamente in bundle con il sistema operativo a partire da Mac OS X 10.3 Panther, sebbene sia in grado di generare programmi per qualsiasi versione di Mac OS X. Estende e rimpiazza il precedente tool di sviluppo della Apple, Project Builder, che era stato ereditato dalla NeXT [2]. Ufficialmente Xcode non funziona su Mac OS X 10.2 Jaguar.

Xcode lavora in congiunzione con Interface Builder (proveniente da NeXT), un tool grafico per realizzare interfacce grafiche.
Xcode include GCC [3], che è in grado di compilare codice C, C++, Objective-C/C++ e Java. Supporta ovviamente i framework Cocoa [4] e Carbon [5], oltre ad altri.
Una delle caratteristiche tecnologicamente più avanzate di Xcode è che supporta la distribuzione in rete del lavoro di compilazione. Usando Bonjour [6] e Xgrid [7] è in grado di compilare un progetto su più computer riducendo i tempi. Supporta la compilazione incrementale, è in grado di compilare il codice mentre viene scritto, in modo da ridurre il tempo di compilazione.

Passando all'analisi del suo utilizzo, Xcode si apre fornendoci subito una schermata nella quale vengono visualizzati subito sia i progetti che abbiamo già creato e sui quali magari abbiamo già lavorato, accessibili semplicemente con un doppio click del mouse, vieni inoltre visualizzato un intuitivo menù dal quale possiamo decidere se iniziare un nuovo progetto, consultare una guida sull'utilizzo dell'ambiente di sviluppo o visitare l'Apple Development

[2] La NeXT Computer è una società fondata nel 1985 da Steve Jobs. L'obbiettivo della società era di creare una rivoluzione nel campo dell'informatica, cercando di ripetere il successo avuto con Apple

[3] GNU Compiler Collection, è un compilatore multi-target. Nato inizialmente come compilatore C, dispone di vari front-end per compilare anche linguaggi come Java, C++, Objective C, etc...

[4] ambiente di programmazione orientato agli oggetti sviluppato da Apple

[5] API contenute nel sistema operativo Mac OS X

[6] Bonjour è una tecnologia generica nata per individuare automaticamente la presenza di servizi nelle LAN. Questa tecnologia viene usata in modo massiccio da Mac OS X per consentire agli utenti di installare una rete senza bisogno di configurarla

[7] tecnologia sviluppata da Apple per i sistemi Mac OS X. Questa tecnologia è consentire lo sviluppo di applicazioni parallele che distribuiscono le operazioni da eseguire su più macchine collegate da una rete riducendo i tempi necessari all'elaborazione

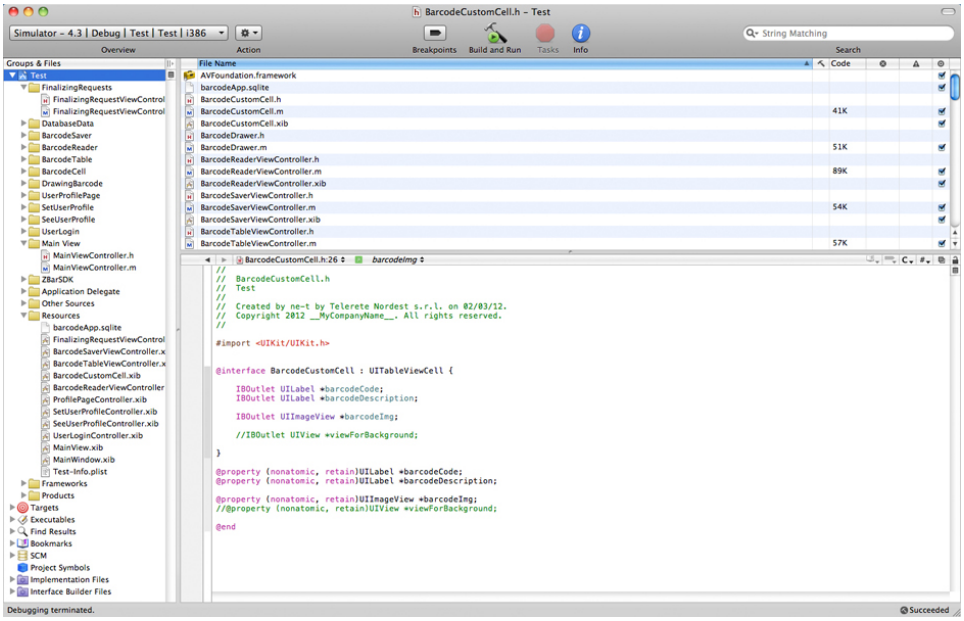

Figura 2.1: Ambiente di sviluppo Xcode

Center aprendo l'indirizzo automaticamente nel nostro browser predefinito. Una volta che si è selezionato o un nuovo progetto od una già esistente ci verrà aperto quello che è l'ambiente di sviluppo vero e proprio (Figura 2.1), questo è diviso in varie schermate che verrano analizzate in maniera approfondita nei capitoli successivi come anche verranno descritte successivamente molte delle sue funzioni, che verranno trattate mano a mano che ne avremo bisogno.
Uno strumento che ho trovato molto utile e del quale non si farà alcun riferimento successivamente è la console dove vengono visualizzati i vari messaggi di errore che possono occorrere durante l'esecuzione delle nostre applicazioni, cosa estremamente utile soprattuto se abbiamo apportato molte modifiche ad un programma e questo ad un certo momento crasha in maniera inaspettata. Per rendere visibile la console basta andare nel menù in alto sotto la voce *Run* e quindi *Console*.

2.2.1 Interface Builder (IB)

Interface Builder è un'applicazione per lo sviluppo software per il sistema operativo Mac OS X e fa parte di Xcode (tool di sviluppo fornito gratuitamente con Mac OS X, precedentemente chiamato Project Builder). Interface Builder permette agli sviluppatori che usano Carbon e Cocoa di disegnare interfacce grafiche per le applicazioni usando uno strumento grafico, senza la necessità di scrivere decine di righe di codice. L'interfaccia risultante è salvata in un file .nib (abbreviazione di NeXT Interface Builder).

Interface Builder deriva dal omonimo software del NeXTSTEP [8] development software. Il suo aspetto è immutato in Gorm, l'equivalente di Interface Builder per GNUstep [9]. Una versione di Interface Builder è usata anche nello sviluppo del software OpenStep [10].

Il programma una volta lanciato si presenta con quattro finestre flottanti almeno per quanto riguarda la versione presente all'interno dell'ambiente di sviluppo Xcode nella versione 3.2.4, mentre nell'ultima versione di Xcode, la 4.3.2, Interface Builder è completamente incorporato nell'IDE. Le quattro finestre che vengono aperte svolgono funzioni dedicate:

- la prima finestra che andiamo a descrivere compare all'apertura normalmente alla sinistra dello schermo, questa raccoglie in una comoda interfaccia tutti gli oggetti che possono essere inseriti, semplicemente trascinandoli all'interno dell'interfaccia grafica dell'applicazione;
- un'altra finestra è quella che visualizza l'aspetto che avrà la nostra interfaccia grafica, da qui potremo spostare e modificare i vari oggetti semplicemente con l'uso del mouse;
- una terza visualizza in maniera compatta tutti gli oggetti che stiamo utilizzando nella nostra interfaccia e da qui possiamo in maniera abbastanza intuitiva creare i collegamenti tra gli oggetti dell'interfaccia e gli oggetti o funzioni definite all'interno del codice sorgente;
- l'ultima schermata ma non meno importante delle altre, serve per gestire tutte le informazioni e proprietà di un singolo oggetto, da qui potremo assegnare la classe di riferimento per quel determinato oggetto, modificarne le dimensioni, assegnare i collegamenti con il codice sorgente e modificare anche l'aspetto grafico dell'oggetto.

Da questa veloce carrellata delle varie funzioni rese disponibili da Interface Builder possiamo vedere come sia semplice anche per un utente poco esperto riuscire a creare un'interfaccia anche se semplice, elegante nell'aspetto.

2.2.2 Simulatore iOS

L'ambiente di sviluppo Mac oltre a mettere a disposizione una comoda interfaccia grafica sia per scrivere il codice sorgente sia per la creazione di interfacce grafiche tramite Interface Builder, mette a disposizione dell'utente anche un simulatore per testare la maggior parte delle applicazioni ed è

[8] NeXTSTEP è un sistema operativo orientato agli oggetti, multitasking che venne sviluppato dalla NeXT di Steve Jobs, tra il 1986 e il 1995

[9] GNUstep è un'implementazione delle librerie OpenStep in Objective C di NeXT (attualmente Apple)

[10] API orientata agli oggetti volta a portare NeXTSTEP anche su altre piattaforme hardware

presente sia la versione per iPhone che per iPad.

Figura 2.2: Simulatore iPhone

Proprio per il fatto che è un simulatore e non un dispositivo reale, non può quindi disporre di tutte le funzionalità che la sua controparte invece possiede, per citare alcuni servizi che non sono disponibili sul simulatore si può ad esempio prendere in considerazione il servizio di composizione (non invio, anche questo comunque non supportato) di messaggi SMS e l'utilizzo della fotocamera.
Tutte le altre funzioni invece, facendo magari ricorso a vari stratagemmi, riescono comunque ad essere eseguite consentendo così allo sviluppatore di poter testare in tutta tranquillità ed economicità il prodotto che sta andando a sviluppare.

Capitolo 3

Pubblicazione delle Applicazioni nell'AppStore

3.1 Acquisto della Licenza Developer

Dopo aver scaricato l'ambiente di sviluppo e realizzato le nostre applicazioni, per rendere queste disponibili anche ad altri utenti iPhone dovremo come prima cosa acquistare la licenza *iPhone Developer* che ha costo di 99$ (circa 79€).

Vediamo quindi in dettaglio i passi da effettuare. Come prima cosa accediamo alla homepage *Apple Developer* ed entriamo nella sezione *iPhone Dev Center*. Sulla destra nella sezione iPhone Developer Program accediamo alla sezione *Join the iPhone Developer Program* tramite il link "Learn More", clicchiamo quindi sul tasto *Enroll Now* e nella pagina successiva nuovamente *Enroll Now*. Possiamo notare che ci sono due tipi di licenza che si possono acquistare: Standard o Enterprise. La seconda è ovviamente rivolta a grandi software house che producono molte applicazioni e che hanno quindi anche molti dipendenti, ovviamente per il nostro progetto selezioneremo la Standard. Procediamo quindi fino a quando non arriveremo alla prima schermata della registrazione vera e propria dove ci verrà chiesto se creare un nuovo *Apple ID* o se utilizzarne uno già esistente. Qui la scelta e personale, io avendo creato il mio primo Apple ID quando mi sono registrato come sviluppatore per scaricare l'ambiente di sviluppo ho deciso di riutilizzare quello.

Avanzando nella registrazione ci verrà proposto di ripondere ad un breve questionario molto simile a quello che abbiamo compilato per registrare il nostro account come sviluppatore.
Accettiamo la licenza, inseriamo il codice ricevuto via e-mail, procediamo controllando le info che abbiamo inserito fino ad ora in modo che siano tutte corrette e procediamo con l'acquisto su *Add to Cart*. Da questo momento

accediamo all'Apple Store dove possiamo proseguire con il pagamento e l'acquisto.
L'attivazione della licenza dovrebbe avvenire circa in 24 ore e una volta che ci sarà arrivata la mail con il codice di attivazione andiamo nella home page iPhone Developer dove verrà visualizzata una nuova voce chiamata *Active Developer Program*. Entriamo in questa pagina ed inseriamo il codice di attivazione e finalmente saremo diventati degli sviluppatore Apple a tutti gli effetti.

3.2 Creazione dei Certificati

Altro passo importante dopo l'acquisto della licenza è la creazione dei certificati.
Grazie a questi infatti potremo montare le nostre applicazioni sul nostro iPhone, condividele con alcuni amici ed inviarla ad Apple per la validazione e quindi la distribuzione al pubblico.
Partiamo quindi subito a vedere quali sono i certificati che dovremo creare. Questa operazione sebbene lunga comporta dei passaggi relativamente semplici.

Cominciamo con il primo certificato, *Certificate Signing Request* (CSR). Dal nostro MAC apriamo nella cartella Applicazioni → Utility l'applicazione Accesso Portachiavi. Dal menù che comparirà successivamente selezioniamo Assistente Certificato → Richiedi un Certificato ad un'Autorità di Certificazione, si aprirà quindi una finestra nella quale ci verrà chiesto di inserire alcune nostre informazioni e la selezione di alcune opzioni.
Inseriamo quindi la nostra e-mail, nome e cognome e selezioniamo l'opzione "Registrato su Disco" e spuntiomo l'ultima opzione per le informazioni di copia di chiave e clicchiamo quindi su Continua.
Nella pagina successiva selezioniamo come dimensione di chiave almeno 2048 bits e come algoritmo l'RSA [1]. La nostra chiave verrà quindi creata sulla scrivania.

Accediamo quindi alla pagina iPhone Developer Program ed inseriamo il nostro UDID [2] nel menù Devices.
Per trovare il nostro UDID apriamo Xcode con il nostro dispositivo collegato al nostro MAC e quindi dal menù di sinistra, dove per intenderci, sono presenti i file del nostro progetto, aprimo la tendina Organizer e selezioniamo il nostro dispositivo, in questo modo potremo vedere il nostro UDID come mostrato in Figura 3.1.

[1] Indica un algoritmo di crittografia asimmetrica, utilizzabile per cifrare o firmare informazioni
[2] Codice di 40 cifre che identifica in modo univoco il nostro dispositivo

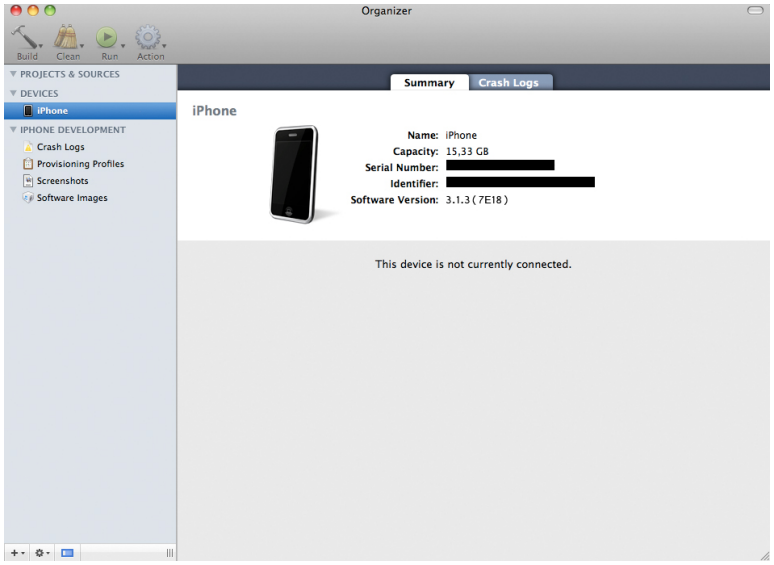

Figura 3.1: UDID iPhone

Copiamo quindi questa stringa e ritorniamo alla pagina iPhone Developer, clicchiamo quindi su Add Devices, inseriamo un nome identificativo (in modo da riconoscere il dispositivo) e incolliamo quindi la stringa che abbiamo precedentemente copiato. Possiamo quindi ora aggiungere il nostro dispositivo.

Andiamo ora a creare i certificati "Developer" e "Distribution". Nel menù Certificates che si trova nella home page iPhone Developer accediamo alla pagina Developement e clicchiamo su Add Certificate.
Nella nuova pagina che si aprirà facciamo l'upload del file CSR precedentemente creato e clicchiamo su *Submit*. Procediamo con lo stesso procedimente anche per l'altro certificato.
Ritornando ora alla voce Development sarà quindi possibile effettuare il download del certificato appena creato grazie al CSR precedente. Salviamoci quindi in qualche luogo entrambi i certificati che abbiamo appena creato e installiamoli subito facendo doppio click su di essi, in questo modo dovrebbero apparire nella lista dei certificati dell'Accesso Portachiavi.

Creiamo ora un AppID accedendo quindi al menù *App IDs* sulla home della nostra iPhone Developer Program. Questo App ID come dice anche il nome servirà a identificare in modo univoco la nostra applicazione. Clicchiamo quindi su *Add New App ID* ed accediamo alla pagina di creazione del nostro AppID (Figura 3.2) e compiliamo i campi richiesti e clicchiamo infine su *Submit*.

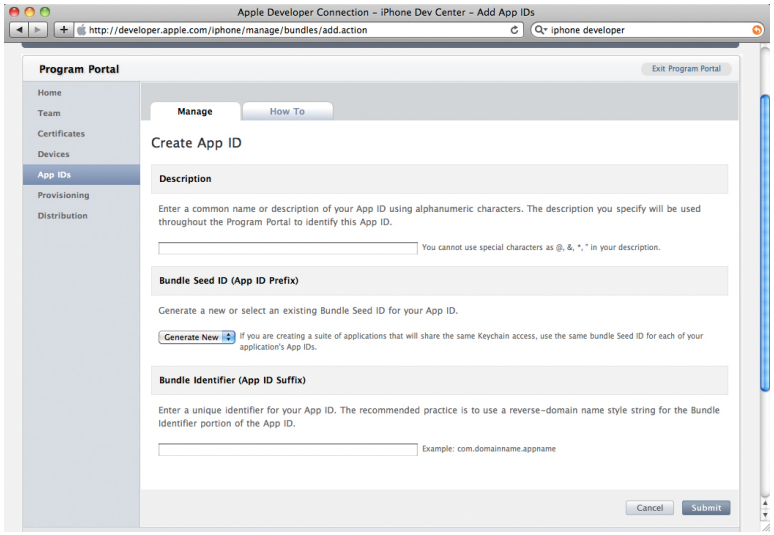

Figura 3.2: Creazione App ID

Siamo quindi arrivati alla creazione del Provisioning Profile dove andremo ad utilizzare tutti i certificati che abbiamo creato fino a questo momento e che ci consentirà di distribuire il nostro prodotto, quindi di fatto di caricare l'applicazione sul nostro cellulare e di inviarla alla Apple. Sempre dalla home page iPhone Developer accediamo alla voce Provisioning → New Profile. A questo punto ci si presenterà un form di compilazione per un nuovo profilo, diamogli quindi un nome, selezioniamo il nostro certificato creato in precedenza e nella lista AppID selezioniamo quella che abbiamo appena creato e nei Devices selezioniamo il nostro dispositivo. Clicchiamo quindi su *Submit*. A questo punto ci verrà data la possibilità di scaricare il nuovo certificato appena creato.
Andiamo ora in Provisioning → Distribution → New Profile, per creare un altro certificato, questa volta relativo al tipo di distribuzione che sceglieremo.
In Distribution Method troveremo due voci: *Ad Hoc* ed *App Store*. La prima delle due scelte ci permetterà di distribuire la nostra applicazione solo a i dispositivi che andremo a specificare nella voce Devices, mentre la voce App Store ci consentirà di impacchettare la nostra App per inviarla ad Apple.
Nel form quindi inseriamo un nome al nostro profilo e selezioniamo la nostra App ID. Clicchiamo quindi su submit ed effettiamo il download come in precedenza. A questo punto avremo altri due certificati che saranno quelli che ci serviranno nei passi sucessivi.

Selezioniamo quindi questi certificati e trasciniamoli semplicemente sulla nostra icona di Xcode in questo modo verranno automaticamente installati all'interno dell'ambiente di programmazione. Per verificare che l'installazio-

ne sia andata a buon fine andiamo su Xcode → Window → Organizer a questo punto sulla sinistra Provisioning Profiles e nella tabella in alto dovremmo vedere installati i nostri profili.

Una volta completata questa ultima operazione passiamo al nostro progetto in Xcode. Apriamo quindi il nostro progetto e clicchiamo con il tasto desto sul nome del progetto che racchiude tutti i file sorgenti nel menù a sinistra, per capirci il nome a destra dell'icona blu che dovrebbe essere di fatto il nome del progetto. Selezioniamo quindi *Get Info*.
Alla voce *Configuration* duplichiamo la voce Release e rinominiamola in Distribution. Andiamo quindi nel menù *Build* e in Configuration selezioniamo la voce Distribution e nella lista in basso, alla voce Code Signing Identity → Any iPhone OS Device selezioniamo il Profilo importato in precedenza e possiamo quindi chiudere la finestra.

Dal nostro progetto in Groups & Files andiamo in *Targets* e selezioniamo l'unica voce presente e con il tasto destro selezioniamo come prima la voce Get Info.
Si aprirà una finestra molto simile a quella precedente ed effettuiamo la stesse operazioni viste sopra ma non chiudiamo la schermata. Andiamo invece nella sezione *Properties* ed in Identifiers inseriamo l'App ID creato. Possiamo ora chiudera la finistra e nel menù *Overview* del nostro progetto selezioniamo la voce *Distribution* e procediamo al Build del nostro progetto. Se tutto è andata a buon fine e senza errori nella sezione Group & Files dovrebbe esserci il file con estensione *.app* del nostro progetto. Clicchiamo quindi con il tasto desto su di esso, selezioniamo *Reveal in Finder* e otterremo il file che dovrà essere poi confezionato ed inviato ad Apple.

3.3 Preparare ed Inviare le Applicazioni in AppStore

Una volta preparati tutti i certificati necessari e compilate le nostre applicazionioni in modalità "Distribution" siamo pronti per come prepararle e inviarle ad Apple per la pubblicazione sull'AppStore.

Sempre dal sito Developer di Apple, facciamo il login ed entriamo nella sezione *iTunes Connect*. A questo punto, in basso, clicchiamo su "Manage Your Application" quindi clicchiamo in alto a sinistra sul pulsante blu "Add New Application".
Si aprirà ovviamente una nuova schermata nella quale dovremo inserire le informazioni necessarie per la pubblicazione della nostra applicazione, le informazioni da inserire sono:

- **App Name**: questo è il titolo della nostra applicazione, quello che apparirà nell'AppStore;
- **SKU Number**: è il numero identificativo univoco, possiamo crearlo casualmente o a nostro piacimento;
- **Bundle ID**: dalla tendina dovremo scegliere l'app ID relativo al progetto che stiamo per sottoporre.

A questo punto clicchiamo su Continua e accederemo ad un'altra pagina, in questa ci verrà chiesto di settare alcuni parametri della nostra applicazione:

- **Availability Date**: la data della presunta pubblicazione della nostra applicazione;
- **Price Tire**: il prezzo a cui vogliamo vendere la nostro applicazione;
- **Discount for Educational Institution**: sconti per scuole, etc...

Infine, in basso possiamo selezionare le nazioni in cui vogliamo (o non vogliamo) che la nostra applicazione sia venduta e quindi in quali store dovrà essere presente la nostra app.
Continuando nella procedura di creazione troveremo un'altra pagina nella quale dovremo inserire altre informazioni come la versione dell'applicazione, una descrizione della stessa, alcuni siti di riferimento se presenti, etc... Più in basso invece troviamo un elenco di contenuti che dovremo specificare se sono presenti o meno nella nostra app, come ad esempio se ci sono riferimenti a bevande alcoliche, alla violenza oppure alle scommesse e così via.
Continuando a scendere nella pagina troviamo i form per l'upload dell'icona della nostra applicazione che verrà usata nell'AppStore e la possibilità di caricare Screenshot che diano l'idea di come si presenti l'applicazione.

Una volta concluso l'insrimento di questi ultimi dati la procedura ci riporterà sulla pagina principale delle nostre appliczioni, clicchiamo su quella appena creata o una di nostro interesse e, nella schermata che ci verrà aperta sotta all'icona clicchiamo su "View Details".
Nella nuova pagina, in alto a destra clicchiamo sul bottone "Ready to Upload Binary", si aprirà quindi una schermata in cui ci verrà proposto di scaricare l'App Loader, un programma per MAC che permette di inviare i nostri progetti ad Apple. Scarichiamolo, se non lo abbiamo già fatto, e quindi clicchiamo su Continua.
A questo punto andiamo su Xcode e apriamo il nostro progetto. Sulla sinistra apriamo *Products*, selezioniamo il nostro progetto con estensione *.app*, quindi tasto destro e *Reveal in Finder*.
Si aprirà quindi una schermata nel Finder con all'interno il file di progetto con l'estensione .app, comprimiamolo in formato zip accertandoci che non

siano presenti spazi nel nome del file, altrimenti l'App Loader non lo accetterà. Creato il file aprimo l'App Loader e dal menù a tendina dovremo essere in grado di selezionare il nome dell'app che abbiamo inserito nell'iTunes Connect; selezioniamola e clicchiamo su *Next*, andiamo quindi a cercare il file zip che abbiamo appena creato e infine clicchiamo su *Send*.
Se tutto va bene, dovrebbe iniziare il caricamento dell'applicazione ed appena finito non ci resterà che aspettare pazientemente che l'app venga approvata da Apple ed in qualche giorno questa dovrebbe essere online e pronta per il download.

Capitolo 4

PhoneGap

PhoneGap è una libreria che permette di scrivere codice per applicazioni per cellulari esulando di fatto il programmatore dall'utilizzare il linguaggio di programmazione della macchina per la quale sta scrivendo il suo programma. Il codice cha andremo a realizzare quindi utilizzerà come linguaggio Javascript e quindi le pagine della nostra applicazione saranno di fatto delle vere e proprie pagine web.
Questo però permette di poter scrivere un'unica applicazione in javascript /html e poi poterla eseguire, dopo essere stata opportunamente compilata, in molte delle più importanti piattaforme per cellulare attualmente in commercio.
La possibilità di essere così portabile su vari sistemi operativi rende anche questo strumento particolarmente impreciso e non sempre sfrutta a pieno tutte le capacità del codice nativo per quella determinata macchina. Per questo motivo ed anche per il fatto che lavorare su pagine web rende la realizzazione dell'interfaccia grafica molto scarna e difficile da personalizzare in maniera semplice e veloce, si è deciso, dopo poche ore di lavoro con questa libreria, di abbandonare questa strada optando per l'utilizzo del linguaggio nativo (Objectiv-C) sebbene più complesso e completamente nuovo per il sottoscritto.

Capitolo 5

Metodi di Salvataggio delle Informazioni

Un passaggio importante nella realizzazione delle nostre applicazioni sarà quello di riuscire a mantenere un certo numero di informazioni disponibili per un prolungato periodo di tempo, questo sarà possibile avvalendosi di due metodi, uno molto più semplice da utilizzare ma con poche possibilità di gestire grandi quantità di dati in maniera ordinata e intuitiva mentre l'altro ovviamente più complesso dal punto di vista della realizzazione ma che una volta implementato semplifica di molto il lavoro che bisognerà svolgere successivamente per recuperare anche grandi quantità di informazioni.
Andiamo ora ad analizzare più nel dettaglio questi due strumenti.

5.1 NSUserDefaults

Attraverso l'utilizzo di questa classe sarà possibile salvare piccole quantità di informazione all'interno del cellulare in maniera permanente, questo metodo di salvataggio viene solitamente utilizzato per immagazzinare le impostazioni di default della nostra applicazione anche se poi di fatto può essere utilizzato anche per altri scopi.

Il suo utilizzo è molto semplice e richiede semplicemente di istanziare un oggetto di tipo NSUserDefault nel seguente modo:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
```

A questo punto possiamo decidere di inserire delle informazioni, ognuna di queste viene gestita secondo la modalità *Chiave-Valore* e a seconda del tipo di valore da inserire cambia il tipo di funzione da utilizzare, ad esempio se vogliamo registrare un *intero* useremo la funzione *setInteger*, per i reali a doppia precisione useremo *setDouble* e per un qualsiasi oggetto sia esso una

stringa o un oggetto definito dall'utente usiamo *setObject*.
Vediamo quindi un esempio di inserimento di un oggetto:

```
[userDefaults setObject:myObject forKey:@‘‘myObjectLabel’’];
```

Quindi *myObject* è fisicamente l'oggetto che verrà salvato mentre la stringa *myObjectLabel* è il nome con il quale andremo successivamente a recuperarlo.
La funzione di recupero dati è quindi:

```
myNewObject = [userDefaults objectForKey:@‘‘myObjectLabel’’];
```

Analizziamo quest'ultima riga, *myNewObject* è l'oggetto dello stesso tipo di *myObject* che conterrà la variabile di ritorno della funzione *objectForKey*, come prima se avessimo avuto a che fare con interi o reali a doppia precisione avremmo dovuto usare le funzioni *integerForKey* e *doubleForKey*.

Per le nostre applicazioni useremo questo tipo di gestione delle informazioni in maniera consistente soprattutto nel momento in cui avremo bisogno di passare alcune informazioni tra le varie pagine che verrano aperte durante l'esecuzione del programma.

5.2 SQLite

Ad un certo punto nella progettazione della nostra applicazione, in particolare quella che gestisce il salvataggio dei codici a barre, si è visto che sarebbe stato utile dare la possibilità all'utente di poter salvare i codici che quest'ultimo ha scannerizzato. A questo punto salvare le informazioni attraverso l'utilizzo della classe UserDefault sarebbe stato totalmente inefficiente, si è quindi deciso di salvare queste informazioni in maniera più ordinata ed efficiente all'interno di un database.

La scelta della base di dati da utilizzare è caduta su SQLite, questa libreria software è infatti è già supportata per gran parte dei linguaggi di programmazione tra cui anche Objective-C che noi stiamo utilizzando, inoltre le sue dimensioni ridotte la rendono ideale nell'utilizzo all'interno di applicazioni per cellulare.
Alcune delle sue principali caratteristiche sono:

- è compatta, come già evidenziato, può occupare meno 500KB per l'intera libreria;
- è molto veloce, in molti casi più dei suoi principali concorrenti;

- il codice sorgente è liberamente disponibile;
- ha transazioni atomiche, consistenti, isolate e durabili (ACID), anche in caso di crash o blackout;
- è multipiattaforma;
- supporta anche database di dimensioni molto elevate (al momento il limite è 2TB).

Oltre a queste caratteristiche aggiungiamo anche che è possibile gestire i database SQLite o tramite linea di comando o tramite apposite interfacce grafiche, alcune delle quali anche direttamente supportate in alcuni dei browser più comuni. L'interfaccia grafica più comune è SQLiteManager, multipiattaforma e basata sul web che consente di amministrare in modo semplificato i database di SQLite.

Per la realizzazione del database SQLite della nostra applicazione invece inserirò, vista la semplicità del database, i comandi SQL direttamente dal terminale senza fare uso di programmi esterni di interfaccia grafica.
Passiamo quindi a vedere come viene realizzato in pratica un database SQLite prendendo come esempio il database che verrà poi utilizzato per l'applicazione di gestione dei codici a barre e QR.

5.2.1 Schema E-R del Database

Iniziamo con il fare alcune considerazioni anche se la nostra base di dati è molto semplice.
Avremo bisogno sicuramente di una tabella che mi contenga le informazioni necessarie per un codice a barre come ad esempio il suo codice, una sua descrizione e probabilmente anche un campo per salvare il percorso di dove è salvata l'immagine del codice a barre (vedremo successivamente che questo campo risulterà non necessario in quanto andremo a generare l'immagine al volo per ogni codice).
Un altra tabella a questo punto si può creare per tenere traccia degli utenti e delle loro informazioni, visto che con la classe UserDefault potevamo al massimo tenere traccia solo dell'ultimo utente che aveva inserito le sue credenziali, in questo modo potremo salvare più utenti e gestire in questo modo in maniera più intelligente il salvataggio dei codici a barre, visualizzando, al momento della richiesta, solo i codici a barre salvati dall'utente attualmente loggato e dell'utente di default.
Tutto questo si traduce nel modello entità-relazione come si può vedere nella Figura 5.1. Abbiamo quindi due relazioni ***utente*** e ***codice_a_barre*** ed una sola relazione che le unisce ***ha_salvato*** di tipo *uno a molti* con la parte molti rivolta verso i codici a barre, quindi quando andremo a creare fisicamente le tabelle dovremo ricordarci di inserire nella tabella dei codici a barre un

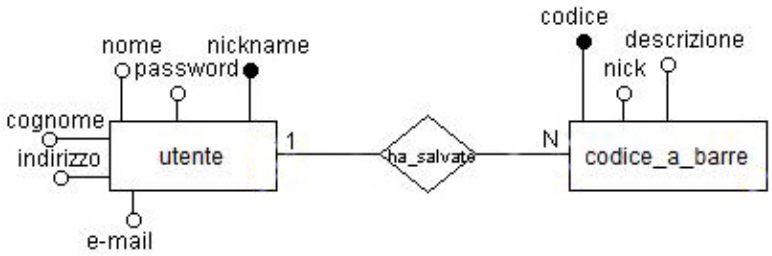

Figura 5.1: Schema ER con relazione 1-N

campo aggiuntivo che sarà la chiave esterna dell'entità utente nell'entità codice a barre.
Da segnalare che la stessa chiave esterna è anche parte della chiave primaria della stessa tabella.

Attraverso questo schema riusciamo quindi a gestire il fatto che più utenti possano salvare lo stesso codice a barre e un codice a barre appartiene quindi a più utenti. Detta in questo modo sembrerebbe che la relazione non fosse di quelle 1-N ma M-N (molti a molti), in realtà questo schema rappresentato in Figura 5.2 sarebbe concettualmente più corretto di quello da noi utilizzato ma di fatto lasciava un'entità, e quindi successivamente una tabella, con un solo attributo e chiave primaria, che non è propriamente elegante da vedere. Si è quindi optato per una scelta meno corretta ma che utilizza la base di

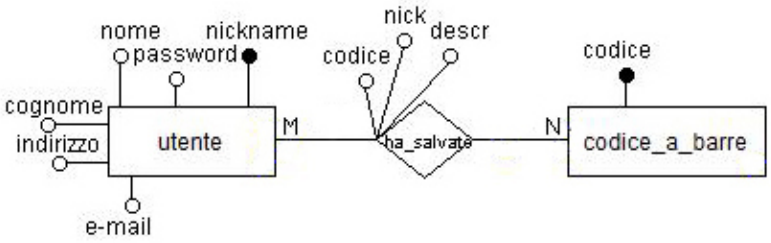

Figura 5.2: Schema ER con relazione M-N

dati in modo più efficace e con l'utilizzo di una tabella in meno, (ricordiamo che la conversione di una relazione M-N si traduce nella realizzazione di una tabella aggiuntiva a rappresentare appunto la relazione) rendendo così più semplice anche le interrogazioni allo stesso database.
Possiamo quindi passare all'implementazione della base di dati concentrandosi nei comandi utilizzati per creare le varie tabelle e le principali interrogazioni che andremo a porre al database nel corso dell'utilizzo della nostra applicazione.

5.2.2 Realizzazione del Database

Per realizzare il database che servirà per la nostra applicazione, dovremo come prima cosa crearlo; per fare ciò apriamo il terminale del nostro Mac e posizioniamoci nella directory nella quale vogliamo che venga creato e lanciamo il seguente comando:

```
sqlite3 nome_database.sqlite
```

a questo punto SQLite3 verrà avviato e il nostro database caricato e pronto per essere utilizzato. A questo punto possiamo dare i comandi per creare le tabelle che ci servono all'interno della base di dati, il codice è il seguente:

```
CREATE TABLE user (
    nickname TEXT NOT NULL,
    password TEXT NOT NULL,
    name TEXT,
    lastname TEXT,
    address TEXT,
    e_mail TEXT,
    PRIMARY KEY (nickname)
);
```

i comandi per la creazione della seconda e anche ultima tabella sono:

```
CREATE TABLE barcode (
    code TEXT NOT NULL,
    nick TEXT NOT NULL,
    descr TEXT,
    PRIMARY KEY (code, nick),
    FOREIGN KEY (nick) REFERENCES user (nickname)
);
```

Come già accennato precedentemente vediamo che la chiave esterna della tabella utente è anche chiave primaria della tabelle codici a barre rendendo così possibile avere lo stesso codice a barre per più utenti.
Vediamo anche velocemente come sarebbe cambiata la struttura delle tabelle nel caso avessimo utilizzato una relazione M-N, visto che comunque questa struttura del database è stata utilizzata.

La tabella riguardante gli utenti sarebbe rimasta uguale mentre sarebbe cambiata quella relativa i codici a barre:

```
CREATE TABLE barcode (
    code TEXT NOT NULL,
    PRIMARY KEY (code)
);
```

Vediamo quindi che la tabella dei codici a barre sarebbe stata composta da un solo attributo rendendo essenzialmente questa tabella del tutto inutile e vedremo che la tabella risultante dalla risoluzione della relazione è del tutto simile con la tabella dei codici a barre attualmente implementata, andiamo quindi a vedere come si sarebbe tradotta questa relazione in codice SQL:

```
CREATE TABLE has_saved (
    nick TEXT NOT NULL,
    code TEXT NOT NULL,
    descr TEXT,
    PRIMARY KEY (nick, code),
    FOREIGN KEY (nick) REFERENCES user (nickname),
    FOREIGN KEY (code) REFERENCES barcode (code)
);
```

Guardando queste ultime righe di comando si può capire per quale motivo siamo passati alla versione con solo due tabelle, di fatto solo le tabelle *user* e *has_saved* contengono tutte le informazioni delle necessarie per il funzionamento della nostra applicazione.
Andiamo quindi a vedere le interrogazioni che andremo a porre alla base di dati e i comandi di inserimento e eliminazione dei dati al suo interno:

- controllare se un determinato utente e' gia' registrato e selezionare le sue informazioni:

  ```
  SELECT * FROM user WHERE nickname = 'usernick';
  ```

- inserire un nuovo utente:

  ```
  INSERT INTO user VALUES ('usernick', 'usurpwd', 'username',
                           'userlastname', 'useraddress', 'usermail');
  ```

- eliminare un utente esistente:

  ```
  DELETE FROM user WHERE nickname = 'usernick';
  ```

- controlla se l'utente ha dei codici a barre associati:

  ```
  SELECT * FROM barcode WHERE nick = 'usernick';
  ```

- e nel caso li elimina:

```
DELETE FROM barcode WHERE nick = 'usernick';
```

- inserimento di un nuovo codice a barre:

```
INSERT INTO barcode VALUES ('codicebarcode', 'usernick',
                                                'descrizione');
```

- controlla se l'utente ha un determinato codice a barre associato al suo nome:

```
SELECT * FROM barcode WHERE code = 'codicebarcode' AND
                                                nick = 'usernick';
```

- e nel caso fosse necessario lo elimina:

```
DELETE FROM barcode WHERE code = 'codicebarcode' AND
                                                nick = 'usernick';
```

Come si può notare nelle query di controllo non andiamo ad utilizzare la funzione **COUNT** per sapere quante righe ci vengono restituite al termine dell'interrogazione, ma utilizziamo una semplice **SELECT ***, questo perché di fatto a noi non ci interessa sapere il numero esatto delle righe che la query ritorna ma solo se l'interrogazione ha trovato almeno una riga che soddisfa le condizioni oppure no.
Per fare questo semplice controllo ci basterà utilizzare una funzione all'interno del codice *Objective-C* e fare un semplice controllo sulle righe.
Passiamo quindi a vedere come gestiamo la base di dati e le sue interrogazioni all'interno del codice dell'applicazione.

5.2.3 Gestione del DB in Objective-C

Per gestire il database e poterlo utilizzare bisogna come prima cosa importare il database all'interno dei file di progetto, questo è possibile cliccando col tasto destro sopra la cartella *Resources*, presente nel riepilogo dei file del progetto, andare su *Add* e quindi cliccare su *Existing File..* a questo punto basterà cercare il nostro database all'interno del hard disk e aggiungerlo ricordandoci di spuntare l'opzione che abilita la copia del file all'interno della cartella del progetto in questo modo avremo sempre una copia del database esterna al progetto che non verrà modificata nel caso in cui volessimo ripristinare il database ai valori di default.
Altra cosa molto importante da fare è quella di importare all'interno del progetto il framework relativo alla gestione dei database SQLite, il framework in questione da aggiungere va sotto il nome ***libsqlite3.0.dylib***, la procedura di inserimento di un nuovo framework è molto simile a quella vista per l'inserimento del file del database solo che invece che cliccare su ***Existing File***

clicchiamo su ***Existing Framework*** e selezionare dal menù il framework desiderato.

Una volta portata la base di dati all'interno del progetto e inserito il framework dobbiamo fare in modo da poter utilizzare il database creando la classe necessaria per gestirla.
Creiamo quindi la coppia di file .h e .m ricordandoci di creare la classe già come sotto classe di NSObject. Apriamo quindi il file .h e modifichiamolo nel seguente modo:

```
#import <Foundation/Foundation.h>
#import <sqlite3.h>

interface DatabaseData :  NSObject {

    NSMutableArray *lista;

}

- (id)init;
- (BOOL)createEditableCopyOfDatabaseIfNeeded;
//metodi che realizzano le query viste nella sezione precedente
- (unsigned)getSize;
- (id)objectAtIndex:(unsigned)theIndex;

property (nonatomic, retain)NSMutableArray *lista;

end
```

Commentiamo queste poche righe di codice: possiamo vedere subito l'inserimento della libreria ***sqlite3.h*** necessaria per l'accesso al database e l'esecuzione delle query. Definiamo anche un array di oggetti che possono essere diversi nel tempo (l'array è infatti definito mutable) in questo modo possiamo inserire oggetti diversi che provengono dall'esecuzione di interrogazioni al database (nel nostro caso le tabelle user e barcode contengono solo oggetti di tipo stringa).
Metodo che merita di essere preso in considerazione è *createEditableCopyOfDatabaseIfNeeded*, questo metodo è molto importante perché il database che abbiamo inserito all'interno del progetto si trova in una posizione dalla quale può essere solo letto e non modificato, questo metodo quindi si prende l'onere di copiare il database dalla sua posizione di default in una posizione all'interno del sistema operativo che sia editabile, rendendo così disponibile la possibilità di inserire nuove voci all'interno del database o di eliminarle.
I metodi *getSize* e *objectAtIndex* vengono utilizzati per accedere al *Muta-*

bleArray e poter estrarre le informazioni al suo interno. Infine mancano nella definizione dell'interfaccia tutti i metodi che vanno ad eseguire effettivamente le query e le andremo ad analizzare più in dettaglio nella sezione seguente.

5.2.4 Esecuzione di una Query in Objective-C

Analizziamo quindi il codice che porta all'esecuzione di una query all'interno di un database e a gestire le informazioni che l'interrogazione potrebbe restituire.
Tutti i metodi hanno una struttura iniziale abbastanza simile e tutti i metodi effettuano subito un controllo sulla editabilità del database creando successivamente una copia della base di dati se questa non è già stata creata. Altro passo comune a tutti i metodi è quello dell'instaurazione di un canale di comunicazione con il *DB* e una volta fatto ciò l'esecuzione dell'interrogazione e il controllo se quest'ultima ha avuto successo oppure no. La struttura a livello di codice risulta quindi la seguente:

```
- (void)metodoEsecuzioneQuery:(NSString *)dbPath /*lista attributi*/ {
    if([self createEditableCopyOfDatabaseIfNeeded]) {
        //copia editabile del database creata
     }
    else {
        //copia del database non creata
    }
    if (sqlite3_open([dbPath UTF8String], &database) == SQLITE_OK) {

        //database aperto correttamente
        //definizione della query SQL come variabile const char *
        if (sqlite3_exec(database,querySQL, NULL,
                                 NULL, NULL) == SQLITE_OK) {
            //query eseguita con successo
        }
        else {
            //query non eseguita
        }
    }
    else {
        //errore in apertura del database
    }
}
```

Questa struttura è quella ideale per le query di inserimento, modifica e eliminazione di voci all'interno della base di dati, nel caso delle SELECT i comandi sqlite3 sono leggermente diversi.
Infatti non si utilizza più il comando *sqlite_exec* ma si utilizza il comando *sqlite_prepare_v2*, quest'ultimo oltre ai parametri del database e della querySQL deve passare anche una variabile di tipo ***sqlite3_stmt ***. Questo comando di fatto esegue la query richiesta e ritorna il risultato della stessa nella variabile di tipo sqlite3_stmt che gli abbiamo passato come parametro, a questo punto eseguendo il comando *sqlite3_step* e passandogli come unico parametro sempre la variabile *stmt* possiamo eseguire singole operazioni scandendo le righe ritornate dall'interrogazione di tipo SELECT. La struttura delle funzione diventa quindi:

```
//dopo la corretta apertura del databse
sqlite3_stmt *selectSatement;
if(sqlite3_prepare_v2(database, querySQL, -1,
                      &selectStatement, NULL) == SQLITE_OK) {

    while(sqlite3_step(selectStatement) == SQLITE_ROW) {
        //operazioni sulle righe
    }

}
sqlite3_finalize(selectStatement);
```

Visto che stiamo analizzando nel dettaglio come eseguire le varie query in codice Objective-C, vediamo anche quali sono i modi per dichiararle.
Se l'interrogazione che dobbiamo eseguire è statica, cioè non cambia nel tempo possiamo direttamente definirla come una variabile di tipo *const char ** nel seguente modo:

```
const char *querySQL = definizione dell'interrogazione;
```

se invece abbiamo una query più complessa e con dei parametri che vengono forniti dall'esterno avremo bisogno di gestire la cosa attraverso la costruzione di una stringa con tutta la struttura dell'interrogazione e quindi convertire la stringa in una variabile char.
Per concatenare due stringhe fra di loro useremo la seguente forma:

```
NSString *querySQL = @SELECT * FROM user WHERE nickname = ';
querySQL = [querySQL stringByAppendingString:userNickname];
```

in questo modo possiamo creare una stringa composta lunga quanto necessario e con tutte le informazioni richieste.
Una volta completata la generazione della nostra stringa contenente l'interrogazione SQL corretta possiamo passare a convertirla per renderla utilizzabile dalle funzione *sqlite3* viste precedentemente. Il tutto si risolve in una singola istruzione:

```
const char *sql = [querySQL UTF8String];
```

Arrivati a questo punto abbiamo tutte le informazioni necessarie per creare i nostri metodi che implementano le interrogazioni per il database , possiamo quindi completare la definizione della classe che gestisce la base di dati e utilizzarla all'interno della nostra applicazione come fosse una qualsiasi altra classe.

Capitolo 6

La Prima Applicazione

Dopo aver visto una panoramica degli strumenti necessari per il nostro lavoro iniziamo a mettere in pratica quello che abbiamo imparato creando la nostra prima applicazione grafica per iPhone. Come obbiettivo per questa applicazione ci poniamo quello di riuscire a gestire almeno due viste dove in una di queste possiamo inserire alcuni dati personali, salvarli e quindi renderli disponibili all'altra pagina.

Iniziamo quindi con l'avviare Xcode e creare un nuovo progetto. Ci viene subito proposto con quale templare iniziare in modo da avere una base di partenza sulla quale lavorare, per i nostri scopi ci basterà partire con una **View-based Application** che si trova sotto il menù *iOS* nella sezione *Application*.

A questo punto ci verrà chiesto che nome dare al nostro progetto, possiamo chiamarlo *UserProfile*, e una volta confermato ci comparirà la finestra del nostro ambiente di sviluppo. La finestra è suddivisa, almeno per quanto riguarda la versione 3.2.6 di Xcode, in tre parti:

- nella sinistra troviamo una sezione che viene definita **Groups & Files** che non è altro che una visualizzazione dei vari file che stiamo utilizzando per creare la nostra applicazione. La visualizzazione delle cartelle in questa sotto-finestra non rappresenta la reale disposizione dei file presenti su disco, vengono solo raggruppati in questo modo per migliorare la gestione del progetto;
- nello spazio restante, la parte superiore della finestra visualizza i file all'interno della cartella/gruppo evidenziati nella sezione precedente;
- nell'ultima finestra invece viene visualizzato l'ultimo file sorgente che è stato evidenziato in una delle due sezioni descritte nei punti precedenti.

Iniziamo quindi con il dare un'occhiata ai file che l'ambiente di sviluppo ha già creato per noi. Nella cartella **Classes** troviamo due file *.m* con relativi file *.h*, i primi con nome **UserProfileAppDelegate** sono di fatto i file che gestiscono le funzioni principali di inizializzazione, chiusura e comunicazione delle notifiche al delegato principale dell'applicazione e che quindi noi non andremo ad esplorare o a modificare. Il secondo gruppo di file invece, che prende il nome **UserProfileViewController**, gestisce di fatto la vista principale dell'applicazione, questa è infatti la prima pagina che viene caricato quando lanciamo il nostro programma. Piccola precisazione, in pratica i file *UserProfileViewController.m e .h* controllano quello che avviene all'interno della vista come ad esempio gestire gli oggetti presenti all'interno della stessa o l'azione da compiere alla pressione di un determinato pulsante o quando viene caricata per la prima volta la pagina, ma di fatto questo file non definisce l'interfaccia grafica della vista, questo compito è lasciato ad un altro file presente all'interno della cartella **Resources** che ha estensione *.xib* e che viene gestito in maniera grafica da **Interface Builder (IB)**.

Ai fini della nostra applicazione utilizzeremo i file di **UserProfileViewController** come maschera per altre due viste che andremo a creare successivamente, in questo modo possiamo visualizzare un menù di navigazione e un titolo comune a tutte le pagine.

6.1 Creazione dell'Interfaccia Grafica

Prima di gestire la grafica e la gestione degli oggetti della pagina principale, creiamo i file per gestire le altre due viste. Andiamo quindi nella barra dei menù in alto e clicchiamo su File → New File, a questo punto si aprirà una schermata molto simile a quella iniziale per scegliere il tipo di progetto e dal solito menù **iOS** andiamo nella sezione **Cocoa Touch Class** e scegliamo il file di tipo **UIViewController Subclass** e prima di dare conferma ricordiamoci di abilitare l'opzione che crea anche il file XIB associato (With XIB for user interface), diamo quindi un nome al file, ad esempio **SeeUserProfileController** e diamo il comando *Finish*. In questa schermata le opzioni dovrebbero giá essere impostate correttamente, controlliamo comunque che l'opzione che genera anche il file *.h* sia attiva.
Un altro modo per eseguire la stessa operazione è cliccare con il tasto desto del mouse sopra la cartella nella quale vogliamo aggiungere il file, nel nostro caso **Classes**, e selezionare dal menù a tendina l'opzione Add → New File e quindi seguire le operazione appena descritte.
Ricordiamoci a questo punto di spostare il file *.xib* generato nella cartella **Resources** e gli altri due file nella cartella *Classes*, semplicemente trascinando i file nel gruppo desiderato. Creiamo quindi con la stessa procedura un altro set di file con nome **SetUserProfileController**.

A questo punto iniziamo a mettere le mani nella realizzazione dell'interfaccia grafica, doppio click quindi su il primo file che abbiamo creato, *SeeUserProfileController.xib*, si aprirà quindi il programma Interface Builder composto da 4 finestre:

- una finestra *Library* nella quale vengono elencati gli oggetti e le classi che possiamo aggiungere all'interfaccia grafica;
- una finestra *Inspector* che ci permetterà di modificare gli attributi degli oggetti inseriti nelle viste;
- una finestra che porta il nome del file che abbiamo aperto e che visualizza gli oggetti che abbiamo inserito nell'interfaccia grafica;
- e l'ultima finestra ci visualizza l'aspetto che avrà la nostra vista.

Figura 6.1: Vista del file SetUserProfileController.xib

Dal menù della finestra *Library* trasciniamo nella vista tre coppie di oggetti **Label** e **TextField** e un oggetto **RoundRectButton**. Disponiamoli come in Figura 6.1 ricordandoci di lasciare un po' di spazio sia in alto che in basso della schermata per la visualizzazione degli oggetti della vista principale.
Selezioniamo ora uno ad uno gli oggetti appena inseriti e modifichiamone alcune caratteristiche attraverso la finestra *Inspector*.

- selezioniamo i tre oggetti **Label** e modifichiamo le caratteristiche *Text* nella prima scheda e diamogli come valore Nome, Cognome ed Età;
- selezioniamo quindi anche il bottone e modifichiamo il campo *Title* e mettiamogli come valore Reset. Questo pulsante infatti ci consentirá di resettare i valori inseriti precedentemente nel profilo.

Salviamo tutto (*File* $\rightarrow$ *Save* o *Command* + *S*) e chiudiamo il file.
Apriamo quindi il file *SetUserProfileController.xib* e modifichiamolo nello

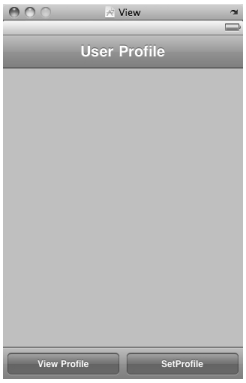

Figura 6.2: Vista del file UserProfileViewController.xib

stesso modo del precedente sempre con tre coppie **Label** e **TextField** e un bottone questa volta però diamo a quest'ultimo l'etichetta *Save*. Questo bottone infatti eseguirà l'operazione di salvataggio delle informazioni inserite all'interno del form.

Ultimo ma non meno importante è il file *.xib* della vista principale, apriamo anche quest'ultimo e aggiungiamo prima di tutto due oggetti **ViewController** nella finestra che visualizza i vari oggetti presenti nella interfaccia grafica, selezioniamo uno ad uno gli oggetti appena inseriti e per ognuno di loro attraverso l'utilizzo della finestra **Inspector** modifichiamo alcuni parametri. Apriamo la scheda **Identity** (*Command + 4*) e selezioniamo dal menù a tendina dell'opzione *Class*, il nome della vista che vogliamo sia visualizzata da quell'oggetto (es. SeeUserProfileController). Andiamo ora nella prima scheda di attributi (*Command + 1*) e dal menù a tendina dell'opzione **NIB Name** mettiamo il nome corrispondente a quello scelto per la classe.
Aggiungiamo ora in basso alla vista una **Toolbar** la quale si presenta giá con un bottone incorporato, noi dovremo aggiungerne un altro quindi prendo e trascino nella barra un oggetto **BarButtonItem**, modifichiamo le etichette a nostro piacimento, o come in Figura 6.2, e come ultimo tocco aggiungiamo una **Navigationbar** in alto alla vista e cambiamo anche a quest'ultima l'etichetta.
Alla fine il file *.xib* della vista principale dovrebbe contenere gli stessi oggetti presenti nella Figura 6.3. A questo punto la parte grafica e finita e possiamo chiudere Interface Builder; ricordiamoci una volta modificato le viste di salvarle o utilizzando la barre dei menù in altro o alternativamente con la scorciatoia *Command + S*.

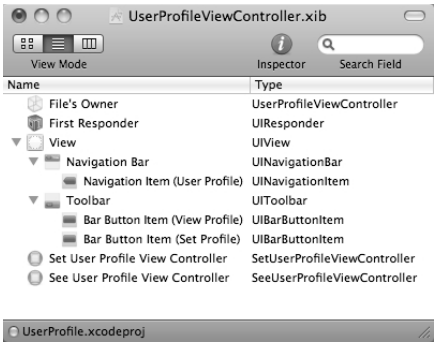

Figura 6.3: Oggetti nel file XIB di UserProfileViewController

6.2 Modifica dei File Sorgente

Andiamo ora a modificare i file di controllo in modo che gestiscano gli oggetti che abbiamo inserito nell'interfaccia grafica. Iniziamo con i due file di header *SeeUserProfileController.h* e *SetUserProfileController.h* che almeno in questa parte sono identici tranne per il piccolo dettaglio che il secondo file deve dichiarare l'utilizzo del delegato `<UITextFieldDelegate>`.
Modifichiamo quindi i due file nel seguente modo:

```
@interface SeeUserProfileController :  UIViewController {

    IBOutlet UITextField *nameField;
    IBOutlet UITextField *lastNameField;
    IBOutlet UITextField *ageField;

}

- (IBAction)resetData:(id)sender; //saveData nel file SetUserProfile

@property (nonatomic, retain) UITextFiled *nameField;
@property (nonatomic, retain) UITextField *lastNameField;
@property (nonatomic, retain) UITextField *ageField;

@end
```

Ora per quanto riguarda la differenza di cui si parlava, questa aggiunta del delegato ci consente di gestire la visualizzazione della tastiera nel momento in cui si inserisce del testo all'interno del form in modo da poterne consentirne la chiusura alla pressione del tasto *return*. Quindi per l'header del file **SetUserProfileController** va modificato nel seguente modo:

```
@interface SetUserProfileController :  UIViewController
                                        <UITextFieldDelegate>
```

Fermiamoci ora un attimo per capire quello che abbiamo fatto fino ad adesso: all'interno dell'interfaccia abbiamo dichiarato tre attributi di tipo **UITextField** che non sono altro che dei riferimenti alle caselle di testo che abbiamo inserito nelle due interfacce grafiche, il comando **IBOutlet** serve a noi per

poter fare i collegamenti successivamente delle variabili con gli oggetti dell'interfaccia grafica attraverso l'Interface Builder.
Cosa molto simile riguarda il metodo **saveData**. L'oggetto IBAction ritornato dal metodo ci consentirà di fatto di poter linkare il metodo al relativo oggetto nell'interfaccia grafica, nel caso più comune ad un bottone.
Le istruzioni **@property** consentono al momento dell'invocazione del comando **@synthetize** di generare automaticamente i metodi *getter* e *setter* per quel determinato attributo.

Passiamo quindi alla modifica dei file *.m* associati, la parte comune ai due file in questo caso è molto ridotta e si tratta di una semplice riga di codice scritta subito dopo il comando **@implementation**:

```
@synthesize nameField, lastNameField, ageField;
```

Passiamo ora alla sovrascrizione del metodo **viewDidLoad**.
Per il file **SeeUserProfileController** questo metodo servirà per caricare le informazioni del profilo negli appositi campi di testo in maniera automatica ogni volta che la pagina viene caricata. Il codice Objective C è il seguente:

```
- (void)viewDidLoad {

    [super viewDidLoad];

    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    [nameField setText:[userDefaults objectForKey:@nameFieldText]];
    [lastNameField setText:[userDefaults objectForKey:@lastNameFieldText]];
    [ageField setText:[userDefaults objectForKey:@ageFieldText]];

    nameField.enabled = NO;
    lastNameField.enabled = NO;
    ageField.enabled = NO;

}
```

Per il file *SetUserProfileController.m* invece scriviamo invece il seguente codice:

```
- (void)viewDidLoad {

    [super viewDidLoad];

    nameField.delegate = self;
    lastNameField.delegate = self;
    ageField.delegate = self;

}
```

Passiamo quindi all'implementazione dei rimanenti metodi dei file *.m*.
Per *SeeUserProfileController.m* implementiamo il metodo **resetData**, invocato quando premeremo il relativo bottone:

```
- (IBAction)resetData:(id)sender {
```

```
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    [userDefaults removeObjectForKey:@''nameFieldText''];
    [userDefaults removeObjectForKey:@''lastNameFieldText''];
    [userDefaults removeObjectForKey:@''ageFieldText''];

    nameField.text = @'''';
    lastNameField.text = @'''';
    ageField.text = @'''';

}
```

Passando al file **SetUserProfileController** implementiamo i seguenti metodi:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {

    [textField resignFirstResponder];
    return YES;

}

- (IBAction)saveData:(id)sender {

    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    NSString *nameFieldText = nameField.text;
    [userDefaults setObject:nameFieldText forKey:@''nameFieldText''];
    NSString *lastNameFieldText = lastNameField.text;
    [userDefaults setObject:lastNameFieldText forKey:@''lastNameFieldText''];
    NSString *ageFieldText = ageField.text;
    [userDefaults setObject:ageFieldText forKey:@''ageFieldText''];

    nameField.text = @'''';
    lastNameField.text = @'''';
    ageField.text = @'''';

}
```

Andiamo ora a modificare la vista principale e quindi i file *.h* e *.m* con nome **UserProfileViewController**. Partiamo dal file di header:

```
@class SeeUserProfileController;
@class SetUserProfileController;

@interface UserProfileViewController :  UIViewController {

    IBOutlet SeeUserProfileController *seeUserProfileController;
    IBOutlet SetUserProfileController *setUserProfileController;

}
```

```
@property (nonatomic, retain)SeeUserProfileController
                                        *seeUserProfileController;
@property (nonatomic, retain)SetUserProfileController
                                        *setUserProfileController;
- (IBAction)loadSeeUserView:(id)sender;
- (IBAction)loadSetUserView:(id)sender;

- (void)clearView;
```

Passiamo quindi al file *.m*. Innanzitutto importare i file .h delle altre due viste (`#import ''`*nomevista*`.h''`) all'interno dell'implementazione della classe facciamo come prima cosa il *synthesize* delle due proprietà precedentemente definite e implementiamo i restanti metodi nel seguente modo:

```
- (void)clearView {

    if (seeUserProfileController.view.superview) {
        [seeUserProfileController.view removeFromSuperview];
        seeUserProfileController.view = nil;
    }
    else {
        [setUserProfileController.view removeFromSuperview];
        setUserProfileController.view = nil;
    }

}

- (IBAction)loadSeeUserView:(id)sender {
    [self clearView];
    [self.view insertSubview:seeUserProfileController.view atIndex:0];
}

- (IBAction)loadSetUserView:(id)sender {
    [self clearView];
    [self.view insertSubview:setUserProfileController.view atIndex:0];
}
```

Sovrascriviamo anche il metodo **viewDidLoad** in modo che carichi automaticamente al primo avvio la pagina che visualizza il profilo:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self loadSeeUserView:nil];
}
```

Ora fermiamoci un attimo e facciamo alcune considerazioni sul codice visto

finora. Per quanto riguarda i tre metodi **viewDidLoad** la chiamata `[super viewDidLoad]`è strettamente necessaria in quanto stiamo sovrascrivendo un metodo della superasse e quindi bisogna richiamare l'esecuzione del metodo della classe padre.
Le altre righe di codice in **viewDidLoad** di *SeeUserProfileController.m* servono per accedere alla memoria del cellulare ed impostare le caselle di testo e disattivarle in modo che non possano essere modificate.
Il restante codice del metodo **viewDidLoad** in *SetUserProfileController.m* e il metodo **textFieldShouldReturn** agiscono in coppia garantendo la scomparsa della tastiera come già precedentemente accennato.
I metodi *saveData* e *resetData* si spiegano di fatto da soli. Il metodo *saveData* salva i dati in memoria e reimposta i campi in maniera che risultino vuoti, pronti per un eventuale nuovo inserimento di dati, mentre il metodo *resetData* elimina i dati salvati precedentemente e cancella le informazioni visualizzate dai campi di testo nella sua pagina in maniera del tutto identica a *saveData*.
Il metedo **clearView** come si può capire serve per pulire la schermata prima che un'altra vista venga caricata, altrimenti si avrebbe una poco chiara sovrapposizione delle pagine.

6.3 Link dei Metodi/Attributi agli Oggetti Attraverso IB

Siamo quindi arrivati alla fase conclusiva, questa fase si poteva anche eseguire subito dopo la modifica dei file di header infatti a quel punto si hanno tutte le informazioni necessarie per andare, grazie a Interface Builder, a porre in relazione i metodi e gli attributi presenti nel codice con gli oggetti presenti nelle viste della nostra applicazione.
Per fare questa operazione quindi riapriamo uno ad uno, se li abbiamo chiusi precedentemente, con doppio click i file .xib *UserProfileViewController*, *SeeUserProfileController* e *SetUserProfileController*.

Iniziamo con il primo dei tre e per collegare gli attributi ai relativi oggetti dobbiamo tenere premuto il tasto *Control(Ctrl)* e cliccare e trascinare il cursore del mouse dall'oggetto *File's Owner* a una dei due oggetti **UIViewController** e rilasciamo il cursore, a questo punto comparirà una finestra dalla quale selezionare l'oggetto di tipo IBOutlet definito nel codice, come si può vedere anche dalla Figura 6.4. Selezioniamo quello corretto e facciamo lo stesso anche per l'altro oggetto **UIViewController**.

Gestiamo ora le azioni che devono compiere i bottoni della vista unendoli con i rispettivi metodi nel file sorgente. Per fare ciò, come prima trasciniamo il cursore questa volta dall'oggetto bottone verso il *File's Owner*. Come

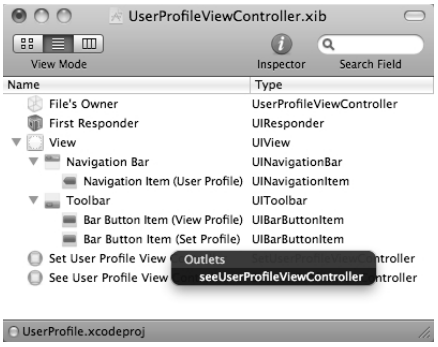

Figura 6.4: Esempio di linkaggio su file XIB

prima ci apparirà una finestra dalla quale dovremo selezionare il metodo che vogliamo che il bottone esegua.

Utilizziamo questo metodo anche per i restanti file *.xib* rimasti, quindi per le caselle di testo trascino da *File's Owner* agli oggetti dei campi di testo e per i bottoni da quest'ultimi al *File's Owner*.
Riusciamo ora a comprendere per quale motivo questa operazione va fatta almeno dopo aver definito i file di header, infatti se non avessimo avuto sia gli oggetti dell'interfaccia sia i metodi o attributi con i quali essi devono lavorare non saremo stati in grado di creare i collegamenti fra di loro.
Come al solito alla fine ricordiamoci di salvare tutto i file che abbiamo modificato, altrimenti non saremo in grado di vedere le modifiche apportate.

6.4 Esecuzione della nostra Prima Applicazione

Siamo quindi arrivati finalmente alla fine, abbiamo creato tutto quello che ci serve per la nostra applicazione con relativi collegamenti l'unica cosa che manca da fare è compilare il tutto ed eseguire il nostro programma sul simulatore.
Xcode sotto questo punto di vista ci permette di fare davvero poco fatica, infatti per eseguire le operazione descritte sopra basterà cliccare con il mouse sopra all'icona del martello con il cerchio verde per mandare in compilazione il codice, e nel caso sia tutto corretto verrà anche lanciato il simulatore con la nostra applicazione già attiva e in primo piano.
È molto probabile che l'applicazione venga eseguita sul simulatore iPad invece che iPhone, per modificare questa cosa basta cambiare il tipo di dispositivo simulato dal menù a tendina nella parte alta a sinistra della finestra del nostro ambiente di sviluppo e a questo punto far ripartire la simulazione.

Capitolo 7

OCR e Lettura di Codici a Barre e Codici QR

La parte fondamentale della nostra applicazione riguarda proprio la possibilità di riuscire a leggere codici a barre o codici QR e quindi con le informazioni raccolte riuscire a svolgere un determinato tipo di operazione. Di fatto noi non andremo a realizzare fisicamente il codice per andare a fare l'OCR di un barcode o codice QR ma utilizzeremo delle librerie che forniscono delle funzioni che permettono di effettuare già la conversione una volta fornita un'immagine appropriata tramite la macchina fotografica o di un'immagine già presente all'interno del cellulare.

7.1 Il Codice a Barre

Andiamo ora a vedere nel dettaglio come è strutturato un codice a barre e qual'è il modo per leggerlo e decodificarlo.
Un codice a barre come dice il nome è formato da delle linee (barre) che in base al loro spessore danno di fatto codifiche diverse. La lettura è relativamente semplice infatti si tratta di un codice unidimensionale che presenta dei pezzi di codice uguale ad ogni tipologia di codice a barre, ad esempio nel caso dei codici a barre EAN-13, che successivamente andremo ad analizzare in maniera più approfondita visto che saranno l'oggetto della nostra applicazione, l'inizio, la parte centrale e la parte finale di ogni codice è uguale a quella di tutti gli altri.

7.1.1 European Article Number EAN-13

Come già accennato precedentemente la nostra applicazione lavorerà principalmente con i codici a barre di tipo EAN-13 (*European Article Number*). In questa sezione andremo ad analizzare in dettaglio questo tipo di codice.

n	A	B	C
0	0001101	0100111	1110010
1	0011001	0110011	1100110
2	0010011	0011011	1101100
3	0111101	0100001	1000010
4	0100011	0011101	1011100
5	0110001	0111001	1001110
6	0101111	0000101	1010000
7	0111011	0010001	1000100
8	0110111	0001001	1001000
9	0001011	0010111	1110100

Tabella 7.1: Codifiche A, B e C

n	sequenza
0	AAAAAACCCCCC
1	AABABBCCCCCC
2	AABBABCCCCCC
3	AABBBACCCCCC
4	ABAABBCCCCCC
5	ABBAABCCCCCC
6	ABBBAACCCCCC
7	ABABABCCCCCC
8	ABABBACCCCCC
9	ABBABACCCCCC

Tabella 7.2: Sequenza di codifica della prima cifra

Iniziamo con la sua struttura: si tratta di un codice pluridimensionale dove possono essere presenti fino a 4 tipi diversi di spessore di barre/spazi, ognuno multiplo di un modulo unitario che nominalmente misura 0,33mm con una tolleranza di -20% / +100% (quindi può variare da 0,264 a 0,66mm). Di fatto i caratteri del codice possono essere codificati utilizzando una stringa di 7 bit dove 0 sta per una spazio bianco e 1 sta per una barra nera.

L'EAN-13 come suggerisce il nome, rappresenta un codice di 13 caratteri che possono avere 3 tipi di codifiche A, B e C come riportato in tabella 7.1.

I caratteri possono essere tutti rappresentati o altrimenti si può evitare di scrivere esplicitamente la prima cifra (usualmente posta al di fuori del codice a barre) poichè può essere identificata grazie a come sono codificati (A o B) i successivi 6 caratteri del codice a barre, come visualizzato in tabella 7.2. Sono inoltre sempre presenti i caratteri di start (101), stop (101) e controllo

centrale (01010).
Alla fine quindi un tipico codice a barre EAN-13 si presenta nel seguente modo:

- 101;
- (6 caratteri codificati in A o B);
- 01010;
- (6 caratteri codificati in C);
- 101.

EAN-13 viene utilizzato per la marcatura di prodotti destinati al mercato globale, vediamo ora cosa significano le cifre di questo codice:

- le prime due cifre identificano il paese dove è stata richiesta la codifica da chi detiene il marchio del prodotto (es. l'Italia è identificata dalle cifre 80 e 81);
- le successive cinque cifre identificano il produttore;
- le ulteriori identificano la denominazione del prodotto all'interno dell'azienda;
- l'ultima cifra rappresenta la cifra di controllo.

Per calcolare l'ultima cifra bisogna sommare tutte le cifre nelle posizioni pari fra di loro (contando che la prima cifra ha indice 0, quindi pari e ovviamente escludiamo da questa operazione l'ultima cifra) e fare lo stesso con le cifre nelle postazioni dispari, queste però vanno precedentemente moltiplicate per tre. A questo punto sommo fra di loro i valori ottenuti, la cifra di controllo si ottiene calcolando il valore che servirebbe per arrivare al primo multiplo di dieci superiore.
Ad esempio prendiamo il codice 1234567890123 e controlliamo se il codice di controllo è corretto:

- sommo le cifre in pos pari: 1 + 3 + 5 + 7 + 9 + 1 = 26;
- moltiplico per tre e sommo le cifre in posizione dispari: (2 + 4 + 6 + 8 + 0 + 2) * 3 = 66;
- sommo i valori ottenuti: 26 + 66 = 92;

Si vede che il primo multiplo di 10 più grande di 92 è 100, quindi 100 - 92 = 8 quindi la vera cifra di controllo per il codice a barre dato in precedenza è 8 e non 3.

L'applicazione che andremo a realizzare sarà in realtà in grado di riconoscere molti altri tipi di codice a barre come la versione ridotta del'EAN-13, l'EAN-8, o come l'UPC ma sarà compatibile e in grado di lavorare solo con gli EAN-13, per tutti gli altri si potrà solamente fare, una volta ricavate le cifre del codice, una semplice ricerca attraverso il motore di ricerca del browser.

7.2 Codice QR

Visto che questo tipo di codice viene letto dalla libreria che andremo ad implementare nella nostra applicazione è bene spendere alcune parole a riguardo.
Questo codice fu inventato da Toyota nel 1994 e venne quinidi inizialmente utilizzato nell'industria automobilistica proprio per tracciare i pezzi delle automobili nelle varie fabbriche della stessa azienda, successivamente vista la sua alta leggibilità e l'elevata capacità di immagazzinare informazioni ha iniziato ad essere utilizzato anche per altri scopi.
La quantità di data che è possibile immagazzinare in un codice QR dipende da vari fattori come il tipo di dato che si intende inserire al suo interno (solo numerico, alfanumerico, binario o Kanji[1]/Kana[2]), la versione del codice (indica la dimensione totale del codice) e il livello di correzione d'errore (L[ow] basso, M[edium] medio, Q[uality], H[igh] elevato). Il massimo numero di informazioni che possono essere contenute all'interno di un QR-code avviene quando si utilizza un codice alla versione 40 con un basso livello di correzione dell'errore.
In queste condizioni per ogni tipo di dato si possono inserire il seguente quantitativo di informazioni:

- **solo numerico**: max 7089 caratteri;
- **alfanumerico**: max 4296 caratteri;
- **binario**: max 2953 caratteri;
- **Kanji/Kana**: max 1817 caratteri.

L'algoritmo utilizzato per la correzione d'errore a quattro livelli permette di poter creare dei codici QR che contengono immagini o scritte che sono di fatto visti dai lettori come errori, ma il codice rimane comunque leggibile, infatti in base al livello del codice di correzione possiamo recuperare approssimativamente la seguente quantità di informazioni:

- **livello basso [L]**: ripristinati fino al 7% delle informazioni;

[1] Caratteri di origine cinese usati nella scrittura giapponese
[2] Termine generico per indicare i due sillabari fonetici giapponesi Hiragana e Katakana

- **livello medio** [**M**]: ripristinati fino al 15% delle informazioni;
- **livello qualità** [**Q**]: ripristinati fino al 25% delle informazioni;
- **livello elevato** [**H**]: ripristinati fino al 30% delle informazioni.

Ci rimane quindi solo da analizzare in maniera molto veloce la struttura di un codice QR, questo è composto da diverse parti come anche visualizzato in figura 7.1.

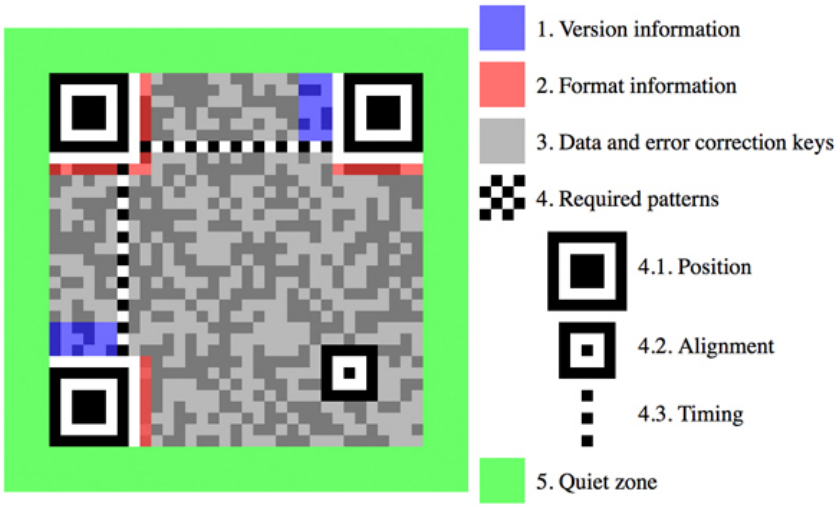

Figura 7.1: Campi codice QR

I campi versione e formato (evidenziati in blu e rosso rispettivamente) di fatto sentano quelle che sono le caratteristiche del codice a barre e che sono state descritte precedentemente, la sezione grigia è la parte che di fatto contiene le informazioni salvate nel codice QR e il codice di correzione dell'errore. Rimangono quindi solo da analizzare dei pezzi di codice che permettono ai lettori di eseguire delle decodifichi corrette, ad esempio i campi Posizione sono i tre "quadrati" posti negli angoli del codice a barre, questi sono sempre tre in tutti i codici e definiscono i limiti del codice QR, ci sono poi i campi di allineamento, la cui funzione è intuibile dal nome, che possono variare in numero in base alla dimensione del codice e nel caso della versione 1 del codice QR questo campo è assente. Un altro campo definito di timing, sono di fatto due righe di codice QR, una verticale e una orizzontale, che alternano un quadratino nero ad uno bianco e svolgono il compito di aiutare il lettore a rimanere "sincronizzato" nella scansione del codice.
Infine come anche già visto per il codice a barre anche per il QR viene definita un'area di contorno che deve rimanere libera, quindi bianca, se vogliamo che il nostro codice possa venire letto in maniera corretta.

Per concludere la trattazione sui codici QR diciamo che le specifiche esposte precedentemente non sono valide per la variante *Micro QR*, infatti questa essendo molto piccola presenta solo un campo di posizione e può contenere fino ad un massimo di 35 caratteri numerici.

7.3 SDK per la Lettura di Codici a Barre e QR

Per la risoluzione delle informazioni contenute all'interno dei codici QR e a barre, per la nostra applicazione finale, ci affidiamo a delle librerie con licenza libera che svolgono già queste funzioni. Andiamo quindi a presentare e a studiare questa libreria nelle sue funzioni principali.
Le librerie provengono dall'applicazione ZBar che si può anche trovare nell'iPHone App Store già funzionante, altrimenti vengono fornite appunto le librerie per poter creare le proprie applicazioni senza dover preoccuparsi di dover gestire tutta la parte di manipolazione dell'immagine per ottenere il risultato di un codice a barre o QR.
Andiamo ora ad analizzare la sequenza di passaggi necessari per poter inserire le seguenti SDK all'interno del progetto e quindi renderle utilizzabili dall'applicazione.

7.3.1 Installazione delle SDK

L'installazione delle SDK può avvenire principalmente in due modi:

- o attraverso il download del pacchetto contenente i sorgenti delle SDK di ZBar;
- attraverso il download dei singoli file di codice e quindi incorporarli all'interno del progetto.

Noi abbiamo utilizzato il primo dei due metodi e abbiamo quindi scaricato il pacchetto *.dmg* al link *http://zbar.sourceforge.net/iphone* e apriamo il file nel *Finder*. All'interno troveremo una cartella, con nome ZBarSDK, contenente le librerie da aggiungere all'interno del progetto, alcuni progetti di esempio e la documentazione necessarie per installare e utilizzare le SDK.
Per rendere queste librerie utilizzabili all'interno del progetto bisogna trascinare la cartella ZBarSDK nella sezione Groups & Files, a questo punto comparirà una finestra nella quale ci verrà chiesto a che progetto aggiungere i seguenti file, l'unica scelta è ovviamente il progetto corrente, e ricordiamoci di far si che i file vengano anche copiati fisicamente all'interno della cartella di progetto abilitando la relativa opzione.
A questo punto se non sono già stati aggiunti al progetto dovremo inserire i seguenti framework al progetto:

- AVFoundation.framework;

- CoreMedia.framework;
- CoreVideo.framework;
- QuartzCore.framework;
- libiconv.dylib.

L'ultimo file anche se con estensioni diversa si trova sempre nel menù per aggiungere framework (tasto destro sopra la cartella framework, Add → Exsisting Framework). Ora possiamo utilizzare le API per la lettura dei Barcode e QR in qualsiasi file, basta aggiungere la seguente riga di codice:

```
#import ‘‘ZBarSDK.h’’
```

7.3.2 Utilizzo all'Interno di un'Applicazione

Nella nostra applicazione andremo quindi a creare una nuova vista completa di file *.m*, *.h* e *.xib*. IN quest'ultimo andremo ad aggiungere un oggetto *ImageView*, un oggetto *TextView* e due bottoni *RoundRectButton* che serviranno per fornire il codice a barre tramite l'uso della fotocamra o attraverso un'immagine già salvata nel telefonino. A questo punto passiamo al file *.h* e modifichiamo l'interfaccia nel seguente modo:

```
#import <UIKit/UIKit.h>
#import ‘‘ZBarSDK.h’’

@interface MyNameViewController :  UIViewController
                                     <ZBarReaderDelegate> {
    IBOutlet UIImageView *resultImage;
    IBOutlet UITextView *resultText;
}
@property (nonatomic, retain) UIImageView *resultImage;
@property (nonatomic, retain) UITextView *resultText;
- (IBAction)scanButtonTapped;
- (IBAction)retriveImage;
@end
```

Possiamo quindi completare le connessioni con l'interfaccia, apriamo il file *.xib* e facciamo le seguenti connessioni:

- connettiamo l'outlet MyNameViewController `resultImage` all'oggetto ImageView;
- connettiamo l'outlet MyNameViewController `resultText` all'ogetto TextView;
- connettiamo l'action MyNameViewController `scanButtonTapped` all'ogetto RoundedRectButton(Scan Camera);
- connettiamo l'action MyNameViewController `retriveImage` all'ogetto RoundedRectButton(Scan Image).

Finiamo ora l'implementazione nel file *.m*, aggiungendo le seguenti righe di codice:

```
- (IBAction)scanButtonTapped {
    //implementare il codice per lettura barcode qui
}

- (void)dealloc {
    self.resultImage = nil;
    self.resultText = nil;
    [super dealloc];
}

- (BOOL) shouldAutorotateToInterfaceOrientation:
                        (UIInterfaceOrientation) interfaceOrientation {
    return(YES);
}
```

Implemento ora il codice per il metodo `scanButtonTapped`, che è del tutto simile all'altro metodo `retriveImage` , l'unica cosa per cui differiscono è il tipo di strumento di input che utilizzano e si traduce nel cambiamento di due sole righe di codice.
Andiamo quindi a vedere nel dettaglio:

```
- (IBAction)scanButtonTapped {
    ZBarReaderViewController *reader = [ZBarReaderViewController new];
    //in retriveImage cambiare ZBarReaderViewController
    //con ZBarReaderController

    reader.readerDelegate = self;
    reader.sourceType = UIImagePickerControllerSourceTypeCamera;
    //in retriveImage cambiare Camera con SavedPhotoAlbum

    ZBarImageScanner *scanner = reader.scanner;
    [scanner setSymbology:ZBAR_I25 config:ZBAR_CFG_ENABLE to:0];
    [self presentModalViewController:reader animated:YES];
    [reader release];
```

Per finire implementiamo il metodo delegato in modo che utilizzi in maniera consona i risultati ottenuti.

```
- (void) imagePickerController:(UIImagePickerController *)reader
                              didFinishPickingMediaWithInfo:
                                          (NSDictionary *)info {
    id<NSFastEnumeration> results = [info
                              objectForKey:ZBarReaderControllerResults];
    ZBarSymbol *symbol = nil;

    resultText.text = symbol.data;
    resultImage.image = [info objectForKey:
                              UIImagePickerControllerOriginalImage];
    [reader dismissModalViewControllerAnimated:YES];
}
```

E questo è tutto, possiamo compilare la nostra applicazione e farla eseguire, ricordiamoci che se siamo su simulatore l'utilizzo della fotocame-

ra è disabilitato e possiamo quindi solo utilizzare immagini che abbiamo precedentemente salvato.

7.3.3 Utilizzo delle Informazioni Raccolte

A questo punto quindi abbiamo la possibilità di scannerizzare l'immagine visualizzarla a schermo e di visualizzare il risultato della decodifica. Di fatto quindi a parte visualizzare il risultato ottenuto dalla decodifica dei codici a barre e QR non facciamo nient'altro. Andiamo quindi ad implementare un metodo che riconosca il risultato ottenuto e che esegua una ricerca attraverso il browser internet. Per fare questo aggiungiamo un bottone all'interfaccia grafica e colleghiamoci un metodo a livello di codice sorgente, come primo step mandiamo al browser qualsiasi cosa ci restituisca il campo di testo una volta effettuato lo scan, per fare ciò utilizzo il seguente codice all'interno di un metodo:

```
NSString *searchString = resultText.text;
[[UIApplication sharedApplication] openURL:
                                    [NSURL URLWithString:searchString]];
```

A questo punto però se passiamo come `searchString`qualcosa che non sia un URL l'applicazione non risponde, si può quindi cercare di migliorare il metodo facendo si che con il testo che ricaviamo lanciamo una ricerca attraverso Google.

```
NSString *searchString = resultText.text;
NSString *finalString = [NSString stringWithFormat:
                  @''http://www.google.com/search?q=%@'', searchString];
[[UIApplication sharedApplication] openURL:
                                    [NSURL URLWithString:searchString]];
```

Questo consente al nostro programma di essere sempre efficace anche se di fatto aggiunge un passaggio nel caso si abbia già un indirizzo valido. Facciamo quindi un'ultima miglioria in modo da riconoscere se ci viene fornito un URL lo passiamo direttamente al browser altrimenti per qualsiasi altra cosa facciamo una ricerca attraverso Google.

```
NSString *urlString = @''http://''
NSString *searchString = resultText.text;
if ([searchString hasPrefix:urlString])
    [[UIApplication sharedApplication] openURL:
                                        [NSURL URLWithString:searchString]];
else {
    NSString *finalString = [NSString stringWithFormat:
                      @''http://www.google.com/search?q=%@'', searchString];
    [[UIApplication sharedApplication] openURL:
                                        [NSURL URLWithString:finalString]];
}
```

Ovviamente questo codice riconosce solo gli indirizzi internet che iniziano con il testo "http://", se vogliamo rendere disponibili altri tipi di indirizzo che iniziano ad esempio con "www" o "https" basterà aggiungere delle condizioni nel costrutto *if/else*.

Capitolo 8

Tabella per Visualizzazione degli Oggetti

Una parte consistente del nostro tempo è stato speso per la realizzazione e corretta implementazione di una semplice tabella per l'inserimento delle informazioni di cui dovremo tenere traccia nelle nostre applicazioni.
Sebbene possa sembrare molto semplice realizzare una tabella per la visualizzazione dei dati, in realtà non è per niente banale, infatti non basterà una semplice etichetta `IBOutlet` all'interno del codice e quindi collegare il tutto su IB per rendere operativo questo tipo di oggetto.
L'oggetto tabella necessita della definizione di alcuni protocolli necessari al suo corretto utilizzo e quindi all'implementazione forzata di alcuni metodi derivanti appunto dai protocolli che abbiamo definito. Oltre a questo la tabella viene fornita con delle righe, che nel codice vengono definite celle, standardizzate che possono contenere soltanto un semplice testo al loro interno, se quindi dovessimo inserire altri tipi di dati all'interno delle righe della nostra tabella dovremo definire un nuovo tipo di cella aumentando così la complessità di realizzazione della tabella.
Nel seguito di questo capitolo andremo quindi a toccare tutti questi punti, dalla realizzazione dello scheletro di una tabella alla modifica e implementazione di celle personalizzate.

8.1 Gestione della Tabella Tramite IB

Iniziamo con l'impostare la visualizzazione della tabella tramite Interface Builder, quindi creiamo un nuovo file con *xib* associato e apriamolo. A questo punto come al solito iniziamo a trascinare gli oggetti che ci servono per la nostra interfaccia grafica.
Faccio notare che la tabella andrà ad occupare di default tutto lo spazio disponibile nella vista, se vogliamo quindi che occupi meno spazio basterà ridimensionarla o tramite interfaccia grafica o modificando le sue proprietà.

Una volta creata la grafica della nostra vista, andiamo a modificare il file *.h* associato, andando a prendere in considerazione solo la parte relativa alla gestione della tabella e trascurando tutto il resto:

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

interface TableViewController :  UIViewController <UITableViewDelegate,
UITableViewDataSource> {

    IBOutlet UITableView *table;
    NSArray *tableData;

}

property (nonatomic, retain)UITableView *table;
property (nonatomic, retain)NSArray *tableData;

end
```

A questo punto salviamo il tutto e torniamo nell'Interface Builder, da qui come al solito tenendo premuto il tasta *Control* colleghiamo l'Outlet della tabella definita nell'interfaccia della classe alla tabella vera e propria nella vista grafica, completata questa operazione il nostro lavoro non è finito, dobbiamo infatti portare a termine altri due collegamenti che sono di estrema importanza se volgiamo che la tabella sia utilizzabile.
Questi due collegamenti di cui parlavamo sono molto semplici da realizzare infatti basterà utilizzare la solita procedura solo che questa volta dovremo trascinare il cursore dall'oggetto tabella verso il File's Owner e selezionare, una alla volta, entrambe le voci che ci vengono visualizzate.
Quest'ultima operazione che abbiamo svolto sebbene sia di fatto molto semplice non è banale da scoprire per chi è la prima volta che si avvicina a questo tipo di programmazione ed è anche per questo motivo che si è deciso di affrontarli in dettaglio.

Andiamo ora a vedere a livello di codice sorgente come visualizzare i dati e lavorare sulla loro disposizione e modifica.

8.2 Gestione della Tabella in Xcode

La realizzazione del codice per la gestione della tabella creata tramite Interface Builder si risolve con l'implementazione di alcuni metodi, che fanno parte dei protocolli che abbiamo aggiunto nell'implementazione dell'interfaccia della classe, quelli fra i segni di maggiore/minore, che verrano lanciati automaticamente nel momento in cui vengono eseguite determinate azioni o

all'apertura della pagina in cui è presente la tabella.
Iniziamo con il primo metodo che definisce il numero di sezioni presenti nella tabelle, queste possono essere ovviamente impostate in maniera fissa, attraverso l'utilizzo di una costante, o attraverso l'utilizzo di una variabile. Le sezioni non sono altro che un modo per dividere la tabella in sotto-tabelle, ad esempio si può usare il seguente codice:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {

    return numeroDiSezioni;

}
```

NumeroDiSezioni può essere quindi qualsiasi cosa come già accennato precedentemente, il suo valori può essere calcolato internamente al metodo o acquisito dall'esterno, l'unica cosa importante è che sia di tipo intero per poter essere ritornato in maniera corretta alla funzione chiamante.
Una volta impostato il numero di sezioni andiamo a definire il titolo per ognuna di queste, per fare ciò si utilizza il seguente metodo:

```
- (NSString *)tableView:(UITableView *)tableView
                          titleForHeaderInSection:(NSInteger)section {

    //operazioni varie sulle variabili temporanee
    return titoloDellaSezione;

}
```

nella parte commentata del metodo vanno eseguite tutte le operazione necessarie a generare il giusto parametro di ritorno per la funzione, teniamo presente inoltre che questo deve essere per forza diverso da una costante nel caso in cui si hanno più di una sezione, infatti questo metodo viene chiamato più volte nel momento del caricamento della schermata, tante quante sono le sezioni presenti nella tabella.
Altro metodo che imposta la tabella è quello che imposta il numero di righe che deve avere ogni sezione ed è il seguente:

```
- (NSInteger)tableView:(UITableView *)tableView
                         numberOfRowInSection:(NSInteger)section {

    emph//operazioni sulle variabili temporanee
    return numeroDiRighePerLaSezione;

}
```

come si può notare questo metodo è molto simile al precedente ed anche lui viene chiamato più volte quante sono le sezioni presenti nella tabella, quindi le operazioni che si compiono prima di ritornare il valore servono ad impostare tale numero di righe in modo corretto per ogni sezione.
Continuando con l'analizzare i vari metodi utilizzati dobbiamo evidenziare quello che permette di impostare le righe e i valori che dovranno essere inserite nelle stesse e visto che il modo per utilizzare una cella personalizzata non è propriamente semplice ne viene riportato il codice completo nelle righe seguenti:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
                        cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    //impostazione delle celle standard o personalizzate
    static NSString *CellIdentifier = @BarcodeCustomCell;

    BarcodeCustomCell *cell = (BarcodeCustomCell *)
                        [tableView dequeueReusableCellWithIndentifier:
                                                        CellIdentifier];
    if (cell == nil) {
        NSArray *topLevelObjects = [[NSBundle mainBundle]
                                        loadNibNamed:@BarcodeCustomCell
                                        owner:nil options:nil];

        for (id currentObject in topLevelObject) {

            if ([currentObject isKindOfClass:[BarcodeCustomCell class]]) {
                cell = (BarcodeCustomCell *)currentObject;
                break;
            }
        }
    }

    //inserimento dei dati all'interno della riga/cella

}
```

Questa parte del codice può essere utilizzato per caricare qualsiasi cella personalizzata all'interno della tabella, basterà ovviamente modificare il nome che abbiamo dato alla classe che definisce la cella che nel nostro caso è BarcodeCustomCell. Una volta completata la parte che imposta le celle basterà inserire all'interno dei campi corretti i valori che otteniamo dal database o da qualsiasi altra fonte. Ultimo metodo che andremo ad analizzare ma non per questo meno importante degli altri è quello che permette di compiere delle azioni nel momento in cui selezioniamo una determinata cella all'interno della tabella. Il metodo è definito nel seguente modo:

```
- (void)tableView:(UITableView *)tableView
                    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    //operazioni da svolgere quando una cella della tabella
    //viene selezionata

}
```

Questo metodo può essere più o meno complesso in base a cosa decidiamo di fargli fare, non c'è alcuna parte già prestabilita, sta a noi realizzare l'intero metodo.
Nel caso della nostra applicazione questa funzione non fa latro che salvare su una variabile temporanea il valore del codice a barre della cella che abbiamo selezionato e richiamare la pagina di visualizzazione dei codici a barre, visualizzando quindi l'ultimo codice selezionato.

Nella prossima sezione andiamo a spiegare quali sono i passi per personalizzare le celle in modo che se dobbiamo inserire più di un singolo dato, come avviene per le celle standard, questo ci sia permesso.

8.3 Personalizzazione delle Celle

La prima cosa da fare per realizzare una cella personalizzata per la nostra tabella è quella di creare i file necessari per la sua realizzazione.
Iniziamo quindi con il creare i file sorgenti, per fare questo ci viene in aiuto il nostro ambiente di sviluppo che quando gli chiediamo di creare un nuovo file, selezionando la tipologia di classi di tipo Objective-C, ci permette di impostare i file automaticamente come sottoclassi della classe UITableViewCell, successivamente sempre seguendo la solita procedura di creazione dei file andiamo sotto la sezione *User Interface* e selezioniamo come tipo di file il *View XIB* ricordandoci di chiamare quest'ultimo file con lo stesso nome che abbiamo scelto per il precedente.

Creati entrambi i file possiamo iniziare modificando il file di intestazione all'interno del quale andremo a dichiarare quali sono gli oggetti che andremo ad utilizzare e che devono essere collegati con l'interfaccia grafica, non ci soffermeremo a vedere il codice di questo file perché l'abbiamo e lo vedremo ancora molte volte nel corso della tesi.

Passiamo quindi alla modifica dell'interfaccia grafica aprendo il file *.xib* creato in precedenza. Eliminiamo come prima cosa l'oggetto ***view*** presente nella schermata nella quale è presente anche l'oggetto File's Owner e sostituiamolo con un oggetto di tipo *Table View Cell* e cliccandoci sopra due volte apriamolo.
A questo punto ci si presenterà davanti una schermata molto simile a quella che avevamo prima con l'oggetto *view* soltanto molto più piccola nella quale inserire tutti gli oggetti che ci serviranno per la visualizzazione delle informazioni necessarie. Nel caso in cui la dimensione della cella di default non fosse di nostro gradimento possiamo sempre decidere di ampliarla (solo in senso verticale, orizzontalmente copre per forza di cose tutta la schermata) ricordandoci però che se effettuiamo questa modifica dobbiamo impostare la nuova grandezza delle righe della tabella, la cosa si può fare o impostando il valore tramite Interface Builder nelle proprietà della tabella o all'interno del codice utilizzando la seguente scrittura:

```
tableObject.rowHeight = heightOfTheRows;
```

Una volta completato questi passi abbiamo di fatto finito il lavoro, infatti

il file *.m* riguardante la cella non va modificato e le informazioni da inserire all'interno della stessa vanno gestite a livello di tabella e non a questo livello.

Capitolo 9

Realizzazione di una Select Box in iPhone

In iPhone la classica *Select Box* che si può trovare in vari form di registrazione nei siti internet, non è presente e viene sostituita da qualcosa di molto simile che si presenta come una "rotella" che facendola ruotare presenta le varie opzioni che si possono selezionare.
La classe che realizza questa funzione è l'*UIPickerView* che per le date ha già una sua classe specifica chiamata *UIDatePicker* dalla quale possiamo selezionare sia la data e l'ora, solo la data o solo l'orario.
L'implementazione sia a livello grafico che di codice ricalca in maniera molto simile quella per la tabella (*UITableView*), avremo quindi nell'interfaccia la dichiarazione di due protocolli quali *UIPickerViewDelegate* e *UIPickerViewDataSource*. Nel codice bisognerà quindi implementare i metodi dei protocolli che sono come forma sono del tutto simili a quelli già visti per la tabella.
Visto che la sintassi non è però uguale andiamo a vedere qual'è la sintassi di dichiarazione di questi metodi.
Il primo che andiamo a vedere è quello che imposta il numero di "ruote" di cui sarà composto il nostro oggetto, queste vengono definite "component":

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)thePickerView;
```

altro metodo è quello per definire il numero di righe per ogni componente della "ruota":

```
- (NSInteger)pickerView:(UIPickerView *)thePickerView
                        numberOfRowsInComponent:(NSInteger)component;
```

e quindi per poter inserire i dati al suo interno:

```
- (NSString *)pickerView:(UIPickerView *)thePickerView
                                titleForRow:(NSInteger)row
                                    forComponent:(NSInteger)component;
```

questi metodi appena descritti vanno obbligatoriamente implementati se vogliamo che il nostro oggetto possa lavorare in maniera corretta e non ricevere messaggi di errore o di avvertimento da parte del compilatore.
Come per la tabella anche qui si può fare in modo che nel momento in cui

selezioniamo un valore dall'UIPickerView vengano eseguite delle operazioni specifiche per quel determinato valore, si utilizza quindi il metodo:

```
- (void)pickerView:(UIPickerView *)thePickerView
                        didSelectRow:(NSInteger)row
                             inComponent:(NSInteger)component;
```

Implementando tutti questi metodi si riesce quindi ad ottenere un buon controllo di questo strumento riuscendo ad inserire, aggiornare ed eliminare se necessario i valori al suo interno a nostro piacimento.
Sebbene la somiglianza con la tabella possa trarre in inganno, per il PickerView non è strettamente necessario che questo venga inserito staticamente all'interno dell'interfaccia grafica, è infatti possibile renderlo disponibile ad esempio solo nel momento in cui andiamo a selezionare un campo di testo come viene fatto con la tastiera standard. In questo modo possiamo gestire più oggetti UIPickerView nella stessa pagina, visto che se li dovessimo inserire staticamente non potremmo andare oltre i due oggetti visto che la loro altezza è standard è non può essere modificata.
Se vogliamo realizzare questa funzione dobbiamo creare un'oggetto di tipo UIPickerView e impostarlo come metodo di input per il campo di testo che abbiamo deciso di utilizzare. Il tutto si traduce nel seguente codice:

```
UIPickerView *myPicker = [[UIPickerView alloc] init];

myPicker.tag = 0;
myPicker.delegate = self;
myPicker.dataSource = self;
myPicker.autoresizingMask = UIViewAutoresizingFlexibleWidth;
myPicker.showSelectionIndicator = YES;
myTextField.inputView = myPicker;
```

La maggior parte di queste righe riguardano semplicemente la inizializzazione del Picker, di particolare importanza risulta il campo tag che permette in caso di più oggetti dello stesso tipo di riuscire sempre ad accedere all'oggetto corretto ed infine l'ultima riga di codice che imposta l'oggetto UIPickerView in modo che compaia al posto della tastiera.
Se vogliamo che scompaia dopo una nostra determinata azione usiamo il seguente codice:

```
[myTextField resignFirstResponder];
```

se non utilizzassimo questo comando il nostro PickerView rimarrebbe sempre visibile fintanto che non si seleziona un'altro campo di testo o venga cambiata pagina.

Capitolo 10

Renderizzazione dei Codici a Barre

Sebbene il titolo possa trarre in inganno la nostra applicazione non sarà in grado di creare o generare in maniera corretta un'immagine da qualsiasi codice gli venga fornito in ingresso, ma solo da codici di tipo EAN-13. Questo viene reso possibile attraverso alcuni controlli preventivi che vengono fatti sui codici passati all'oggetto preposto alla realizzazione dell'immagine. Ovviamente il codice di verifica non è infallibile e può succedere che la nostra applicazione renderizzi un codice non EAN-13 anche se la cosa è altamente improbabile.
I controlli apportati sono due, il primo semplicemente sulla lunghezza della stringa, infatti se la stringa generata dopo la lettura dell'immagine misura più o meno di tredici sappiamo già che non è un codice EAN-13 e quindi non va renderizzato.
L'altro controllo eseguito dopo che è stato verificato il precedente non fa altro che controllare che l'ultima "cifra" verifichi le condizioni per essere la cifra di controllo corretta.
Questo controllo viene eseguito dalla seguente funzione:

```
- (BOOL)checkEAN13:(NSString *)bCCode {

    int even = 0;
    int odd = 0;
    int dim = [bCCode length];

    for (int i=0; i<dim-1; i=i+2) {
        NSString *tempStr = [bCCode substringFromIndex:i];
        char tempChr = [tempStr characterAtIndex:0];
        int x = (int)tempChr - 48;
        even = even + x;
```

```
    }

    for (int i=1; i<dim; i=i+2) {
        ..  //codice del tutto simile al for precedente
    }

    int checkNumSum;

    if ((even + (odd*3))%10 != 0) {
        checkNumSum = 10 - ((even + (odd*3))%10);
    }
    else {
        checkNumSum = (even + (odd*3))%10;
    }

    char checkNumChar = [bCCode characterAtIndex:12];
    int checkNumCode = (int)checkNumChar - 48;

    if (checkNumSum == checkNumCode) {
        return TRUE;
    }
    else {
        return FALSE;
    }

}
```

Una volta passati entrambi i controlli arriviamo quindi a poter renderizzare il nostro codice. Questa operazione comporta l'utilizzo di specifiche librerie create appositamente per poter creare e modificare immagini e che andremo ad introdurre nella sezione successiva.

10.1 CoreGraphics Framework

La libreria CoreGraphics è una delle due possibili librerie che si possono scegliere per creare e modificare immagini, l'altra è UIKit che però non ha le stesse potenzialità della prima e quindi non la utilizzeremo nella nostra applicazione per questo motivo.
Vediamo ora quali sono i comandi che andremo ad usare più frequentemente e tralasciamo completamente (vista la quantità) quelli che non andremo ad utilizzare nella nostra applicazione. Primo comando, forse tra i più importanti, è quello che definisce di fatto la creazione dell'immagine:

```
CGContextRef context = CGBitmapContextCreate
             (NULL, width * imageScale, height * imageScale,
               8, 0, colorSpace, kCGImageAlphaPremultipliedLast);
```

saltiamo la discussione di molti parametri comunque quelli che più saltano all'occhio sono *width * imageScale* e *height * imageScale* che come si deduce dal nome fissano di fatto la grandezza dell'immagine, il fattore *imageScale* viene settato di default a 1.0 e modificato a 2.0 solo nel caso si stia lavorando su un dispositivo ad alta risoluzione.
Altro parametro importante è il colorSpace che deve essere definito precedentemente nel seguente modo:

```
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
```

Oltre a RGB si può impostare che l'immagine sia in scala di grigi o in bianco e nero ecc..

Altro comando che utilizzeremo spesso all'interno del codice è:

```
CGContextSetRGBFillColor(context, red, green, blue, alpha);
```

Come si può intuire questo comando serve per fissare il colore che verrà utilizzato dai successivi comandi fintanto che questo non verrà cambiato. I parametri red, green, blue sono di tipo *CGFloat* e possono variare dal valore 0.0 a 255.0 mentre alpha sempre *CGFloat* ma questo varia da 0.0 a 1.0.
Una volta che siamo riusciti a settare il colore non ci rimane quindi che sapere come disegnare rettangoli e scrivere numeri, ci basteranno ovviamente solo queste cose per riuscire a disegnare completamente il nostro codice a barre.
Gli ultimi due comandi sono quindi:

```
CGContextFill Rect(context, CGRectMake(x, y, offsetx, offsety));
```

Il rettangolo viene quindi disegnato partendo dalle coordinate (x, y) (ricordiamoci che l'origine degli assi si trova in basso a sinistra) e termina a x + offsetx e y + offsety compresi. Questo vuol dire che se noi diamo come offset il valore 0.0 di fatto non vedremo disegnato nulla, se vogliamo ad esempio disegnare un semplicissimo pixel dovremo dare come valore agli offset 1.0.
Per quanto riguarda invece la scrittura di testi all'interno dell'immagine la cosa è leggermente più complicato perché richiede qualche comando preliminare prima di poter dare effettivamente l'ordine al programma di scrivere quello che vogliamo.
I comandi che dovremmo dare preliminarmente sono:

```
CGContextSelectFont(context, ''Arial'',
                    characterDim, kCGEncodingMacRoman);
CGContextSetTextDrawingMode(context, kCGTextFill);
```

Il comando che effettivamente da l'ordine di scrivere nell'immagine è:

```
CGContextShowTextAtPoint(context, x, y, text, strlen(text));
```

Anche qui i parametri sono abbastanza semplici da intuire, le coordinate x e y servono alla funzione per sapere dove posizionare il testo, *text* è ovviamente il testo che verrà scritto mentre *strlen(test)* da informazioni sulla lunghezza del testo che si andrà a scrivere.

A questo punto una volta presa la mano con questi comandi andiamo a scrivere il codice necessario per generare l'immagine di un codice a barre data la sua stringa numerica.
Per svolgere questa operazione abbiamo creato una nuova classe che va sotto il nome di ***BarcodeDrawer***.

10.2 Classe BarcodeDrawer

La classe BarcodeDrawer quindi rende disponibile la possibilità di poter disegnare un codice a barre, al momento solo di tipo EAN-13, dando semplicemente come unica informazione la stringa numerica del codice a barre. Andiamo quindi a vedere come questo viene gestito a livello di codice. Iniziamo con l'interfaccia, *BarcodeDrawer.h*:

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

interface BarcodeDrawer :  NSObject {

    int barDimension;
    int base;
    CGFloat imgWidth;
    CGFloat imgHeight;
    NSString *countryCodex[10];
    NSStrnig *code;

}

- (UIImage *)drawBarcode;

@end
```

È quindi presente solo un metodo pubblico che sarà anche quello che si prenderà carico di disegnare il codice a barre, notiamo che la stringa contenente il codice non viene passata direttamente al metodo ma verrà invece salvata come attributo nella variabile ***code*** di tipo NSString definita nell'interfaccia. Questo metodo inoltre non conterrà al suo interno tutta la generazione dell'immagine ma si avvarrà di altre funzioni, non visibili all'esterno della classe, che genereranno pezzi di immagine che poi il metodo principale dovrà solo posizionare nel modo corretto.
L'attributo contenente la stringa di codice, come anche tutti gli altri attributi, vengono settati al momento della chiamata del metodo ***init***, quindi all'inizializzazione dell'oggetto.
Iniziamo quindi a studiare alcuni delle funzioni presenti all'interno della classe BarcodeDrawer che sono implementate all'interno del file *BarcodeDrawer.m*.
Iniziamo scrivendo il codice per la funzione di inizializzazione:

```
- (void)init:(NSString *)newCode withBarDimension:(int)dim {

    [super init] //chiamata al metodo init della superclasse
```

```
    code = newCode;
    barDimension = dim;
    base = 10 * barDimension;
    imgWidth = 120.0 * barDimension;
    imgHeight = 100.0 * barDimension;
    countryCodex[0] = @AAAAAACCCCCC;
        .
        .
        .
    countryCodex[9] = @ABBABACCCCCC;

}
```

Questo metodo quindi imposta oltre alla stringa di codice anche la dimensione nominale della barra che rappresenta il bit a 1 nell'immagine, questo valore oltretutto imposta anche il resto degli attributi svolgendo da fattore moltiplicativo a dei valori base, in questo modo potremo avere un immagine più o meno grande in base a quanto elevato sarà il valore di *barDimension* che avremo impostato ma il tutto rimarrà sempre in scala.
Il concetto di scala è molto importante per quanto riguarda i codici a barre, infatti ci sono alcuni parametri minimi che devono essere rispettati perché il codice possa essere letto e riconosciuto come tale, ad esempio lo spazio bianco che deve essere lasciato ai margini destro e sinistro del codice.
Per quanto riguarda il metodo ***drawBarcode*** andiamo ad analizzare solo alcune parti che danno comunque l'idea di come questa funzione lavori. Iniziamo con la scrittura del codice e andiamo successivamente ad analizzarlo per parti:

```
-(UIImage *)drawBarcode {

    //inizializzazione di variabili temporanee e dello spazio di disegno

    [self drawBeginMiddleEnd:context];

    for (int i=0; i<6; i++) {
        char codeNum = [code characterAtIndex:(i+1)];
        char realCountryChar = [countryCodex[countryNum]
                                                characterAtIndex:i];
        int pos = (15*bardimension) + (i*(7*bardimension));

        [self drawCodePart:codeNum fromCountry:realCountryChar
                            inPosition:pos ofContext:context];

        NSString *temp = [code substringFromIndex:(i+1)];
        NSString *strTemp = [temp substringToIndex:1];
        char *text = (char *)[strTemp
                            cStringUsingEncoding:NASCIIStringEncoding];
        CGContextSelectFont(context, Arial, (12*barDimension),
                                                kCGEncodingMacRoman);
        CGContextSetTextDrawingMode(context, kCGTextFill);
        CGContextSetRGBFillColor(context, 0, 0, 0, 1);

        CGContextShowTextAtPoint(context, pos,
                            base - (10 * (barDimension)), text,
```

```
                                                    strlen(text));
    }

    //altro ciclo for che disegna la seconda parte del codice
    //a barre dopo lo 01010 di controllo

    //ritorna l'immagine dopo averla convertita in un formato approppiato

}
```

Allora, nella prima parte di commento oltre all'inizializzazione delle variabili e di tutto il resto viene anche di fatto inserita la parte di codice che scrive il primo numero, quello esterno al codice a barre, che identifica parte del codice del paese e che determina la successiva codifica delle prime sei cifre del codice a barre.
Completata questa prima parte, vado a disegnare i segmenti di controllo 101, 01010 e 101 rispettivamente all'inizio, nella parte centrale e alla fine del codice a barre sfruttando la funzione da noi precedentemente creata, definita:

```
- (void)drawBeginMiddleEnd:(CGContextRef)ctx;
```

alla quale devo quindi solo passare come parametro l'oggetto nel quale sto disegnando, in parole povere la nostra immagine. Si arriva quindi al primo dei due costrutti *for* che disegnano i dodici segmenti che rappresentano fisicamente il codice a barre. All'interno del ciclo si possono definire tre parti, una prima dove vengono settate tre variabili, la prima estrae il carattere del numero del codice a barre all'indice attualmente eseguito dal ciclo, la seconda variabile molto simile alla prima salva il tipo di codifica, A, B o C sempre in base al segmento che stiamo disegnando e l'ultima invece da la posizione di dove deve iniziare a disegnare le barre il metodo chiamato alla riga di codice successiva.
Arriviamo quindi al punto in cui viene fatta la chiamata ad un metodo che non fa altro che decidere, in base al carattere del codice a barre passato come parametro, quale sia il metodo corretto per disegnare quel determinato valore. Di fatto quindi il metodo ***drawCodePart*** non fa altro che decidere, in base ai parametri passati, quale sia il metodo corretto da chiamare per disegnare il codice ricevuto. La terza parte del codice di fatto imposta e scrive nell'immagine il numero in base al segmento di codice che sta venendo attualmente disegnato.
Tutto quello che vale per il primo ciclo vale anche nel secondo solo con indici ovviamente modificati in modo da andare a disegnare la seconda parte di codice che va dal segmento di controllo 01010 a quello finale 101.
Ultima parte riguarda la conversione dell'immagine che stiamo modificando in un oggetto di tipo UIImage, questo viene risolto dalle seguenti righe di codice:

```
CGImageRef cgImage = CGBitmapContextCreateImage(context);
UIImage *retImage = [UIImage imageWithCGImage:cgImage];
```

A questo punto non manca altro che rilasciare lo spazio di memoria per gli oggetti che non andremo più ad utilizzare come *context* e *cgImage* e quindi ritornare l'oggetto UIImage appena creato.

Non andremo a vedere come sono stati realizzati i metodi *drawBeginMiddleEnd* e *drawCodePart* perché di fatto non sono di alcun interesse, non fanno altro che disegnare una serie di barre in base al numero e alla sua codifica nel codice a barre.

Capitolo 11

Applicazione per la Lettura di Codici a Barre

Andremo ora ad analizzare qui di seguito quali sono le funzioni che questa applicazione dovrà essere in grado di fornire e la struttura delle pagine che l'applicazione presenteà all'utente finale.

11.1 Funzioni da Implementare nell'Applicazione

Vedremo di seguito una descrizione esaustiva delle funzioni che l'applicazione andrà ad implementare e i motivi per cui certe scelte sono state fatte in sede di realizzazione.

11.1.1 Gestione degli Utenti dell'Applicazione

L'applicazione può gestire la possibilità di salvare diversi profili utente, questo sebbene sia una funzione implementata più per una questione puramente didattica permette all'utente di salvare alcune delle proprie informazioni che potranno quindi essere utilizzate in automatico al momento dell'invio del SMS di conferma del pagamento facendo in modo che l'utente non debba inserire le stesse informazioni ogni volta che fa tale richiesta.
Inizialmente per implementare questa funzione si era fatto ricorso all'uso di variabili di sistema che tenessero in memoria semplicemente solo le ultime informazioni inserite dall'utente, quindi di fatto ci poteva essere un solo utente salvato in memoria del cellulare, quello attivo.
Successivamente con la necessità di dover tenere memoria dei vari codici a barre che erano stati scannerizzati nel tempo si è deciso di creare una tabella all'interno della base di dati che potesse contenere le informazioni degli utenti, in questo modo veniva anche reso possibile il fatto che un utente non potesse vedere i codici a barre che erano stati salvati e quindi scanerrizzati dagli altri utenti dell'applicazione e la possibilità ovviamente di avere più

profili salvati all'interno dell'applicazione e facilmente accessibili attraverso una pagina di login.

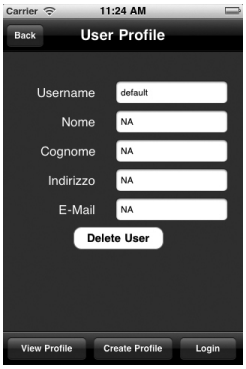

Figura 11.1: Pagina di scannerizzazione codici a barre

Per il modo in cui è stata realizzato e pensata l'applicazione non è possibile essere non connessi con almeno un utenza, nel momento in cui quindi un utente si slogga viene in maniera automatica connesso l'utente definito di default con il quale sarà possibile svolgere tutte le attività che possono svolgere gli altri utenti solo che le informazioni salvate rimarranno disponibili a tutti.

11.1.2 Salvataggio e Reperimento dei Codici a Barre

Altra funzione essenziale per il programma è quella di poter salvare un qualsiasi codice a barre di tipo EAN-13 che venga scannerrizato in maniera permanente se l'utente ritiene che possa tornagli utile nel futuro.

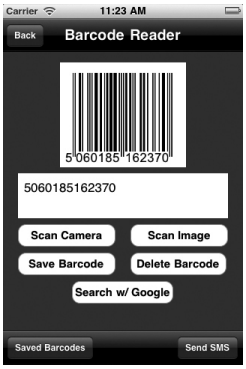

Figura 11.2: Vista delle informazioni dell'utente

Questa funzione dell'applicazione viene abilitata solo nel caso lo scanner abbia acquisito un codice di tipo compatibile (al momento solo gli EAN-13 sono supportati) quindi verrà richiesto successivamente di fornire una breve descrizione, del tutto facoltativa, che possa ricordare il codice a barre salvato. Una volta fatto ciò sarà possibile reperire il codice tramite un apposito elenco che visualizza lo storico di tutti i codici a barre salvati dall'utente attualmente loggato e dall'utente di default, in questo modo viene fornita la possibilità di rendere disponibile a tutti gli utenti un determinato codice a barre andandolo a salvare quando si è loggiati come utente di default.
I codici a barre registrati verranno quindi visualizzati su una semplice tabella nella quale sarà possibile vedere una piccola anteprima del codice, il codice stesso e la descrizione che gli abbiamo associato. Tutte le righe della tabella sono cliccabili e se selezionate verrà riaperta la schermata precedente da dove è possibile svolgere alcune azione sul codice selezionato, come ad esempio eliminare il codice dalla lista o inviare l'SMS con le informazioni del codice evidenziato.
Le funzioni di salvataggio ed eliminazione di un determinato codice sono sempre attive, fintanto che la condizione di codice EAN-13 viene soddisfatta, ma vengono comunque effettuati controlli per cui non sarà possibile eliminare un codice che non è stato precedentemente salvato dall'utente o salvare un codice già presente nel database.

11.1.3 Composizione Automatica degli SMS

Questa funzione di fatto ci consentirà di inviare le informazioni necessarie una volta completato il processo di acquisizione delle informazioni e quindi effettuare il pagamento del servizio che abbiamo richiesto.
L'idea iniziale era di comporre il messaggio e quindi inviarlo direttamente, questo però non è consentito dalle applicazioni Apple che per motivi di sicurezza non consento l'invio automatico degli SMS senza una conferma esplicita da parte dell'utente, dobbiamo quindi comporre il messaggio in maniera questa si automatica, visualizzarlo all'utente, quindi renderlo modificabile e se l'utente lo riterrà opportuno, inviarlo.

11.1.4 Lettura della Posizione Tramite GPS

La funzione GPS la analizziamo per ultima perché in parte opzionale, nel senso che di fatto è implementata nell'applicazione, ma nel caso in cui la ricezione del segnale non sia ottima o per altri motivi ci siano problemi a reperire le informazioni tramite questo strumento si potrà comunque selezionare la propria posizione in modo manuale attraverso un menù a scelta multipla.
Questa funzione è necessaria perché nella fase precedente alla composizione dell'SMS di conferma, la conoscenza della posizione attuale ci sarà utile per

ottenere le informazioni da un web-server per comporre in maniera appropriata la richiesta di pagamento che effettueremo tramite il messaggio che andremo ad inviare alla fine.

11.2 Struttura dell'Applicazione

L'applicazione si apre subito su una pagina di ingresso dove viene visualizzato il titolo della pagina e in basso viene visualizzato un semplice menù di navigazione composto da due pulsanti che portano alle due sezioni principali dell'applicazione, la parte che gestisce gli utenti e la parte di acquisizione, salvataggio e utilizzo dei codici a barre.

11.2.1 Gestione del Profilo Utente

La parte che gestisce i profili degli utenti è composta da un totale di tre facciate che hanno in comune il menù che si trova in basso nella pagina e dalla quale è possibile accedere a tutte le pagine e la barra del titolo nella quale è presente un pulsante che permette di ritornare alla pagina principale. Le tre pagine svolgono funzioni diverse, quali:

- visualizzazione delle informazioni sull'utente attualmente connesso, e possibile eliminazione del profilo corrente;
- creazione del profilo di un utente;
- login di un utente già presente all'interno del database, attraverso username e password.

Ai fini dell'utilizzo dell'applicazione, la creazione e gestione dell'utenza è ininfluente, infatti è comunque sempre presente un utente di default che non può essere eliminato dal database e che viene automaticamente loggiato nel momento in cui vado ad eliminare l'utente attualmente loggato nell'applicazione.

11.2.2 Gestione del Codice a Barre

Arriviamo quindi alla parte più corposa della nostra applicazione, attraverso la prima pagina potremo svolgere praticamente la maggior parte delle funzioni di cui abbiamo bisogno. Sarà proprio da qui che potremo acquisire i codici a barre o QR sia tramite la fotocamera sia tramite delle immagini già salvate all'interno del nostro cellulare.
Potremo anche, una volta acquisita un'immagine, salvarla all'interno del database attraverso l'apposito pulsante presente nella pagina principale di questa sezione.
Una volta salvato il nostro codice a barre possiamo vederlo all'interno della

tabella che riassume tutti i codici salvati nel tempo e ricaricarlo nella pagina principale come se fosse stato appena acquisito, e a questo punto decidere se avanzare con l'esecuzione dell'applicazione che porterà alla realizzazione di un SMS che andrà a gestire un determinato servizio o eliminare il codice evidenziato.

Capitolo 12

Applicazione Pagamento Biglietti per TPL

Andiamo ora a prendere in considerazione lo sviluppo di una applicazione che permette all'utente di vedere quali sono i servizi di trasporto pubblico presenti in zona e quindi poter comprare tramite l'invio di un SMS uno o più biglietti o rinnovare il proprio abbonamento nel caso in cui l'utente abbia già la tessera.
Dovremo quindi realizzare diverse facciate che permettano di svolgere queste diverse funzioni per poi ritornare le informazioni necessarie in una pagina di riepilogo dalla quale sarà poi possibile effettuare il pagamento tramite l'invio dell'SMS.
Bisognerà inoltre implementare la parte di server che permette di recuperare le informazioni sulle compagnie di trasporto pubblico disponibili attraverso l'implementazione di un database e di alcune pagine internet dinamiche che restituiscano le informazioni richieste all'applicazione.
Andiamo quindi a vedere quali saranno le funzioni e le pagine principali che andremo ad implementare in questa applicazione.

12.1 Pagina Informativa

La prima pagina che ci verrà presentata nell'applicazione sarà una pagina informativa dove troveremo informazioni sulle modalità di pagamento ed altre funzioni che l'applicazione svolge, questa pagina presenta un'interruttore in modo che se non volessimo più visualizzare questa pagina all'avvio dell'applicazione basterà settare questo pulsante su "off". Che si scelga o meno la possibilità di disabilitare questa pagina, comunque l'unica opzione possibile a questo punto sarà quella di avanzare alla pagina successiva che ci introduce nell'applicazione vera e propria.

12.2 Selezione della Provincia e dell'Azienda TPL

Sebbene sia di fatto la prima pagina della nostra applicazione è proprio qui che una delle azioni più importanti ha luogo. Infatti è in questo punto che la nostra applicazione si connetterà ad un server esterno per recuperare tutte le informazioni necessarie per lo svolgimento corretto del programma. Le informazioni verranno recuperate dal server subito nel momento in cui carichiamo la pagina, rendendo quindi tutte le informazioni subito disponibili possiamo subito decidere in quale provincia siamo e con che compagnia di trasporto pubblico vogliamo viaggiare.
La selezione della provincia può essere fatta in due modi, o tramite l'utilizzo del GPS che ci fornirà la più vicina città riconosciuta o potremo sempre decidere di selezionare manualmente la città/provincia tramite una comoda "rotella" nella quale saranno presenti tutte le opzioni sicuramente disponibili.
Fatta la scelta per quanto riguarda la nostra posizione, ci verrà data la possibilità quindi di scegliere anche la compagnia con la quale viaggiare.
Selezionate entrambe le scelte possiamo quindi passare alla vista successiva che ci darà la possibilità di scegliere di quale servizio usufruire.

12.3 Selezione del Tipo di Servizio

Questa vista è molto semplice e presenta solo tre pulsanti dai quali si raggiunge altrettante pagine dalle quali si andrà a finalizzare il tipo di servizio richiesto. I bottoni visualizzati in questa pagina saranno utilizzabili (cliccabili) solo nel caso in cui il servizio sia effettivamente erogato dalla compagnia selezionata.
Le tre opzioni disponibili sono, la possibilità di comprare un biglietto singolo, più biglietti di diverso tipo o rinnovare un abbonamento.

12.3.1 Acquisto Biglietto Singolo

Da questa pagina sarà possibile selezionare un singolo biglietto del servizio di trasporto pubblico precedentemente selezionato. La selezione viene resa possibile grazie all'utilizzo di una semplice tabella, una volta selezionata una voce di quest'ultima sarà possibile proseguire nell'applicazione arrivando alla schermata di riepilogo per il pagamento.

12.3.2 Acquisto di Biglietti Multipli

Questa pagina fornisce la possibilità di comprare più biglietti di tipo diverso, l'implementazione sarà quindi leggermente più complessa rispetto a quella per la selezione del biglietto singolo. In questa vista infatti è presente una

tabella con righe modificate in modo che possano venire visualizzate le informazioni del biglietto e il numero di biglietti che si intende comprare.

Figura 12.1: Vista acquista biglietto multiplo

La funzione di incremento o decremento dei biglietti viene elegantemente svolta da due pulsanti che si trovano alla destra della tabella e che operano sulla riga della tabella attualmente selezionata.
Mano a mano che si effettuano modifiche viene inoltre aggiornato il costo totale in modo da avere un'anteprima del costo finale.

12.3.3 Acquisto di un Abbonamento

La pagina per l'acquisto di un abbonamento si presenta con quattro campi di testo a compilazione obbligatoria. Questi campi servono per impostare il tipo di abbonamento che si intende acquistare e per quale periodo.
Il campo della data se impostato con la data odierna fa si che l'abbonamento venga impostato per essere utilizzato all'interno di quella settimana, mese, semestre o anno, in base al periodo di tempo che si era impostato.
Una volta impostati tutti i campi si può avanzare alla pagina successiva dalla quale sarà possibile anche vedere il prezzo dell'abbonamento.
In questa pagina si è fatto largo uso degli oggetti UIPicker a scomparsa vista la quantità dei campi da riempire tramite la modalità a selezione.

12.4 Pagina di Riepilogo

Siamo quindi arrivati alla pagina di riepilogo, questa visualizzuerà in maniera chiara il numero e il tipo di biglietti che abbiamo scelto precedentemente.
Il riepilogo viene visualizzato su un oggetto di tipo UITableView con i dati visualizzati standardizzati nelle vista dalla quale siamo giunti.

Figura 12.2: Vista pagina di riepilogo pagamento

La standardizzazione è uguale per tutte le 3 pagine precedenti, in questo modo ci viene semplificato molto lavoro in questa parte finale accedendo semplicemente ad un numero limitato di array di dati.

Capitolo 13

Sviluppi Futuri

Entrambe le applicazioni viste finora di fatto funzionano utilizzando dei dati statici ovvero inseriti dal programmatore nel momento in cui ha scritto il codice e quindi non modificabili nel tempo. Se volessimo fare in modo che questi dati siano dinamici e quindi che possano cambiare nel tempo dovremmo recuperare queste informazioni dall'esterno facendo richiesta, ad esempio, a pagine internet dinamiche.
Il prossimo passo da fare sarebbe stato quello di implementare un server web composto da un database tipo *MySQL* collegato a delle pagine *php*[1] che potessero gestire le richieste provenienti dalle applicazioni che abbiamo creato in precedenza, restituendo in particolare dei file di tipo *xml*[2] contenenti le informazioni necessarie per le nostre applicazioni.
Andiamo quindi a vedere quale sarebbe stata la realizzazione di una configurazione tipo per un server web, in questo caso la realizzazione del database viene tralasciata perché di fatto molto simile a quella già vista precedentemente per gestire il salvataggio dei codici a barre, in questo caso l'unica differenza deriva dal fatto che non andremo più ad utilizzare SQLite ma MySQL che quindi hanno dei tipi di dati leggermente differenti fra di loro ma il linguaggio per creare ad esempio una tabella, rimane invariato.
Andremo invece ad analizzare più in dettaglio i comandi che renderanno possibile il collegamento con il database tramite le pagine php e come vengono letti i file *xml* in iPhone.

13.1 Funzioni principali di PHP

13.1.1 Connessione a MySQL

Perché un'applicazione realizzata in *PHP* possa utilizzare le informazioni contenute all'interno di una base di dati dobbiamo come prima cosa fare in

[1] Hypertext Preprocessor o processore di ipetesti
[2] eXtensible Markup Language

modo che questa comunichi con l'RDBMS che gestisce il database, questo è possibile implementando una procedura iniziale chiamata connessione; questa connessione avviene tra lo script php e il programma che gestisce la base di dati e non tra lo script e la base di dati stessa.
Per aprire una connessione in PHP si utilizza una funzione che è nativa nel linguaggio chiamata `mysql_connect()` , essa restituisce un identificativo di connessione MySQL in caso di successo, altrimenti restituisce *FALSE*.
La funzione *connect* ha inoltre bisogno del passaggio di tre parametri:

- **hostname**: è il nome o indirizzo IP della macchina nella quale risiede il databse manager MySQL, nel caso di un'installazione locale l'hostname è generalmente chiamato "localhost";
- **username**: è il nome di uno degli utenti abilitati alla manipolazione di uno o più database;
- **password**: per questioni di sicurezza ad ogni utente che può utilizzare un database è bene assegnare una password, questo parametro permette la corretta autenticazione al momento della connessione con l'RDBMS;

Come in tutti i linguaggi sarà possibile esprimere i parametri sia sotto forma di variabile sia che come valori puri, l'ordine da rispettare è quello visto nell'elenco precedente, per cui potremmo usare questa forma:

```
$nomehost = ''localhost'';
$nomeutente = ''username'';
$password = ''password'';

// connessione all'RDBMS
$connessione = mysql_connect($nomehost, $nomeutente, $password);
```

oppure la seguente:

```
$connetti = mysql_connect(''localhost'', ''username'', ''password'');
```

13.1.2 Chiusura di una Connessione a MySQL

Nella sezione precedente abbiamo visto come si può instaurare una connessione tra lo script PHP e il database manager, ora quindi andremo a veder come effettuare l'operazione opposta.
Come per il comando precedente anche questo è presente in maniera nativa nel codice denominata `mysql_close()`, questo restituisce *TRUE* nel caso in

cui la chiusura della connessione abbia avuto successo, mentre restituisce *FALSE* quando si verifica un errore o un malfunzionamento, inoltre se non viene specificato alcun parametro la funzione cercherà di chiudere l'ultima connessione aperta.
La chiusura di una connessione è una procedura estremamente importante perché consente di liberare risorse utili per il sistema, generalmente la connessione viene chiusa automaticamente quando lo script termina la sua esecuzione, ma è comunque buona abitudine utilizzare `mysql_close()` per evitare possibili problemi o inutili sprechi di risorse.

13.1.3 Selezione del Database

Una volta connessi con il database manager non manca altro che accedere al database al quale vogliamo porre le nostre interrogazioni.
Per poter quindi lavorare sulla nostra base di dati dovremo procedere con un'operazione denominata "selezione del database", questa procedura è possibile grazie ad un'apposita istruzione nativa denominata `mysql_select_db()` che ha come parametri il nome del database al quale vogliamo accedere e l'identificativo della connessione corrente aperta tramite l'invocazione della funzione `mysql_connect`.
Il tutto si traduce nel seguente codice:

```
$connessione = mysql_connect(‘‘hostanme’’, ‘‘username’’, ‘‘password’’);

$nome_db = ‘‘nomedatabase’’;
$selezione = mysql_select_db($nome_db, $connessione);
```

Volendo essere del tutto pignoli in entrambe le istruzioni, di connessione e di selezione, si potrebbe aggiungere in coda il comando `mysql_error()`, questo permette di ricevere eventuali notifiche nel caso dovessero verificarsi errori o malfunzionamenti. Una istruzione tipo si traduce nel seguente modo:

```
$connessione = mysql_connect(‘‘host’’, ‘‘user’’, ‘‘pwd’’)
                                                or die (mysql_error());
```

13.1.4 Esecuzione di una Query

Connessi al database manager e selezionata la base di dati nella quale dobbiamo lavorare possiamo iniziare a eseguire le nostre interrogazioni attraverso l'utilizzo della funzione `mysql_query()` che ha per parametro la stringa che contiene la query vera e propria.
Come anche per le altre funzioni si può aggiungere in coda all'istruzione la chiamata a `mysql_error()` in modo che se qualcosa andasse storto avremo una segnalazione da parte dell'applicazione.
Vediamo quindi un semplice esempio su come utilizzare questo comando:

```
$query_sql = ‘‘SELECT * FROM nome_tabella_1’’;
$sql1 = mysql_query($query_sql) or die (mysql_error());

$sql2 = mysql_query(‘‘SELECT * FROM nome_tabella_2’’)
                                                 or die (mysql_error());
```

Il risultato della query se eseguita correttamente verrà inserito all'interno della variabile `$sqlx` e pronto per essere utilizzato da altre funzione che svolgono il compito di leggere il dato contenuto all'interno di questa variabile e quindi elaborarlo in vari modi.
Senza entrare troppo in dettaglio vediamo quali sono i possibili modi per elaborare una query di selezione:

- mysql_fetch_row();
- mysql_fetch_array();
- mysql_fetch_assoc();
- mysql_fetch_object();

il più intuitivo da utilizzare è sicuramente `mysql_fetch_row()` perché come ricorda anche il nome restituisce l'array di una riga caricata e ciascuna colonna del risultato viene caricata all'interno di un indice dell'array, a partire dall'indice "0", rendendo quindi estremamente semplice il recupero dei dati di cui abbiamo bisogno.
Per tutti gli altri tipi di interrogazione, come ad esempio un inserimento o una cancellazione di un record, non c'è bisogno che il risultato venga salvato all'interno di una variabile perché la query semplicemente può essere eseguita con successo o meno e nel caso in cui volessimo gestire il malfunzionamento basterà, come già visto più volte, aggiungere in coda il comando `mysql_error()`.

13.2 Gestione dei file XML in iPhone

13.2.1 Instroduzione a XML

XML è un metalinguaggio, in quanto non è un linguaggio di programmazione vero e proprio, di markup, ovvero un linguaggio che consente di estendere o controllare il comportamento di altri linguaggi.
XML è l'acronimo di *eXtensible Markup Language*, da cui si capisce che la sua caratteristica principale è quella di creare **tag** personalizzati, in base alle nostre esigenze.
Per integrare XML nelle nostre applicazioni avremo bisogno di un oggetto che si occupa di recuperare i dati dal file XML e cioè un **parser**.
Esistono due principali tipi di interfaccie, presenti ed implementate nella maggior parte dei linguaggi di programmazione:

- SAX[3];
- DOM[4].

[3]Simple API for XML
[4]Document Object Model

Cocoa Touch mette a disposizione una classe NSXMLParser che permetterà di analizzare XML utilizando l'interfaccia SAX. Sempre in Cocoa esiste anche una classe NSXMLDocument che permette di utilizzare l'interfaccia DOM ma non questa non è stata inclusa nelle librerie per iPhone.

13.2.2 Implementazione di NSXMLParser

L'interfaccia SAX lavora leggendo il file XML linea per linea ed è basata su eventi. Durante la lettura al verificarsi di un evento come ad esempio l'inizio di un tag o la sua chiusara, viene chiamata una funzione specifica che può gestirlo.
I vantaggi dell'interfaccia SAX sono principalmente legati alla velocità e all'utilizzo della memoria. Con questa interfaccia non è possibile creare file XML ma solo leggere e processare file XML già esistenti.

Ci sono tre metodi che vanno implementati all'interno della classe che dovrà leggere il file XML:

```
-(void)parser:(NSXMLParser *)parser didStartElement:(MSString *)elementName
    namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
    attributes:(NSDictionary *)attributeDict

-(void)parser:(NSXMLParser *)parser didEndElement:(MSString *)elementName
    namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName

-(void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
```

Come suggerito dal loro nome **didStartElement** viene chiamato quando si incontra l'inizio di un tag, **didEndElement** alla fine di un tag e **foundCharacters** serve per poter ottenere il dato contenuto all'interno di un tag. Cominciamo dal metodo *didStartElement*, questo viene chiamato ogni volta che il parser incontra un nuovo tag e il puntatore elementName ci darà il nome dell'elemento. Prendiamo ad esempio il seguente codice:

```
<Nominativo Nome=‘‘Luca’’ Cognome=‘‘Orietti’’ />
```

questo contiene due attributi, il codice all'interno di didStartElement per poter prendere il valore dei due attributi sarà il seguente:

```
if([elementName isEqualtoString:‘‘Nominativo’’]) {
    NSString *nome = [attributeDict objectForKey:‘‘Nome’’];
    NSString *cognome = [attributeDict objectForKey:‘‘Cognome’’];
}
```

Come possiamo vedere *attributeDict* è un dizionario che contiene la lista di tutti gli attributi, e con il metodo *objectForKey:key* possiamo attingere al valore dell'attributo con chiave "key". Quindi quello che faremo all'interno di *didStartElement* è solitamente una serie di *if* in cascata dove, una volta riconosciuto il tag dato il suo nome possiamo procedere alla memorizzazione dei dati.

Il metodo *didEndElement* viene invece utilizzato in maniera simmetrica quando arriva un evento di tag chiuso. Utilizzato assieme a didStartElement ci permette di capire quando abbiamo finito di memorizzare i nostri dati e possiamo quindi salvarli in una nostra struttura dati.

Il metodo *foundCharacters* viene chiamato anche più di una volta all'interno dello stesso tag, conviene quindi utilizzare una struttura dati diversa dal dizionario per poter memorizzare correttamente tutto il contenuto presente all'interno dell'elemento.

Capitolo 14

Conclusioni

Ricapitolando il lavoro svolto, abbiamo visto come è stata svolta la realizzazione di due applicazioni per iPhone andando ad analizzare come vengono creati i collegamenti fra le varie pagine ed il passaggio di informazioni fra di esse attraverso due metodi principali, il primo con il salvataggio di variabiali di sessioni interne al programma, mentre l'altro attraverso l'utilizzo di un database.
Siamo quindi passati alla realizzazione della nostra prima applicazione nella quale ci veniva chiesto di implementare un modo per leggere un certo tipo di codice a barre e possibilmente salvarlo in modo da poterlo successivamente riutilizzare.
Abbiamo quindi utilizzato una libreria già esistente per la lettura dei codici a barre e l'abbiamo implementata all'interno della nostra applicazione in modo che ci restirtuisse quella che è la traduzione numerica del codice a barre. Da questo siamo quindi riusciti, attraverso una semplice procedura, a ricreare il codice appena letto in modo da renderlo più chiaro e leggibile nel momento del salvattaggio all'interno del database.
Per quanto riguarda la seconda applicazione abbiamo creato una procedura di inserimento dati che ci permette di poter scegliere quale tipo di servizio vogliamo utilizzare e attraverso l'invio di un SMS finale permettere il pagamento di tale servizio.
L'applicazione al momento riceve informazioni solo attraverso la scelta di voci statiche preimpostate all'interno dell'applicazione e non modificabili esternamente. Il successivo passo sarebbe infatti stato quello di ricevere le informazioni attraverso il collegamento ad un server esterno e in continuo aggiornamento, questo non è stato possibile realizzarlo in quanto le risorse non ci sono state potute fornire a causa di problemi da noi non gestibili.
Per concludere quindi attraverso lo sviluppo di questa tesi abbiamo appreso un nuovo linguaggio di programmazione e l'utilizzo di un nuovo sistema operativo per poter realizzare la base per le applicazioni appena citate che ovviamente rimangono da rifinire e ampliare secondo quanto sopra riportato.

Bibliografia

[1] Wikipedia Italia: Codice a barre,
http://it.wikipedia.org/w/index.php?title=Codice_a_barre&oldid=51446199

[2] Wikipedia: Barcode,
http://en.wikipedia.org/w/index.php?title=Barcode&oldid=515085975

[3] Wikipedia Italia: Codice QR,
http://it.wikipedia.org/w/index.php?title=Codice_QR&oldid=52801870

[4] Wikipedia: QR code,
http://en.wikipedia.org/w/index.php?title=QR_code&oldid=515479345

[5] Wikipedia Italia: Xcode,
http://it.wikipedia.org/w/index.php?title=Xcode&oldid=51970297

[6] Wikipedia: Xcode,
http://en.wikipedia.org/w/index.php?title=Xcode&oldid=515241109

[7] Wikipedia Italia: Interface Builder,
http://it.wikipedia.org/w/index.php?title=Interface_Builder&oldid=50779435

[8] Wikipedia: Interface Builder,
http://en.wikipedia.org/w/index.php?title=Interface_Builder&oldid=508932504

[9] StackOverFlow,
http://stackoverflow.com/

[10] DevApp,
http://www.devapp.it/

[11] ZBar: Barcode Reader,
http://zbar.sourceforge.net/