Università degli Studi di Modena e Reggio Emilia

Dipartimento di Scienze Fisiche,

Informatiche e Matematiche

Corso di Laurea in Informatica

Progetto e Sviluppo di un'Applicazione Mobile per il Calcolo dei Percorsi con Mezzi Pubblici ed Ecologici

Relatore:

Prof. Riccardo Martoglia

Candidato:

Andrea Zuccarini

A. A. 2014 / 2015

Ringraziamenti:

Ringrazio l'Ing. Riccardo Martoglia per tutta la disponibilità e competenza dimostrata durante la realizzazione del progetto e la scrittura della tesi.
Un ringraziamento particolare alla mia morosa che mi è stata sempre vicina in questi mesi di lavoro sopportandomi e rincuorandomi nei momenti peggiori, non ce l'avrei mai fatta senza di lei.
Ringrazio anche tutti i miei amici per i tanti bei momenti di svago passati insieme.
Infine ringrazio calorosamente i miei genitori, mia sorella e i miei nonni che mi hanno sempre aiutato in tutto e non mi hanno mai fatto mancare niente dalla vita.

Indice

Principali caratteristiche del linguaggio Swift e novità introdotte in Swift 2 5
1.1 Cenni Storici 5
1.2 Principali caratteristiche del linguaggio 6
1.2.1 Tipi di dato e valori opzionali 7
1.2.2 Controllo di flusso e Funzioni 9
1.2.3 Classi e Strutture 12
1.2.4 Gestione della Memoria 14
1.3 Novità e cambiamenti introdotti in Swift 2 16
1.3.1 Estensioni dei protocolli 16
1.3.2 Gestione degli errori 18
1.3.3 Controllo versione iOS 19
Framework e Servizi per la Mobilità 20
2.1 Ambiente Xcode 20
2.2 Map Kit e Google Maps iOS sdk 22
2.2.1 Framework Map Kit 22
2.2.2 Google Maps iOS sdk 24
2.2.3 Differenze tra i due servizi 27
2.3 Servizi Google sulla mobilità 27
2.3.1 Directions API 28
2.3.2 Geocoding API 29
2.4 Core Location 30
2.5 Core Data 31
Progettazione 35
3.1 Scelte implementative 37
3.2 Recupero e gestione dei dati 38
3.2.1 Creazione Modello Core Data 38
3.2.2 Importazione colonnine e biciclette 41
3.2.3 Importazione Stradario 43

3.3 Scelte grafiche 44

3.3.1 Map View Controller 45

3.3.2 Ricerca Indirizzo View Controller 47

3.3.3 Itinerario View Controller 48

Utilizzo reale dell'applicazione 50

4.1 Modalità trasporti pubblici 51

4.2 Modalità veicolo elettrico 56

Conclusione e Sviluppi Futuri 63

Indice delle figure

Figura 1.1: Tipi di dato e valori opzionali 9

Figura 1.2: Varie tipologie di funzioni 11

Figura 1.3: Classi e strutture 13

Figura 1.4: Gestione della memoria e weak reference 15

Figura 1.5: Estensione dei protocolli 17

Figura 1.6: Gestione degli errori 18

Figura 1.7: Controllo versione 19

Figura 2.1: Creazione di una mappa e di un'annotazione utilizzando Map Kit 24

Figura 2.2: Contenuto di un podfile 25

Figura 2.3: Utilizzo sdk Google maps per la creazione di una mappa 26

Figura 2.4: Modello di gestione di Core Data 32

Figura 3.1: Differenti visualizzazioni del tool per gestire i modelli 39

Figura 3.2: Creazione classe Fermata 39

Figura 3.3: Fermate degli autobus rappresentate sulla mappa 40

Figura 3.4: Visualizzazione colonnine di ricarica per veicoli elettrici 42

Figura 3.5: Visualizzazione postazione biciclette 43

Figura 3.6: Esempio di ricerca indirizzo 44

Figura 3.7: Storyboard completo 45

Figura 3.8:Rappresentazione delle due modalità nel view controller principale 46

Figura 3.9: Table view utilizzata per impostare la partenza e la destinazione 46

Figura 3.10: Ricerca indirizzo 48

Figura 3.11: Dettagli sugli itinerari 49

Figura 4.1**:** Impostazione partenza 51

Figura 4.2: Selezione dell'indirizzo di destinazione 52

Figura 4.3: Visualizzazione della partenza e destinazione 53

Figura 4.4: Ricerca dell'itinerario 54

Figura 4.5: Visualizzazione dettagli dell'itinerario 55

Figura 4.6: Ultimo step dell'itinerario 56

Figura 4.7: Alert Controller 57

Figura 4.8: Visualizzazione colonnine di ricarica per veicolo elettrico 58

Figura 4.9: Impostazione partenza e destinazione 59

Figura 4.10: Ricalcolo percorso causa autonomia insufficiente 60

Figura 4.11: Dettagli del percorso 61

Figura 4.12: Postazione di ricarica per veicoli elettrici 62

Figura 4.13: Fine dell'itinerario 62

Introduzione

Fra i vari problemi che interessano la popolazione che vive nelle grandi metropoli vi è la necessità di utilizzare i mezzi di trasporto in modo efficiente, al fine di spostarsi rapidamente e, possibilmente, in maniera ecologica. E' negli ultimi anni che il settore informatico si sta dedicando alla creazione di servizi specifici nel campo dei trasporti. I primi che furono realizzati erano progettati per funzionare in ambienti desktop o per essere usufruibili attraverso il web, oggi invece, grazie all'utilizzo sempre maggiore di dispositivi portatili come smartphone e tablet, questi servizi vengono rilasciati sotto forma di applicazioni per i vari sistemi operativi mobile. iOS, distribuito da Apple, è utilizzabile solamente nei dispositivi proprietari fabbricati dalla casa di Cupertino mentre Android, il sistema operativo *open source* rilasciato da Google, viene installato su svariati terminali di differenti produttori.

Esaminando le due applicazioni sviluppate per ambienti iOS maggiormente utilizzate per la gestione delle mappe e il calcolo dei percorsi, Google Maps e Maps fornita da Apple, ci si è accorti della mancanza di dati specifici dettagliati ed intuitivi riguardanti i mezzi di trasporto pubblici e i servizi di supporto per veicoli elettrici nella città di Bologna. Le mappe di Google offrono la ricerca del percorso desiderato utilizzando mezzi pubblici, ma non visualizzano in primo piano le

annotazioni riferite alle fermate degli autobus, quelle di Apple si limitano invece alla ricerca del percorso da affrontare solamente con autovetture, non supportano ricerche per utilizzare mezzi pubblici ed ecologici all'interno del percorso desiderato.

Nasce dunque l'idea di progettare un'applicazione, usufruibile all'interno dell'area urbana di Bologna, che fornisca agli utenti informazioni relative ad autobus, veicoli elettrici e biciclette. Questa applicazione è stata sviluppata per ricercare il percorso più breve dopo aver settato partenza e relativa destinazione. Sono disponibili due modalità differenti di utilizzo: una permette di giungere a destinazione utilizzando mezzi pubblici; l'altra è indirizzata agli utenti che si trovano alla guida di veicoli elettrici e consente l'inserimento dell'autonomia residua del veicolo prima della ricerca dell'itinerario, in questo modo l'applicazione è in grado di consigliare un percorso alternativo compreso di sosta per la ricarica in una delle varie colonnine nel caso in cui l'autonomia non sia sufficiente a completare il percorso impostato.

L'applicazione è stata sviluppata per funzionare in ambiente iOS, quindi utilizzabile su dispositivi Apple. Il progetto è stato realizzato in XCode, l'IDE fornito dalla casa madre, utilizzando la seconda versione del nuovo linguaggio di programmazione *Swift* per la scrittura del codice. Sono stati utilizzati differenti *framework* nativi e servizi specifici di supporto alle mappe e alla mobilità, offerti da Google, indispensabili per la realizzazione del progetto.

La presente tesi si divide in due parti principali, la prima formata da due capitoli e la seconda da tre.

Parte Prima

- **Capitolo 1: Principali caratteristiche del linguaggio Swift e novità introdotte in Swift 2**

  Il capitolo verte sulle principali caratteristiche del linguaggio utilizzato per la scrittura del codice: la definizione di variabili e costanti di un certo tipo, possibilità di valori opzionali, definizione di classi e strutture, gestione degli errori. L'ultimo paragrafo introduce le novità che ha portato Swift 2.

- **Capitolo 2: Framework e Servizi per la mobilità**

  Il capitolo introduce l'ambiente di sviluppo Xcode, parla delle mappe fornite da Apple e quelle di Google mettendo in luce le differenze. Successivamente descrive due dei servizi di supporto alle mappe offerti dalla casa di Mountain View, *Directions API* e *Geocoding API*. Infine vengono descritti due framework nativi molto importanti, *Core Data* e *Core Location*.

Parte seconda

- **Capitolo 3: Progettazione**

  Nel capitolo vengono discussi i dettagli implementativi tenuti in considerazione durante la creazione dell'applicazione, si parla del recupero e della gestione di tutti i dati da visualizzare sulla mappa, infine si analizzano le scelte grafiche adottate.

- **Capitolo 4: Utilizzo reale dell'applicazione**

  Questo capitolo presenta una prova reale dell'applicazione all'interno dell'area urbana di Bologna. Sono state testate entrambe le modalità, all'interno del capitolo sono presenti svariate figure per aiutare la comprensione delle simulazioni effettuate.

- **Capitolo 5: Conclusione**

  L'ultimo capitolo è volto ad analizzare i limiti presenti all'interno dell'applicazione e gli sviluppi implementativi futuri che potrebbero essere attuati su di essa, per migliorarne l'efficienza e la stabilità.

Parte Prima

Linguaggio ed Ambiente di Sviluppo

Capitolo 1

Principali caratteristiche del linguaggio Swift e novità introdotte in Swift 2

1.1 Cenni Storici

Nel lontano 2008 Steve Jobs introdusse un *marketplace*, chiamato App Store, nei dispositivi dotati di sistema operativo iOS. Chiunque poteva sviluppare un'applicazione utilizzando l'IDE Xcode, utilizzabile solamente in ambiente OSX, e pubblicarla sul market dopo essere stata revisionata. Per richiedere la pubblicazione serviva un'iscrizione annuale di novantanove dollari. Il linguaggio utilizzato per la programmazione era Objective-C, sempre molto quotato da Steve Jobs, supportando parti di codice scritte in linguaggio C. Dal 2008 al 2014 ci sono

stati vari aggiornamenti del sistema operativo iOS e OSX ma il linguaggio utilizzato per scrivere codice non è mutato fino alla presentazione di Swift [1] avvenuta nell'estate del 2014. Swift è un nuovo linguaggio derivato da anni di ricerca da parte di sviluppatori Apple e non. L'azienda di Cupertino ha presentato la prima versione nell'estate del 2014 durante la loro consueta conferenza (WWDC). Un anno dopo, alla stessa conferenza, è stata mostrata la seconda *major release* del linguaggio con l'introduzione di importanti novità e annunciato il rilascio open source del compilatore e delle librerie che dovrebbe avvenire negli ultimi mesi del 201*5.* Questo aprirà la possibilità di sviluppare software scritto in Swift anche in ambienti Windows e Linux ricordando che il linguaggio non è stato ideato solo per creare applicazioni utilizzabili in ambienti iOS e OSX; dai test effettuati si è potuto notare come in alcune circostanze sia più veloce nella creazione ed esecuzione di script lato server rispetto a linguaggi specifici come Python, Ruby e PHP.

1.2 Principali caratteristiche del linguaggio

Nei paragrafi successivi vengono introdotti i principali aspetti che caratterizzano il linguaggio. Il primo è dedicato ad introdurre i tipi di dato che possono essere assegnati alle espressioni e il concetto di tipo opzionale. Nel secondo paragrafo sono elencati i controlli di flusso supportati dal linguaggio e definito il concetto di funzione. Il terzo elenca gli aspetti comuni e differenze tra classi e strutture. L'ultimo è dedicato alla gestione della memoria con l'utilizzo del meccanismo di *automatic reference counting* (ARC) da parte del linguaggio. Alla fine di ogni paragrafo sarà presente un frammento di codice per chiarire i concetti introdotti.

1.2.1 Tipi di dato e valori opzionali

Uno degli aspetti fondamentali che caratterizza qualsiasi linguaggio di programmazione, che sia statico o dinamico, è la dichiarazione di variabili o costanti alle quali viene assegnato un tipo di dato specifico a seconda del comportamento che devono assumere all'interno del codice. Linguaggi come il C o il C++ sono di tipo *unsafe*, non fanno controlli sui tipi di dato assegnati alle variabili o costanti a tempo di compilazione. Swift come Java è un linguaggio *type safe,* esegue controlli sui tipi favorendo una programmazione chiara sulla loro assegnazione. In Swift a tempo di compilazione si esegue un controllo sui dati detto *type checking,* per scovare eventuali errori sul tipo che gli è stato assegnato. Se durante la dichiarazione di un'espressione a questa non viene assegnato esplicitamente un tipo, il compilatore utilizza un meccanismo detto *type inference* per attribuirgli uno specifico tipo di dato in base al valore (*literal value*) che gli è stato assegnato.

Swift supporta i tipi e le strutture dati dei principali linguaggi di programmazione ad oggetti. Introduce le *tuple* non presenti in Objective-C come nuova struttura dati, dove possono venire raggruppati al suo interno valori di tipi differenti. Le tuple si utilizzano spesso per raggruppare valori correlati tra loro da passare attraverso il codice. Per memorizzare strutture dati più complesse e durature all'interno del software è però preferibile utilizzare *classi* o *strutture.*

Per supportare la possibilità di non contenere nessun valore, o meglio la possibilità di contenere l'assenza del valore da parte di una variabile o costante in un determinato momento nell'esecuzione del codice, viene utilizzato un approccio differente da Objective-C. Il concetto di valore opzionale non esiste in Objective-C dove solamente gli oggetti possono contenere un valore nullo (*nil*) per indicare l'assenza di un oggetto valido, invece per le strutture e gli operatori base del *C* viene utilizzato un valore di tipo *NSNotFound* per indicare l'assenza di un valore valido al loro interno. Nel nuovo linguaggio di Apple si può dichiarare qualsiasi oggetto di qualsiasi tipo come opzionale, cioè che potrebbe avere al suo interno un valore del tipo

assegnatosi in fase di inizializzazione oppure potrebbe anche non contenere nessun valore (nil). Solamente gli oggetti dichiarati opzionali possono contenere al suo interno il valore nil senza sollevare un errore da parte del compilatore.

In Objective-C il valore nil indica un puntatore a un oggetto non esistente invece in Swift non è un puntatore ma rappresenta l'assenza di un valore del tipo dell'oggetto che lo contiene. Ogni volta che si lavora con delle variabili opzionali, è buona norma controllarne sempre il contenuto prima di utilizzarle e dopo aver verificato che la variabile risulti non priva di valore la si assegna ad una costante o variabile temporale appositamente creata per utilizzarla. Questo meccanismo prende il nome di *optional binding*. Potrebbe capitare, analizzando il contesto del codice, di comprendere che alcune variabili opzionali avranno sempre un valore al loro interno dopo la loro inizializzazione. In questo caso si può fare a meno di controllare il loro contenuto creando un optional binding e utilizzarle come se fossero delle classiche variabili (*implicity unwrapped optionals*). Sono inizializzate come variabili opzionali ma assumono le caratteristiche di quelle non opzionali, il compilatore ritornerebbe un errore se trovasse al loro interno il valore nil.

Nella figura 1.1 viene riportato un frammento di codice per chiarire i concetti spiegati in questo paragrafo sui tipi di dato e tipi opzionali. Viene mostrato il *forced unwrapping* dopo aver controllato il contenuto della variabile oppure a tempo di inizializzazione.

```
1  /* La variabile "str" è definita senza assegnarle un tipo
2     specifico. Durante la compilazione, tramite il
3     meccanismo di type checking, il type inference abilita
4     il compilatore a dedurre il tipo da assegnare alla
5     variabile andandando ad analizzare il valore assegnato.*/
6     var str = "Hello, playground"
7
8   /* Definizione di una variabile non opzionale di tipo
9      String, in questo caso non entra in gioco il type
10     inference ma il type checking verifica solo che il
11     valore assegnato sia compatibile col tipo. */
12     var strNonOpzionale:String = "Stringa non opzionale"
13
14  /* Definizione di una variabile opzionale di tipo String.
15     Si inserisce il ? alla fine della dichiarazione del
16     tipo, per qualsiasi tipo. La variabile può contenere
17     qualsiasi dato di tipo String oppure il valore nil ma
18     non può contenere nessun tipo di altro dato. */
19     var strOpzionale:String?="Stringa opzionale"
20
21  /* Ogni volta che bisogna utilizzare una variabile
22    opzionale è consigliabile controllare prima il suo
23    contenuto. */
24
25  if strOpzionale != nil {
26   print(strOpzionale!) //forced unwrapping
27   print(strOpzionale) //stampa Optional("Stringa opzionale")
28     } else {
29        print("La stringa contiene il valore nil") }
30  let strNumber="123456" //Costante di tipo Int
31
32  if let optionaBinding=Int(strNumber) { //optional binding
33   print(optionaBinding) }
34
35  // Forced unwrapping a tempo di inizializzazione
36   var strImplicityUnwrap:String!="Implicity unwrapped opt"
```

Figura 1.1: Tipi di dato e valori opzionali

1.2.2 Controllo di flusso e Funzioni

Swift eredita i più importanti metodi per il controllo di flusso del codice da linguaggi come il C e l'Objective-C. Supporta la creazione di cicli *for, for-in* per

iterare il ciclo attraverso una sequenza e i cicli *while*. Una variazione del classico ciclo while si presenta col ciclo *do-while* che, a partire da Swift 2 , è stato sostituito con *repeat-while*. La parola chiave *do* con l'ultima versione del linguaggio viene utilizzata per creare blocchi di codice che raggruppano funzionalità correlate tra loro. Sono presenti i classici operatori condizionali *if* e *switch* per gestire tutti i possibili valori che può assumere un dato. Quest'ultimo comando è molto utilizzato abbinato al controllo su dati *enumerati*. Gli sviluppatori Apple stanno lavorando molto per migliorare e aumentare le funzionalità degli enumerati.

Uno degli aspetti principali che caratterizza qualsiasi linguaggio è la possibilità di dichiarare delle *funzioni*, blocchi di codice che svolgono specifici compiti, ognuna di queste viene identificata da un nome. Ogni funzione può essere definita con zero o più parametri in ingresso e valori di ritorno, viene a sua volta rappresentata da un tipo che è formato dall'insieme dei tipi dei parametri in ingresso e dei valori di ritorno. La funzione stessa, a sua volta, può essere definita come parametro in ingresso o valore di ritorno di un'altra funzione. Si può creare anche una struttura nidificata di funzioni definite una all'interno del corpo di un'altra. I valori in ingresso, salvo casi speciali, sono parametri passati per valore. Si utilizza una copia di questi e le modifiche apportate saranno visibili solamente all'interno dello *scope* della funzione. I parametri in ingresso etichettati con la parola chiave *inout* saranno passati per riferimento, le modifiche apportate all'interno della funzione saranno visibili in tutto il programma.

Nella figura 1.2 viene riportato un esempio di codice, scritto in Swift, che contiene la creazione di alcune funzioni.

```
1 var message:String="it is a message"
2 func noParametriUscita(message:String) {
3    print(message) }
4 noParametriUscita(message)
5
6  /* Definizione funzione senza parametri in ingresso e un
7     parametro in uscita di tipo String. */
8
9  func noParametriIngresso() -> String {
10      var returnMessage:String?="Messaggio di ritorno"
11 if (returnMessage != nil) { // ciclo if-else
12      return returnMessage!
13      } else {
14      return "Errore" }
15 }
16 print(noParametriIngresso())
17
18 /* Definizione funzione con due parametri in ingresso di
19    tipo Int? e nessun parametro in uscita */
20 var inizio:Int?=100
21 let fine:Int?=105
22 func duePARAMETRIDiIngresso(var inizio:Int,fine:Int)->Void{
23      for ;inizio<fine;inizio++ { //ciclo for
24          print("Posizione attuale Km " + String(inizio) +
25                "termine al Km" + String(fine))
26          print("Percorso 1 Km") }
27      print("Arrivo")
28 }
29 dueParametriDiIngresso(inizio!, fine: fine!)//forced unwrap
30
31 /* Definizione funzione che presenta due parametri in
32    ingresso,il primo è passato per riferimento,le modifiche
33    apportate a questo saranno visibili fuori dalla
34    funzione.Si aggiunge la parola chiave "inout" per
35    definire i parametri in ingresso che saranno passati per
36    riferimento. */
37 var valoreA:Double=10
38 var stringaB:String="Esterni alla funzione"
39 func passaggioParRif(inout rifA:Double,var copiaB:String) {
40      rifA++
41      copiaB="Interni alla funzione"
42      print("riferimentoA:" + String(rifA) + copiaB)
43 }
44 passaggioParRif(&valoreA, copiaB: stringaB)
45 print("rifA:"+String(valoreA)+String(stringaB)) //stampa 11
```

Figura 1.2: Varie tipologie di funzioni

1.2.3 Classi e Strutture

Con l'avvento della programmazione ad oggetti è cambiato radicalmente l'approccio alla scrittura del codice, è stato introdotto il concetto di classe utilizzata per raggruppare metodi e attributi correlati tra di loro. Swift usa un approccio simile a molti altri linguaggi per la creazione e gestione delle classi. Le strutture invece sono state ripensate per comportarsi in modo più simile alle classi. Possono entrambe definire proprietà per contenere valori, metodi per fornire funzionalità, contenere inizializzatori. Possono sfruttare entrambe le funzionalità dei protocolli dove al loro interno sono definiti metodi standard di un certo tipo. In ogni caso la classe resta ricca di più funzionalità, può sfruttate l'ereditarietà tra di esse per ereditarne le caratteristiche, la deinizializzazione permette a un'istanza di classe prima di essere eliminata di liberare tutte le risorse che possedeva e apportare delle modifiche.

Nella figura 1.3 viene presentato un esempio d'uso per illustrare la creazione e l'utilizzo di una struttura, la creazione di una classe e di una sua sottoclasse che eredita metodi e proprietà. Nella riga 1 viene definito un dato enumerato formato da due membri. Ogni volta che si definisce un enumerato, si crea un nuovo tipo di dato, in questo caso si definisce un enumerato di tipo *Orari*. Nella riga 5 viene istanziata una variabile di questo tipo ed inizializzata con uno dei possibili valori . Nella riga 7 viene creata una struttura di tipo *Giorno*, formata da due attributi e un'inizializzatore che riceve in ingresso come parametro un dato di tipo *Orari.* All'interno di questo si trova uno switch, a seconda del valore che assume la variabile definita in precedenza i due attributi della struttura assumono valori differenti. Nella riga 2.1 viene definita una classe di tipo *Veicolo* che contiene tre attributi, un inizializzatore e due metodi. L'inizializzatore ha tre parametri in ingresso, l'ultimo è di tipo *Giorno*. Nella riga 38 viene definita una sottoclasse di tipo *Cab* della classe *Veicolo*. Eredita tutti i metodi e attributi della sua superclasse, definisce un nuovo attributo e un metodo e presenta un inizializzatore che richiama quello della sua superclasse. Nelle ultime righe di codice viene definita

una costante di tipo *Veicolo*, viene cambiato il valore della variabile di tipo *Orari* e definita una variabile di tipo *Cab* .

```
1 enum Orari {
2  case Settimanale
3  case Festivo
4  }
5 var now=Orari.Settimanale
6
7 struct Giorno {
8   let inizio:String
9   let fine:String
10  init(imp:Orari) {
11     switch imp {
12        case .Festivo:
13          self.inizio="08:00"
14          self.fine="24:00"
15        case .Settimanale:
16          self.inizio="6:00"
17          self.fine="23:00" }
18   }
19 }
20
21 class Veicolo {
22   let targaVeicolo:String
23   let tariffa:Double
24   let orario:Giorno
25   init(targa:String,prezzo:Double,ore:Giorno){
26        self.targaVeicolo=targa
27        self.tariffa=prezzo
28        self.orario=ore
29   }
30   func visualizzaTariffa() {
31      print("Tariffa oraria:"+String(self.tariffa)+" euro")
32   }
33   func visualizzaOrario() {
34      print("Inizio:"+orario.inizio+"Fine:"+orario.fine)
35   }
36 }
37
38  class Cab:Veicolo {
39    var postiDispo:Int
40    init(targa:String,prezzo:Double,ore:Giorno,posti:Int) {
41       self.postiDispo=posti
42       super.init(targa:targa , prezzo: prezzo, ore:ore)
43    }
44    func stampaPostiDisponibili() {
45       print("Posti Disponibili:" + String(self.postiDispo))
46    }
47 }
48
49 let taxi=Veicolo(targa:"a1",prezzo:5,ore:Giorno(imp:now))
50 now = .Festivo
51 let cab=Cab(targa:"b2",prezzo:3,ore:Giorno(imp:now),posti:2)
```

Figura 1.3: Classi e strutture

1.2.4 Gestione della Memoria

Swift come altri linguaggi utilizza il meccanismo di automatic reference counting (ARC) per la gestione della memoria utilizzata. Ogni volta che si crea una nuova istanza di una classe, ARC alloca in memoria alcune informazioni riguardanti l'oggetto appena creato. Quando questo oggetto non sarà più utilizzato, il meccanismo in automatico libera la memoria sfruttata dalla istanza. Per ogni classe ARC mantiene un contatore degli oggetti che sono stati creati di questa, per evitare il rischio di liberare della memoria utilizzata da istanze ancora referenziate. Un serio problema si può riscontrare quando due oggetti di due classi si mantengono referenziati a vicenda innescando un ciclo in cui nessuno dei due verrà mai eliminato per liberare spazio in memoria. Questi cicli sono detti *strong reference cycles*. Il linguaggio di casa Apple per risolvere questo problema, abilita un'istanza a riferirsi all'altra senza mantenere una referenza forte, detta *strong reference,* permettendo al linguaggio di eliminare un oggetto e liberare memoria anche se questo rimane referenziato da una referenza di questo tipo. Si possono distinguere due referenze non forti, si utilizza una *weak reference* per referenziare oggetti che potrebbero assumere anche valori nil. Se l'oggetto referenziato conterrà sempre un valore è consigliabile utilizzare una *unowned reference*.

Le strutture non sfruttano l'automatic reference counting, questo perché di default vengono passate attraverso i blocchi di codice facendone una copia e non per riferimento. All'interno delle strutture non si possono definire variabili o costanti weak/unowned e non si può sfruttare la fase di deinizializzazione per apportare modifiche. Nella figura 1.4 viene presentato un esempio di strong reference cycle e un weak reference cycle.

```
1 class Persona {
2   var nome:String
3   var cognome:String
4   var macchina:Macchina?
5   weak var macchinaWeak:MacchinaWeak?
6   init(nome:String,cognome:String) {
7       self.nome=nome
8       self.cognome=cognome
9   }
10  deinit {
11    print("Deallocazione in corso")
12  }
13 }
14
15 class Macchina {
16   var targa:String
17   var proprietario:Persona?
18   init(targa:String) {
19  self.targa=targa
20  }
21  deinit {
22    print("Deallocazione in corso")
23  }
24 }
25
26 class MacchinaWeak {
27  var targa:String
28  weak var proprietario:Persona?
29  init(targa:String) {
30    self.targa=targa
31   }
32   deinit {
33     print("Deallocazione in corso")
34   }
35 }
36 var adulto:Persona?
37 var bmw:Macchina?
38 var mercedes:MacchinaWeak?
39 adulto=Persona(nome:"mario",cognome:"rossi")
40 bmw=Macchina(targa:"a2b3c4d5")
41 mercedes=MacchinaWeak(targa:"a2s3d4fk")
42
43 adulto!.macchina=bmw
44 adulto!.macchinaWeak=mercedes
45 bmw!.proprietario=adulto  //strong reference cycle
46 mercedes!.proprietario=adulto
47
48 //Strong reference cycle, memoria non liberata
49 adulto=nil
50 bmw=nil
51 //Weak reference, memoria liberata
52 mercedes=nil //viene deferenziato correttamente
```

Figura 1.4: Gestione della memoria e weak reference

1.3 Novità e cambiamenti introdotti in Swift 2

Essendo un linguaggio nato recentemente, i cambiamenti introdotti ad ogni aggiornamento sono stati sostanziali. A seguito della prima versione è stata rilasciata la release 1.2, portando cambiamenti nel modo di gestire il *casting* tra oggetti. Nei mesi a seguire si è svolto un lavoro intenso da parte degli sviluppatori per migliorare e testare il linguaggio, aiutato da un programma aperto al pubblico per testare i vari rilasci beta e inviare feedback riguardanti i problemi riscontrati. Questo duro lavoro ha portato all'uscita ufficiale della seconda versione introducendo importanti cambiamenti. Il primo paragrafo introduce un'importante novità nell'uso delle estensioni, nel secondo vengono descritti i cambiamenti apportati nel gestire gli errori all'interno del codice. L'ultimo descrive il metodo standard introdotto da Apple per verificare la versione del sistema operativo sul dispositivo che esegue il codice. Ogni paragrafo sarà accompagnato da un semplice esempio di codice per chiarire i concetti introdotti.

1.3.1 Estensioni dei protocolli

Le estensioni, parola chiave *extension,* vengono utilizzate per aggiungere nuove funzionalità a classi e strutture esistenti. La seconda versione del linguaggio introduce la possibilità di utilizzarle anche con i protocolli per estenderne le loro funzionalità. Un'altra importante novità introdotta è la possibilità di fornire implementazioni di default per i metodi definiti all'interno del protocollo. Nella figura 1.5 è mostrato un esempio di codice per chiarire l'utilizzo delle estensioni.

```
1 // Definizione protocollo "Contenitore"
2 protocol Contenitore {
3   var elementi:[String] {get set}
4   func numeroDiElementi() -> Int
5 }
6
7 /*  Classe che adotta il protocollo "Container",deve
8 implementare il metodo numeroDiElementi() */
9
10 class ScatolaAttrezzi:Contenitore {
11   var elementi:[String] = ["Colla","Forbici","Martello"]
12   func numeroDiElementi() -> Int {
13     return elementi.count
14   }
15 }
16
17 /* Swift 2 introduce la possibilità per ogni metodo
18    definito all'interno del protollo di fornire un'
19    implementazione di default utilizzando le extension. */
20
21 extension Contenitore {
22   func numeroDiElementi() -> Int {
23     return elementi.count
24   }
25 }
26 /* Classe che non deve implementare obbligatoriamente il
27    metodo numeroDiElementi(),se richiesto verrà chiamato
28    quello definito all'interno dell'estensione del
29    protocollo. */
30 class cestinoForno:Contenitore {
31   var elementi:[String]=["Stria","Comune","Pane","Pizza"]
32   }
33
34 /* Classe che implementa il metodo numeroDiElementi()
35    modificandone le funzionalità, se richiesto non verrà
36    chiamato quello di default definito all'interno
37    dell'extension ma questo definito all'interno della
38    classe. */
39 class CestinoFrutta:Contenitore {
40   var elementi:[String] = ["Arancia","Mela"]
41   func numeroDiElementi() -> Int {
42     return elementi.count*3
43   }
44 }
45
46 var contenitore:Contenitore=ScatolaAttrezzi()
47 print(contenitore.numeroDiElementi()) // Stampa 3
48 contenitore=cestinoForno()
49 print(contenitore.numeroDiElementi()) // Stampa 4
50 contenitore=CestinoFrutta()
51 print(contenitore.numeroDiElementi()) // Stampa 6
```

Figura 1.5: Estensione dei protocolli

1.3.2 Gestione degli errori

Quando si sviluppa un'applicazione, uno degli aspetti fondamentali è saper gestire tutti i possibili scenari che può assumere il software. Molte funzioni e metodi non possono garantire il loro completamento, potrebbe verificarsi un errore dipendente dal contesto durante l'esecuzione del codice al loro interno. Un caso molto comune di possibile errore è l'inoltro di una richiesta a un web server, questo potrebbe essere non raggiungibile oppure la pagina richiesta potrebbe non trovarsi più sul server. Il software potrebbe aver problemi a recuperare i dati richiesti, sempre più di frequente salvati su cloud server, i file cercati potrebbero non esistere più oppure si potrebbe non avere i diritti necessari per recuperarli. Nella figura 1.6 è messa in luce la differenza nel gestire gli errori tra Swift 1.2 e l'ultima versione rilasciata.

```
1 //Gestione degli errori in Swift 1.2,deprecata in Swift 2
2
3 let req=NSURLRequest(URL:NSURL(string:"http://www.aol.it")!)
4 var response:NSURLResponse?
5 var error:NSError?
6
7 /* Come parametro di ingresso viene passato un puntatore a
8    un oggetto di tipo NSError, che verrà settato col tipo di
9    errore in caso di anomalie nel software. */
10
11 let data=NSURLConnection.sendSynchronousRequest
12          (req,returningResponse:&response,error:&error)
13
14 // Per controllare se la richiesta ha sollevato un errore
15 if error == nil {
16    print(risposta) //Stampa risposta
17   //Gestisci risposta
18    } else {
19    // Gestisci errore
20    }
21
22 /* Gestione degli errori in Swift 2, viene introdotto il
23    modello try/throw/catch. */
24
25 let req=NSURLRequest(URL:NSURL(string:"http://www.aol.it")!)
26 var risposta:NSURLResponse?
27
28 do {
29 /* Si deve inserire il comando "try" prima della chiamata
30    del metodo per tutti i metodi che possono sollevare
31    errori.In ambiente Xcode sono tutti i metodi indicate
32    con la parola chiave "throws" */
33 let data=try NSURLConnection.sendSynchronousRequest
34              (req,returningResponse:&response)
35   print(risposta)
36   // Gestisci risposta
37   } catch {
38  // Gestisci errore
39   print(error)
40  }
```

Figura 1.6: Gestione degli errori

1.3.3 Controllo versione iOS

Ogni applicazione dovrebbe supportare il corretto funzionamento in diverse versioni del sistema operativo. Se venissero utilizzate nuove API nel codice, il software non funzionerebbe con vecchie versioni del sistema. Swift 2 introduce un supporto built-in per il controllo della versione iOS utilizzata dal dispositivo. Nel codice riportato in figura 1.7 si evidenzia il differente approccio implementativo che veniva utilizzato per controllare la versione del sistema operativo in Swift 1.2 rispetto a quello utilizzato nella versione attuale.

```
1 /* Nelle versioni precedenti a Swift 2 non veniva
2   implementato un metodo standard per controllare la
3   versione del sistema operativo (iOS) utilizzato dal
4   dispositivo. */
5
6 /* Un metodo per controllare se la classe esiste, classe
7    "NSURLQueryItem" disponibile a partire da iOS 8.0 */
8
9   if NSClassFromString("NSURLQueryItem") != nil {
10
11      // Gestione versione iOS 8.0 o superiore
12
13    } else{
14
15    // Gestione versioni precedenti ad iOS 8.0
16  }
17
18  /* Swift 2 introduce un metodo standard per il controllo
19     sulle versioni iOS. */
20
21 if #available(iOS 8,OSX 10.11,*) {
22
23  /* In ambiente iOS , questo blocco di codice verrà
24     eseguito solo dalla versione 8 dell'OS
25     In ambiente OSX , questo blocco di codice verrà
26     eseguito solo dalla versione 10.11 dell'OS */
27
28   } else {
29
30     //codice eseguito nelle versioni precedenti dell OS.
31
32   }
```

Figura 1.7: Controllo versione

Capitolo 2

Framework e Servizi per la Mobilità

2.1 Ambiente Xcode

L'uscita dell'ultima versione dell'IDE Xcode, la settima, introduce le sdk per iOS 9, OSX 10.11 e le nuove per watchOS 2, utilizzate per la creazione di applicazioni compatibili con l'Apple Watch, l'ultimo dispositivo introdotto.

Xcode 7 introduce nuove funzionalità che rendono la creazione, il testing e il debugging di un'applicazione ancora più efficiente. Molto importante è la possibilità di poter testare in modo gratuito un'applicazione in fase di creazione direttamente sul dispositivo e non solo sul simulatore all'interno dell'ambiente di

sviluppo. Con le versioni precedenti era necessaria l'iscrizione di 99 dollari all'anno per effettuare il test dell'applicazione sul device fisico, adesso il pagamento rimane obbligatorio solo per richiederne la pubblicazione sull'App Store. Per supportare gli sviluppatori è stato introdotto un servizio detto *TestFlight* che supporta fino a duemila inviti rivolti ad utenti che voglio provare l'applicazione in anteprima per rilasciare feedback sull'esperienza di utilizzo e sui bug riscontrati. Con la settima versione viene rilasciato anche l'update di Swift, che giunge alla seconda release. All'interno dell'IDE è presente un tool per convertire il codice scritto in Swift 1.2, rendendolo compatibile con l'ultima versione.

Ogni nuovo rilascio del sistema operativo è formato da pacchetti di framework che racchiudono metodi e attributi correlati tra loro. Per utilizzarli vanno importati all'interno dei file che compongono il progetto, con la direttiva *import* seguita dal nome del framework desiderato. *Map Kit* è uno dei tanti framework contenuti in iOS e OSX, si occupa della gestione e visualizzazione delle mappe all'interno dell'applicazione. Core Location, invece, è responsabile di recuperare e aggiornare la posizione geografica del dispositivo.
Oltre alle funzionalità che compongono il sistema operativo, è possibile utilizzare librerie scritte da sviluppatori di terze parti.
Google, per esempio, offre le proprie librerie da sostituire a Map Kit per interagire con le mappe. La casa di Mountain View, oltre a queste, ne possiede svariate contenenti funzionalità aggiuntive, come la ricerca di un percorso e la traduzione di un indirizzo human-readable in un punto con coordinate geografiche espresse in latitudine e longitudine e viceversa. Il primo paragrafo si occupa di descrivere l'utilizzo di Map Kit e delle sdk di Google Maps, termina parlando delle differenze tra i due. Il secondo paragrafo introduce due funzionalità molto importanti rilasciate da Google, le Directions e Geocoding API. Il terzo e quarto paragrafo parlano rispettivamente di Core Location e Core Data, quest'ultimo è il framework nativo di iOS responsabile della creazione di modelli utilizzati per la gestione dei dati.

2.2 Map Kit e Google Maps iOS sdk

Prima dell'introduzione da parte di Apple della propria applicazione dedicata alle mappe, il framework Map Kit utilizzava i servizi di Google per la visualizzazione e la gestione di queste. Ad oggi è possibile scegliere quale servizio utilizzare, ognuno caratterizzato dai propri pregi e difetti. Per usare Map Kit [2] basta importarlo essendo un pacchetto contenuto nel sistema operativo. Per utilizzare le sdk di Google Maps [3] bisogna importare le librerie esternamente attraverso l'utilizzo di *Podfile*.

Entrambi i servizi forniscono la possibilità di visualizzare una mappa nella propria applicazione. L'interfaccia dei frameworks può essere utilizzata per modificare il contenuto della mappa. Si possono aggiungere punti d'interesse usando annotazioni e creare *overlays* personalizzati da visualizzare sulla mappa. Per esempio si potrebbero utilizzare le annotazioni per mostrare dei punti d'interesse vicini alla posizione attuale dell'utente come ristoranti o cinema e, grazie agli overlays si potrebbe disegnare sulla mappa il percorso di un bus.

2.2.1 Framework Map Kit

Prima di utilizzare il framework è richiesta l'attivazione della funzionalità *maps* nelle *capabilities* all'interno del progetto Xcode. Map Kit utilizza il *mercator map projection* [4] per mappare i punti tridimensionali che compongono la superficie terrestre in punti bidimensionali per essere gestisti dal framework. Sono supportati tre sistemi di coordinate per definire i punti da visualizzare sulla

mappa. Un *map coordinate* è definito come una struttura di tipo *CLLocationCoordinate2d* che rappresenta una latitudine e longitudine comprese in una rappresentazione sferica della terra, è il metodo più utilizzato per rappresentare un punto nel globo. Un *map point* rappresenta un punto bidimensionale all'interno della mappatura della terra creata dal mercator map projection, anch'esso è definito attraverso una struttura. Infine si possono rappresentare con un *point,* una struttura rappresentante un'unità grafica associata alle coordinate che assume la Map View all'interno del sistema. I map coordinates e i map points vengono mappati come points prima di essere disegnati sulla Map View. Il framework inoltre offre tutti i metodi necessari per la conversione dei punti tra i tre sistemi. Nella figura 2.1 viene mostrato un frammento di codice esemplificativo scritto in Swift 2 che rappresenta la creazione di una mappa, la sua visualizzazione e la creazione di un'annotazione attraverso funzionalità definite in Map Kit. Viene creata una struttura di tipo CLLocationCoordinate2d che definisce un *map coordinate* da poter essere utilizzato all'interno della mappa, successivamente viene settata la posizione e la dimensione che dovrà avere la *mapView,* istanza di tipo *MKMapView,* all'interno della view della sua superclasse. Infine viene settato lo zoom che dovrà avere la mappa e la regione che dovrà mostrare alla sua prima inizializzazione definendo un centro che è rappresentato dal map coordinate definito in precedenza. Nelle ultime cinque righe di codice viene creata un'annotazione di tipo *MKPointAnnotation*, vengono settate le coordinate, il titolo, il sottotitolo e infine viene aggiunta alla mappa.

Nel framework, oltre alle funzionalità riguardanti la gestione della mappa e l'inserimento di annotazioni, è integrato un servizio che interroga i server Apple per la ricerca di un percorso definita una partenza e una destinazione.

```
import MapKit


// creazione di un map coordinate

let location = CLLocationCoordinate2D(
        latitude: 51.50007773,
        longitude: -0.1246402
)

// posizione e dimensione della map view

let frame=CGRect(x: 0, y: 0, width: 200, height: 200)
let mapView=MKMapView(frame:frame)

// zoom e regione che dovrà mostrare la mappa.

let zoom = MKCoordinateSpanMake(0.05, 0.05)
let regione = MKCoordinateRegion(center: location, span: zoom)
mapView.setRegion(regione, animated: true)

//inserimento di un'annotazione sulla mappa

let annotazione = MKPointAnnotation()
annotazione.coordinate = location
annotazione.title = "Big Ben"
annotazione.subtitle = "London"
mapView.addAnnotation(annotazione)
```

Figura 2.1: Creazione di una mappa e di un'annotazione utilizzando Map Kit

2.2.2 Google Maps iOS sdk

Le librerie riguardanti la gestione delle mappe, distribuite ufficialmente da Google, sono disponibili come pacchetto scaricabile attraverso *CocoaPods,* un gestore di dipendenze per progetti scritti in Swift e Objective-C Cocoa. Il gestore è scritto in

Ruby, prima dell'utilizzo necessita di essere installato in ambiente OSX utilizzando il comando da terminale:

$ sudo gem install cocoapods

Il comando *gem* è utilizzato per la gestione dei pacchetti in ambiente Ruby.
Fatto questo, per installare le API di Google attraverso il gestore, si crea un file di testo nominato Podfile con le specifiche necessarie e deve essere importato nella directory del progetto Xcode. In figura 2.2 è mostrato il contenuto del Podfile utilizzato per richiedere l'installazione del pod di Google Maps supportato per iOS 9.0.

```
source 'https://github.com/CocoaPods/Specs.git'
platform :ios, '9.0'
pod 'GoogleMaps'
```

Figura 2.2: Contenuto di un podfile

Salvato il file, si esegue da terminale, dopo essersi posizionati nella directory del progetto contenente il Podfile, il commando:

$ pod install

Installato il pod desiderato, con le rispettive dipendenze, viene creato all'interno della directory un nuovo file per lanciare il progetto Xcode con estensione *.xcworkspace* invece della classica *.xcodeproj*. D'ora in avanti si dovrà utilizzare l'ultimo creato per lanciare il progetto.
Prima di poter utilizzare il servizio di mappe offerto da Google, attraverso la *Google Developers Console* raggiungibile da pagina web è necessario attivare il servizio desiderato e richiedere una *key* da associare all'interno del progetto Xcode. Nella figura 2.3 è mostrato un esempio di codice riportato sulla pagina web ufficiale delle sdk di Google Maps.

```
import UIKit
import GoogleMaps

class YourViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let camera = GMSCameraPosition.cameraWithLatitude(-33.86,
            longitude: 151.20, zoom: 6)
        let mapView = GMSMapView.mapWithFrame(CGRectZero, camera: camera)
        mapView.myLocationEnabled = true
        self.view = mapView

        let marker = GMSMarker()
        marker.position = CLLocationCoordinate2DMake(-33.86, 151.20)
        marker.title = "Sydney"
        marker.snippet = "Australia"
        marker.map = mapView
    }
}
```

Figura 2.3: Utilizzo sdk Google maps per la creazione di una mappa

Guardando il codice si può vedere che ultimata la creazione della view associata al Controller, il codice crea una costante di tipo *GMSCameraPosition* assegnadole una latitudine, longitudine e uno zoom. Viene inizializzata una costante di tipo *GMSMapView*, la classe principale del framework, assegnadole i valori della costante creata in precedenza per il posizionamento della mappa; successivamente viene abilitata la visione della posizione attuale del dispositivo. Per rendere visibile la mappa sul dispositivo, la costante di tipo GMSMapView è aggiunta alla view associata al controller. Nelle ultime righe di codice viene creata un'istanza di tipo *GMSMarker,* eredita dalla classe astratta *GMSOverlay* la quale rappresenta alcuni *overlay* che possono essere raffigurati sulla mappa. All'oggetto viene assegnata una posizione misurata in gradi, un titolo, un sottotitolo e la mappa in cui deve essere mostrato l'oggetto.

2.2.3 Differenze tra i due servizi

Google può contare su un lavoro più che decennale nell'ambito della mobilità, andando di anno in anno a migliorare i servizi esistenti e introdurne sempre di nuovi. I dettagli presenti all'interno delle mappe sono maggiori rispetto a quelli presenti nelle mappe di Apple, specialmente fuori dal territorio americano. Le sdk possono contare su aggiornamenti più frequenti: ad ogni nuovo rilascio da parte di Google, basta lanciare da terminale il comando di aggiornamento dei Podfile, invece per Apple la cadenza tra un rilascio e l'altro è più distante dovendo aspettare di aggiornare il sistema operativo. L'azienda di Mountain View, inoltre, può contare sul vantaggio di fornire la stessa esperienza di utilizzo del servizio sia in ambienti iOS che Android.
Map Kit, dalla sua parte ha la caratteristica di essere stato creato per lavorare in modo nativo in ambienti iOS e OSX garantendo un funzionamento più stabile rispetto alle sdk di Google. Infine ha una migliore integrazione con altri due framework nativi, Core Location e Core Animation, fondamentali nell'ottimizzare la gestione delle mappe e l'animazione che avviene su di esse.

2.3 Servizi Google sulla mobilità

L'azienda di Mountain View offre svariate API per usufruire di servizi riguardanti la mobilità. Nel paragrafo 2.3.1 sono presentate le Directions API, utilizzate per ricercare un percorso dopo aver impostato una partenza e una destinazione, nel 2.3.2 viene introdotto il significato di geocoding, reverse geocoding e l'utilizzo delle Geocoding API. Anche per questi due servizi, come per le sdk delle mappe, è necessario richiedere una key, da associare all'applicazione, attraverso la Google Developer Console. Google per entrambi i servizi permette di effettuare gratuitamente massimo 10 richieste al secondo e 2.500 richieste giornaliere. Si può

pagare una somma di 0.50 dollari per ogni 1000 richieste addizionali fino a un massimo di 100.000 richieste giornaliere.

2.3.1 Directions API

Le Directions API [5] offrono un servizio che calcola il percorso tra due locazioni utilizzando una richiesta https ai server di Google. Questa può essere effettuata per un tragitto da percorrere in macchina, a piedi, in bicicletta oppure utilizzando i mezzi pubblici dove siano supportati. Questo servizio è stato progettato per calcolare l'itinerario tra indirizzi statici forniti in anticipo, non per rispondere alle richieste in tempo reale. All'interno della richiesta https vanno inseriti i parametri dell'indirizzo di partenza e destinazione, il formato della risposta, che può essere di tipo JavaScript Object Notation (JSON) oppure eXtensible Markup Language (XML), infine possono essere aggiunti parametri opzionali se ritenuti necessari. L'indirizzo può essere scritto in formato human-readable oppure espresso con i parametri di latitudine e longitudine. Oltre alla classica ricerca del percorso più breve da una partenza a una destinazione, è supportato anche l'inserimento di waypoints, locazioni intermedie da inserire all'interno dell'itinerario. I punti intermediari non sono supportati nella modalità *transit*, quando si ricerca un percorso da compiere coi mezzi pubblici. Di seguito è riportato un esempio di richiesta https, sono presenti i parametri obbligatori della partenza, della destinazione espressi in gradi e la chiave ottenuta da Google. Come parametro opzionale è settata la modalità di tragitto richiesto (**transit**), si vogliono utilizzare i mezzi pubblici, il formato (***json***) e la lingua (***it***) del file di risposta.

https://maps.googleapis.com/maps/api/directions/json?origin=44.4938530952723,11.3392391055822&destination=44.5038324322405,11.3564800098538&mode=transit&language=it&key=YOUR_API_KEY

2.3.2 Geocoding API

Prende il nome di geocoding il processo di convertire un indirizzo scritto in formato human-readable in coordinate geografiche definite attraverso la latitudine e la longitudine. Il reverse geocoding è il processo inverso, converte una coordinata geografica in un indirizzo.
Come per le Directions API, il servizio [6] è stato progettato per effettuare l'operazione di geocoding/reverse geocoding di un dato statico, conosciuto prima di avviare l'operazione, non supporta l'inserimento di input in tempo reale.
Queste operazioni richiedono tempo e risorse eccessive, quando è possibile si dovrebbe utilizzare una *cache* per memorizzare i valori che vengono recuperati, per riutilizzarli se vengono richiesti all'interno dello stesso progetto o in uno nuovo.

Di seguito viene mostrata una richiesta https indirizzata ai server di Google per richiedere l'operazione di geocoding sull'indirizzo di Via Achillini 20 situato a Bologna in Italia, il file di risposta sarà di tipo JSON. In fondo alla richiesta si inserisce la key associata all'applicazione nella quale si vuole sfruttare il servizio.

https://maps.googleapis.com/maps/api/geocode/json?address=Via+Achillini+20,Bologna,Italy&key=*YOUR_API_KEY*

Sotto viene mostrata una richiesta di reverse geocoding della coordinata geografica di latitudine 44.493853° e longitudine 11.339239°. Come per la richiesta precedente viene impostato il formato JSON come file di risposta e inserita la key.

https://maps.googleapis.com/maps/api/geocode/json?latlng=44.493853,11.339239&key=YOUR_API_KEY

Il file di risposta di entrambi i servizi è pieno di informazioni dettagliate da recuperare per essere utilizzate all'interno dell'applicazione.

2.4 Core Location

Il framework [2][7] è utilizzato per recuperare e gestire la posizione del dispositivo. L'oggetto base che viene creato è un'istanza della classe *CLLocation,* contiene varie proprietà: latitudine, longitudine, altitudine, verso, accuratezza verticale e orizzontale, timestamp ecc.
La latitudine e longitudine sono espresse attraverso una struttura di tipo CLLocationCoordinate2d*,* la stessa che viene utilizzata in Map Kit per la creazione dei *map point.* Si possono impostare vari valori di accuratezza per l'aggiornamento della posizione corrente in base al servizio che deve offrire l'applicazione. Più accuratezza significa più energia consumata da parte del dispositivo, è buona norma impostare sempre il livello minimo necessario e non utilizzarne uno con frequenze di aggiornamento così alte da essere inutilizzate per far sì di non sprecare batteria inutilmente.

Il framework può restituire la posizione, a seconda del livello di accuratezza impostato, in tre modalità differenti: tramite la triangolazione cellulare che è la meno precisa dei tre ma consuma poca batteria, attraverso i nodi wifi che utilizzano il database lookup, più accurata della prima ma più esosa di energia . Come ultima modalità troviamo l'utilizzo dell'hw gps che risiede all'interno del dispositivo per recuperare la posizione, veramente molto precisa ma porta a consumare la batteria in modo drastico. E' molto importante tenere conto del consumo di energia durante la creazione di un'applicazione, potrebbe risultare inutilizzabile se consumasse eccessivamente. Si può dividere la mappa costruendo varie regioni personalizzate e assegnare ad ognuna di queste un livello differente di accuratezza.

In realtà quando il framework aggiorna la posizione attuale, restituisce un *array* di posizioni invece che solamente l'ultima posizione recuperata; se serve solo la

posizione attuale si recupera il dato contenuto nell'indice più basso dell'array, se invece si ha il bisogno di utilizzare varie posizioni precedenti, per esempio in un'applicazione dedicata alla corsa in cui si vuole conoscere tutto il tragitto percorso dall'utente fino a quel momento, si utilizza tutto l'array. Ogni elemento contenuto nell'array ha un campo timestamp per conoscere il momento temporale di ogni posizione recuperata, utile per ricostruire in modo dettagliato il tragitto. Quando si accede per la prima volta ad un'applicazione che utilizza il gestore di Core Location, verrà mostrato un alert nella vista principale per chiedere l'autorizzazione all'utente di poter recuperare informazioni riguardanti la sua posizione. Si può richiedere il permesso di monitorare la posizione indipendentemente dallo stato dell'applicazione o solamente quando è in uso; si deve inserire nel file *info.plist* presente in Xcode il tipo di autorizzazione che si vuole richiedere.

All'interno dell'IDE sono presenti una lista di locazioni che possono essere testate quando l'applicazione è avviata attraverso il simulatore. Oltre a quelle presenti se ne possono importare nuove creando dei file con estensione GPX tramite editor o pagine web apposite. Core Location sfrutta anche la nuova tecnologia iBeacon per scambiare informazioni tramite protocollo bluetooh con altri dispositivi situati nelle vicinanze. Si può monitorare per esempio lo spostamento dei dispositivi vicini, se si allontanano o avvicinano, se entrano o escono da una regione definita.

2.5 Core Data

Core Data [8] è un framework utilizzato per gestire il ciclo di vita di oggetti appartenenti a un certo tipo, la loro persistenza e le relazioni tra essi. Esistono varie modalità di memorizzazione dei dati a seconda dello scopo, la più utilizzata sfrutta un database di tipo SQLite ma non si comporta in modo relazionale. Core Data è un ambiente maturo e testato sufficientemente, Apple continua ad investire importanti risorse per migliorarlo. Essendo molto popolare, nel web si trovano

svariate guide per l'utilizzo e progetti dimostrativi. Utilizzare Core Data per gestire i dati è un buon metodo per minimizzare le dipendenze da librerie di terze parti.

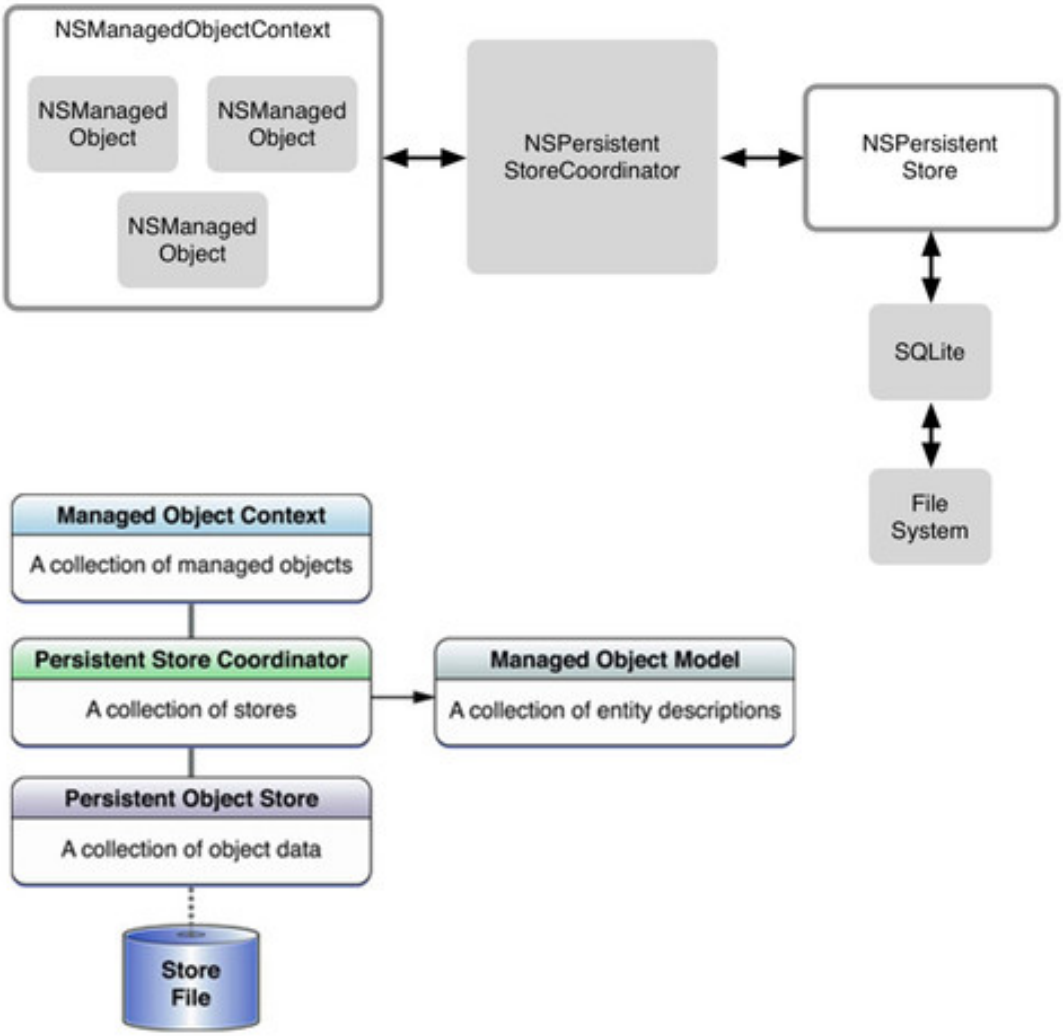


Figura 2.4: Modello di gestione di Core Data

Nella figura 2.4 è rappresentato il modello di gestione che può essere visto come uno stack di oggetti dove nello strato superiore si trova il *managed object context,* un'istanza di tipo *NSManagedObjectContext,* contesto nel quale si trova una collezione di oggetti del tipo *NSManagedObject* o che ereditano da questo. Questi

oggetti rappresentano la visione interna dei dati memorizzati nei vari *stores* (*SQLiteStore,BinaryStore,InMemoryStore*) utilizzati all'interno del progetto. Il contesto è responsabile delle operazioni di manipolazione, salvataggio degli oggetti e creazione delle relazioni tra di essi.

Si possono creare contesti differenti ma devono essere connessi allo stesso *Persistent Store Coordinator,* istanza di tipo *NSPersistentStoreCoordinator,* il quale associa una configurazione di un modello per ogni tipo di salvataggio utilizzato e coordina le operazioni tra il contesto e il Persistent Store, incaricato di recuperare i dati memorizzati, che si trova al livello più basso dello stack.
Molte delle funzionalità di Core Data dipendono dalla schema che è stato creato per descrivere le entità , le loro proprietà e le relazioni tra di esse. Il framework utilizza il *managed object model,* un'istanza della classe *NSManagedObjectModel,* per mappare i dati contenuti all'interno del sistema di memorizzazione utilizzato in un oggetto di tipo NSManagedObject, per essere utilizzati nell'applicazione. Il modello è una collezione di oggetti di tipo *NSEntityDescription,* ognuno di questi descrive un'entità e le relazioni definite al suo interno.
XCode offre la possibilità di importare il framework durante l'inizializzazione di un nuovo progetto. Se viene importato, l'IDE provvederà a creare tutti gli oggetti che compongono lo stack di Core Data tramite una configurazione di default. Verrà creato anche un modello di base dove poter aggiungere tramite un editor visuale varie entità, attributi e relazioni.

Parte Seconda

L'Applicazione

Capitolo 3

Progettazione

E' stata sviluppata una semplice applicazione, scritta in Swift 2, che può essere utilizzata per ricercare il tragitto più breve su strada, dopo aver definito una partenza e una destinazione. Il servizio è disponibile all'interno dell'area urbana nella città di Bologna. Sono presenti due modalità differenti di utilizzo, una indirizzata agli utenti che vogliono usufruire dei mezzi pubblici per spostarsi all'interno della città e l'altra, invece, è stata pensata per essere utilizzata se si è alla guida di veicoli elettrici. In entrambe le modalità viene visualizzata una mappa della città, nella prima vengono mostrate tutte le fermate delle linee degli autobus appartenenti all'azienda TPER (Trasporto Passeggeri Emilia Romagna) e inoltre le postazioni per il prelievo gratuito di biciclette pubbliche, servizio offerto dal

comune della città Nella seconda sono presenti, sulla mappa, le colonnine dove è possibile usufruire della ricarica per veicoli elettrici.

Esistono vari modi per impostare la partenza e la destinazione, può essere inserito un *pin* al centro della mappa oppure sulla posizione attuale del dispositivo, in entrambi i casi esiste la possibilità di spostarlo all'interno di essa. Possono essere anche settate come tali, qualsiasi fermate dei bus o noleggio di bici presenti sulla mappa se si utilizza la modalità riguardante i mezzi pubblici oppure qualsiasi colonnina di ricarica nell'altra modalità. Infine si può impostare una partenza o una destinazione ricercando direttamente un indirizzo tra quelli presenti nello stradario della città.

Nella modalità riferita ai trasporti, dopo aver ricercato e selezionato il percorso ottimale, viene visualizzato nella mappa l'itinerario da percorrere il quale può comprendere uno o più tragitti in autobus e percorsi a piedi. Se non è necessario prendere l'autobus, per esempio se la partenza e la destinazione si trovano a una distanza ravvicinata, viene mostrato il tragitto da percorrere interamente a piedi. Quando si è a bordo di veicoli elettrici, l'utente inserisce l'autonomia del veicolo prima di effettuare la ricerca dell'itinerario. Una volta effettuata, se l'autonomia non è sufficiente per raggiungere destinazione, l'applicazione avvisa il guidatore di questo e propone di ricalcolare il percorso inserendo nel tragitto la sosta alla colonnina di ricarica più vicina. Se il guidatore accetta, viene mostrato il nuovo itinerario con l'aggiunta della ricarica come punto intermedio quando l'autonomia residua del veicolo è sufficiente per raggiungere la colonnina, altrimenti nel peggiore dei casi avvisa che l'autonomia non basta per raggiungere questa e consiglia di utilizzare i mezzi pubblici per spostarsi. Se l'autonomia del veicolo è alta, quindi non significativa per la ricerca, può essere impostata come tale e il percorso verrà calcolato non tenendola in considerazione, come una classica ricerca.

Nei tre paragrafi successivi vengono presentati gli aspetti fondamentali che hanno portato alla creazione e al completamento del progetto. Il primo descrive tutti gli aspetti implementativi di cui si è tenuto conto e delle scelte progettuali. Il secondo si incentra nel descrivere i metodi differenti che sono stati utilizzati per il recupero

e la gestione dei dati. L'ultimo paragrafo è dedicato alle scelte implementative riguardanti la gestione della grafica.

3.1 Scelte implementative

Il progetto è stato sviluppato all'interno dell'ambiente XCode, giunto alla settima versione. Per la scrittura del codice si è deciso di utilizzare la seconda versione del linguaggio Swift, nonostante essere in fase sperimentale non si sono riscontrati bug gravi durante l'utilizzo. Si è fatto uso delle API per iOS 9.1. Durante la scrittura del codice, si sono eseguiti test giornalieri sul simulatore dell'IPhone 6s presente nell'IDE e terminato il progetto, è stato installato anche su un dispositivo fisico. Come framewors nativi, oltre agli indispensabili *Foundation* e *UIKit,* si è utilizzato Core Location per recuperare e gestire la posizione attuale del dispositivo sulla mappa e Core Data per la creazione e gestione di un modello dove al suo interno risiedono tutte le fermate degli autobus con le relative informazioni per ognuna. Per la visualizzazione delle mappe si è scelto di utilizzare le sdk distribuite da Google a discapito del framework nativo Map Kit per fare esperienza sull'importazione di pacchetti di terze parti tramite i podfile e soprattutto perchè sono risultate, dopo un confronto, più dettagliate all'interno della città di nostro interesse. Inoltre sono stati utilizzati i servizi di Directions API e Geocoding API, sempre offerti dalla casa di Mountain View, il primo per la ricerca del percorso ottimale mentre il secondo per la traduzione degli indirizzi. I file, recuperati in formato CSV da vari siti web, contenenti i dati per la creazione dei pin da visualizzare sulla mappa, prima di essere importati all'interno del progetto sono stati creati diversi scripts python appositi per la modellazione e la conversione di questi in formato JSON, preferito per la facilità d'uso. Oltre a questo, è stato creato un altro progetto di supporto, come viene spiegato nel paragrafo 3.2.1, utilizzato solamente per creare un modello, popolarlo con tutte le informazioni riguardanti le fermate degli autobus e infine importarlo in quello principale.

3.2 Recupero e gestione dei dati

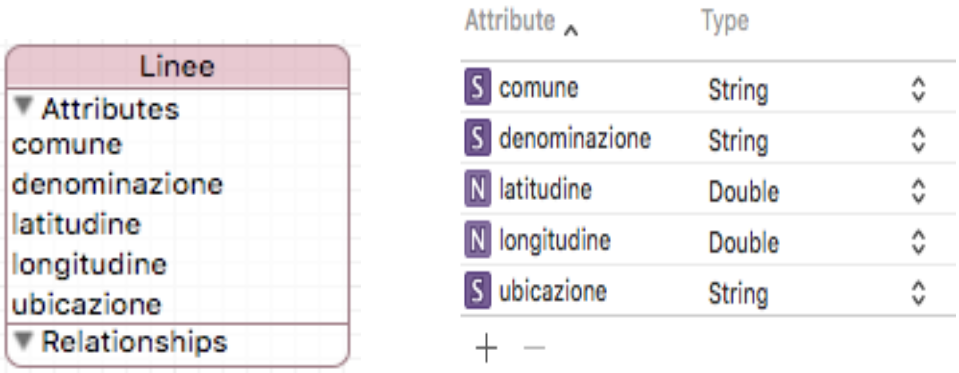
E' stato necessario recuperare diversi dati sul web resi disponibili in modo gratuito, essenziali per la riuscita del progetto. Nei paragrafi successivi verranno descritte le scelte e le modalità utilizzate per renderli disponibili all'interno dell'applicazione, il primo descrive la creazione del modello per gestire le fermate degli autobus utilizzando Core Data e le varie problematiche che si sono riscontrate durante il procedimento, nel secondo viene descritto il metodo per importare e gestire i dati riguardanti le postazioni per ricaricare i veicoli elettrici e per noleggiare le biciclette. L'ultimo verte sull'importazione dello stradario compreso di tutti gli indirizzi facenti parte dell'area urbana di Bologna e su com'è possibile ricercali da parte dell'utente finale.

3.2.1 Creazione Modello Core Data

Per la gestione e la visualizzazione di tutte le fermate degli autobus è stato creato un modello utilizzando il framework nativo Core Data. Il file con all'interno l'elenco di tutte le fermate è stato scaricato, in formato CSV, dalla sezione relativa agli *open data* presente nella pagina web di proprietà dell'azienda TPER [9] la quale gestisce i trasporti su strada e ferroviari principalmente nelle città di Bologna e Ferrara. Il file, dato che comprendeva anche molte fermate che si trovano al di fuori dell'area urbana interessata, è stato modificato eliminando tutti i record non necessari e successivamente è stato convertito in formato JSON, di più facile lettura e gestione all'interno dell'ambiente di sviluppo. La modifica e la conversione sono avvenute utilizzando uno degli scripts python creati appositamente. Nel file

risultante, per ogni fermata, è presente un dizionario di valori informativi su questa: è definita la latitudine, la longitudine, l'ubicazione, la denominazione e il comune di appartenenza.

Ottenute le informazioni necessarie per aggiungere le fermate nella mappa, si è creato un modello, dove immagazzinarle, attraverso la creazione di un progetto Xcode dedicato, importando il framework Core Data durante l'inizializzazione così la creazione degli oggetti necessari per modellare e salvare i dati è affidata all'ambiente di sviluppo. Inizializzato il tutto, si è creata un'entità con i rispettivi attributi definendo un tipo per ognuno, usufruendo del tool offerto dall'IDE, come mostrato in figura 3.1.



(a) (b)

Figura 3.1: Differenti visualizzazioni del tool per gestire i modelli

Una volta creato il modello, per agevolare l'importazione e il recupero dei dati nel database, si crea una classe *Fermata* di tipo *NSManagedObject* come illustrato in figura 3.2, la si assegna all'entità appena creata, attraverso l'editor di Core Data.

```
import Foundation
import CoreData

class Fermata:NSManagedObject{

    @NSManaged var ubicazione:String?
    @NSManaged var denominazione:String?
    @NSManaged var comune:String?
    @NSManaged var latitudine:NSNumber?
    @NSManaged var longitudine:NSNumber?

}
```

Figura 3.2: Creazione classe Fermata

Fatto questo, si passa all'importazione del file JSON contenente le informazioni riguardanti le fermate e al popolamento del database con queste. Il modo più semplice di importare qualsiasi file all'interno dell'*IDE* è, affiancata la schermata della directory dove risiede il file e quella del progetto, utilizzare un trascinamento *drag-and-drop*. Vanno ispezionati tutti gli elementi all'interno del file e per ognuno di essi viene creata un'istanza della classe *Fermata* definendo tutti i suoi attributi con i valori recuperati dal file JSON. Conclusa l'importazione, viene salvato il contesto dove risiede il modello.

Compilato il progetto, come ultima cosa si importa il database contenenti tutte le fermate, composto da tre file con estensione SQLITE, SQLITE-SHM, SQLITE-WAL (Core Data utilizza la *WAL journal mode* di default per creare modelli persistenti), all'interno del progetto ufficiale utilizzando la stessa tecnica del drag-and-drop descritta prima. Arrivati a questo punto, si crea la stessa entità mostrata in figura 3.1 e la classe mostrata in figura 3.2 per interagire con questo. E' importante che il nome dell'entità e gli attributi definiti al suo interno siano gli stessi del progetto utilizzato per la creazione del database, se no il modello risulterebbe incompatibile e sarebbe impossibile accedere ai dati.

Reso compatibile, vengono ispezionati tutti i record al suo interno e per ognuno viene creato un pin e visualizzato sulla mappa per rappresentare la relativa fermata degli autobus utilizzando le sdk fornite da Google.

Figura 3.3: Fermate degli autobus rappresentate sulla mappa

La figura 3.3 mostra la mappa che visualizza una piccola zona della città. Tutti i pin rossi rappresentano le fermate degli autobus, se uno di questi viene selezionato dall'utente, verrà visualizzata una view che fornisce la denominazione della fermata e la possibilità di impostarla come partenza o destinazione.

3.2.2 Importazione colonnine e biciclette

Per i dati necessari alla visualizzazione sulla mappa delle postazioni per ricaricare veicoli elettrici e per usufruire del noleggio biciclette è stato scelto di recuperarli e gestirli attraverso file JSON senza creare nuove entità all'interno del modello fornito da Core Data, essendo meno numerosi di quelli dedicati alle fermate.
Per recuperare le informazioni riguardanti la posizione e i dettagli delle colonnine per la ricarica di veicoli elettrici, si è utilizzato il sito web che gestisce gli Open Data [10] offerti dal comune di Bologna, al suo interno si possono recuperare gratuitamente svariati dataset. Recuperati i dati in formato CSV, si è convertito il file in formato JSON sempre attraverso uno script python com'è avvenuto per i dati nel paragrafo precedente. Una volta importato all'interno del progetto, si è utilizzata la classe nativa *NSJSONSerialization* per il recupero dei dati. Sempre con le solite *sdk* si sono creati i *pin* per essere visualizzati sulla mappa utilizzando le informazioni recuperate.

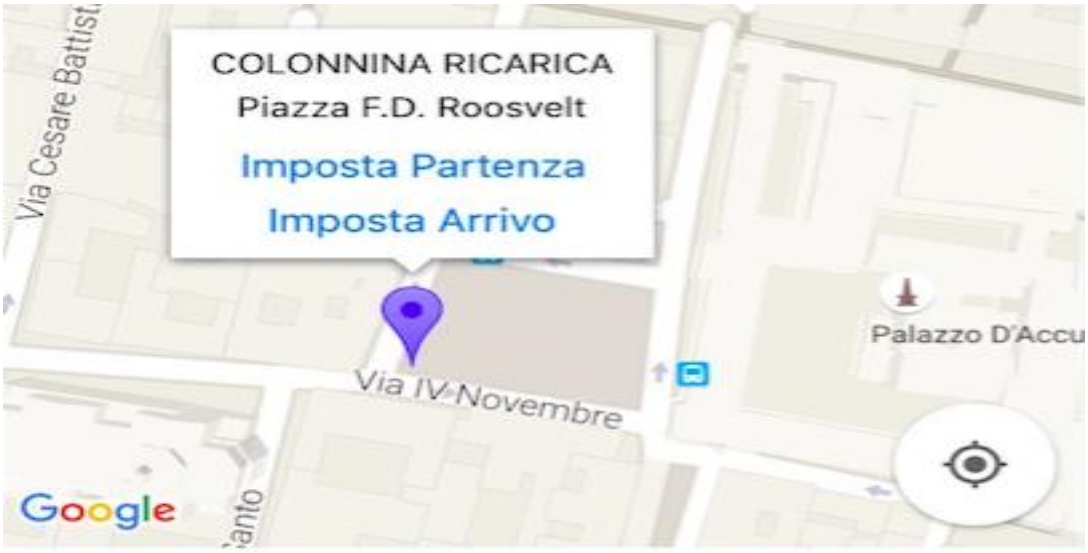


Figura 3.4: Visualizzazione colonnine di ricarica per veicoli elettrici

La figura 3.4 mostra una postazione per ricaricare veicoli elettrici, anche in questo caso può essere impostata come una partenza e una destinazione.

Per l'inserimento delle postazioni di noleggio biciclette, non sono stati trovati dati scaricabili ma solamente consultabili attraverso una mappa presente sulla pagina web [11] del comune di bologna. Con un procedimento manuale e con l'aiuto di un tool online, si sono recuperati per ogni fermata i dati relativi alla latitudine, longitudine, al numero di bicilette prelevabili e sono stati inseriti in un file JSON. Dopo averlo importato nel progetto, come per le colonnine di ricarica elettrica è stato utilizzato direttamente per la creazione e la visualizzazione dei pin sulla mappa.

Figura 3.5: Visualizzazione postazione biciclette

La figura 3.5 mostra una postazione dove si possono prelevare biciclette, quando si preme sul pin anche in questo caso può essere impostato come partenza o destinazione. Oltre all'ubicazione della fermata, viene visualizzato anche il numero di biciclette prelevabile che varia a seconda della postazione.

3.2.3 Importazione Stradario

La partenza e la destinazione possono essere definite eseguendo una ricerca in formato human-readable tra gli indirizzi presenti all'interno dell'area urbana. Il file contenente tutto lo stradario della città è stato recuperato dal sito web degli Open Data e convertito in formato JSON come per le colonnine di ricarica. Importato nel progetto, sono stati copiati tutti i dati all'interno di un array per la necessità di filtrare gli elementi al suo interno durante la ricerca.

In Figura 3.6 è mostrata la ricerca di un indirizzo, grazie alla definizione di un filtro specifico per analizzare la stringa di ricerca inserita dall'utente, vengono recuperati dall'array solamente gli indirizzi compatibili con la ricerca, cioè quelli che iniziano con le stesse lettere, e mostrati. Ogni volta che si varia la stringa di ricerca, inserendo o eliminando caratteri, il filtro riesamina tutti gli indirizzi

recuperando quelli aggiornati da visualizzare. Dopo aver selezionato l'indirizzo desiderato può essere inserito anche il numero civico per una maggior precisione nella ricerca.

Figura 3.6: Esempio di ricerca indirizzo

3.3 Scelte grafiche

Per creare i View Controllers, gli oggetti che si occupano della gestione e presentazione delle rispettive viste all'utente finale, e connetterli tra di loro, si è scelto di utilizzare lo storyboard, tool grafico presente all'interno dell'ambiente di sviluppo. Questo permette di aggiungere e posizionare oggetti all'interno delle views, per esempio bottoni, label, subviews, altri controllers di diverso tipo in modo semplice ed efficace. Inoltre permette di creare dei collegamenti tra i vari View Controllers, detti *segue*, utilizzati per il trasferimento dei dati e informazioni tra essi. Il progetto in questione è composto da tre View Controllers che vengono presentati nel dettaglio nei paragrafi successivi. Il primo descrive quello incaricato a mostrare la view che si presenta al lancio dell'applicazione, quella principale, nel secondo viene descritto quello che gestisce la ricerca e la selezione di un indirizzo in formato human-readable. L'ultimo paragrafo descrive il controller che si occupa della visualizzazione dell'itinerario che è stato selezionato dalla ricerca. Nella figura 3.7 è mostrato lo storyboad completo del progetto

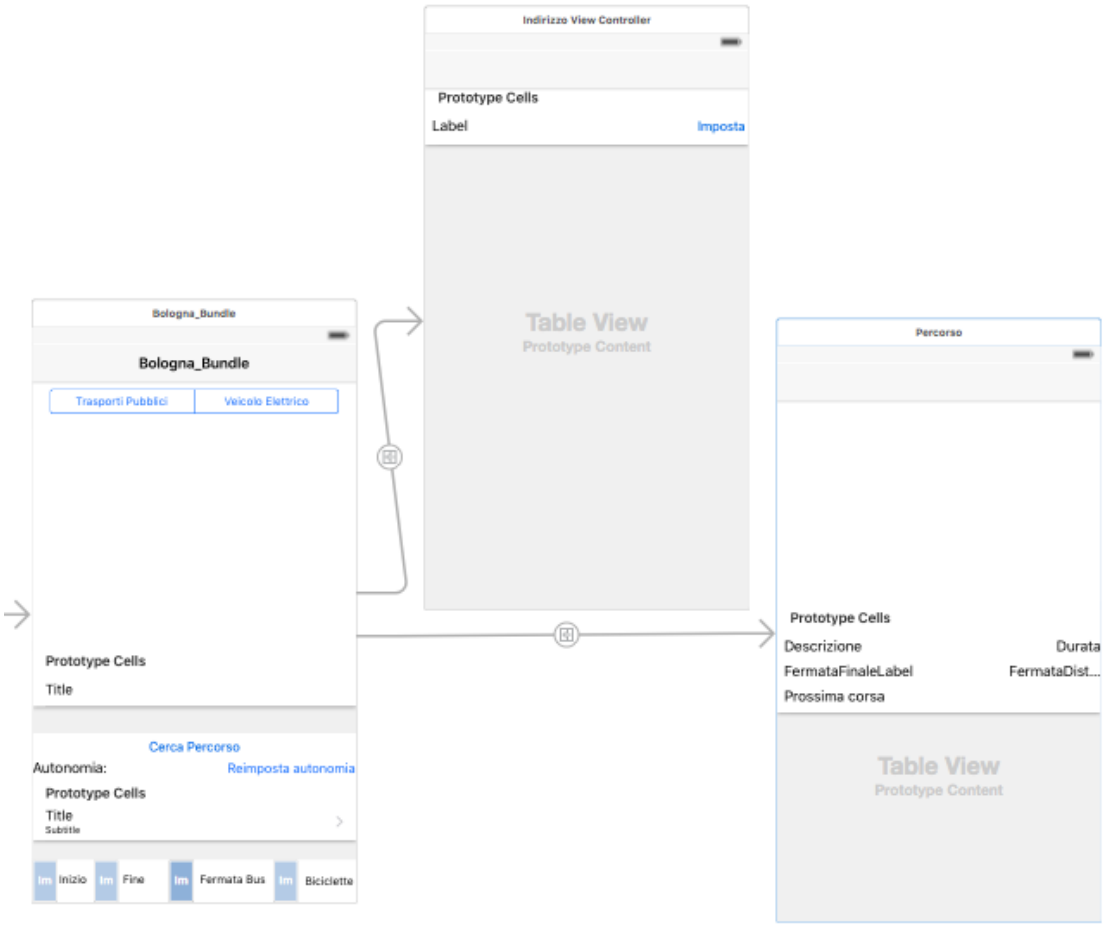


Figura 3.7: Storyboard completo

3.3.1 Map View Controller

Questo è il View Controller che appare al lancio dell'applicazione, si occupa della visualizzazione e gestione della mappa, dell'inserimento in essa di tutte le informazioni recuperate e permette all'utente di impostare l'indirizzo di partenza e destinazione nei vari modi possibili descritti. Effettuata la ricerca, mostra il risultato con vari dettagli sull'itinerario. Sempre in questo controller esiste la possibilità di cambiare modalità rimanendo al suo interno come mostrato in figura 3.8, utilizzando il *Segmented Control* posizionato in alto sopra alla mappa.

Per ogni modalità, la mappa mostra tutti i punti di interesse, per facilitarne la comprensione è stata inserita una legenda in fondo alla view. Ogni volta che si

imposta una partenza viene mostrato un *pin* verde, invece grigio per la destinazione. Sotto alla mappa è visualizzata una Table View, mostrata nel dettaglio in figura 3.9, utilizzata per ricercare la partenza e la destinazione.

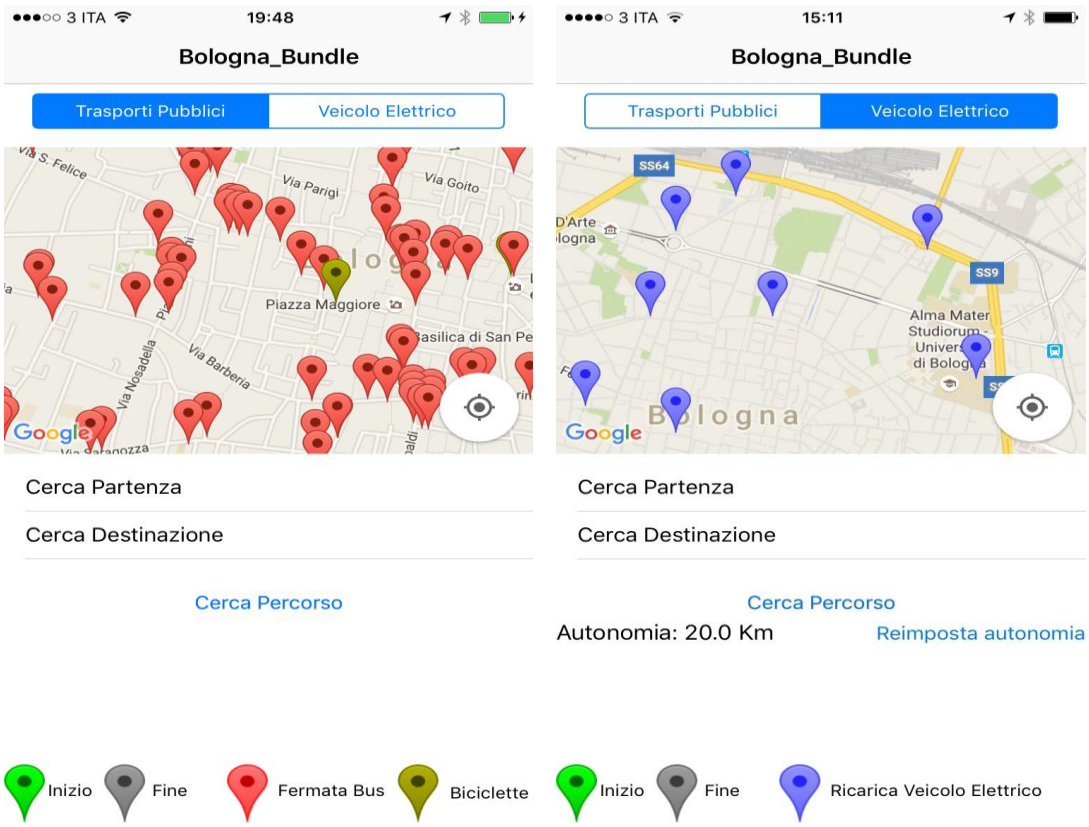


Figura 3.8: Rappresentazione delle due modalità nel view controller principale

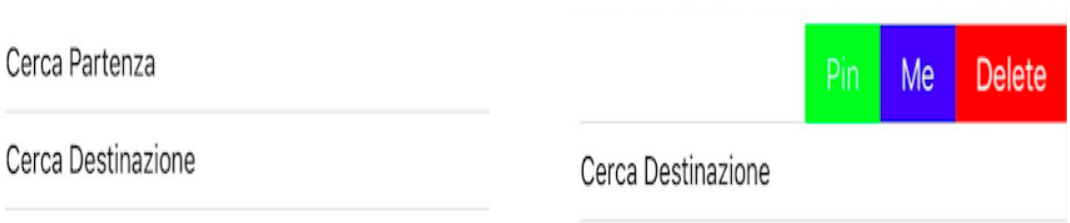


Figura 3.9: Table view utilizzata per impostare la partenza e la destinazione

Se viene premuta una delle due celle presenti nella tabella, viene mostrato un nuovo View Controller, descritto nel paragrafo 3.3.2, utilizzato esclusivamente per ricercare l'indirizzo da impostare. Trovato l'indirizzo desiderato, si torna al View Controller principale e viene inserita un'annotazione sulla mappa nella posizione desiderata. Effettuando uno swipe a sinistra sulla cella, invece, appariranno tre opzioni come si vede in figura. La prima a sinistra, **Pin**, se premuta inserisce un pin al centro della mappa, la seconda, **Me**, lo inserisce nella posizione attuale del dispositivo. L'ultima, **Delete**, elimina l'annotazione dalla mappa se presente. I pin che rappresentano la partenza e la destinazione, se premuti a lungo, possono essere spostati all'interno della mappa. Al centro è presente il bottone per avviare la ricerca dell'itinerario, sotto viene presentato il risultato una volta conclusa. Solo nella modalità per veicoli elettrici, viene visualizzata l'autonomia residua impostata da parte dell'utente, con la possibilità di modificarla in qualsiasi momento.

3.3.2 Ricerca Indirizzo View Controller

Questo View Controller è incaricato di gestire la ricerca e la selezione dell'indirizzo richiesto, appare sul dispositivo ogni volta che si preme su qualsiasi delle due celle che compongono la Table View presente nel View Controller principale descritto nel paragrafo 3.3.1 .

Come mostrato in figura 3.10, sono due gli oggetti principali. Un Search Controller, in alto, dove inserire l'indirizzo da ricercare in formato human-readable. All'interno di questo controller è presente un filtro che entra in azione ogni volta che si modifica la stringa di ricerca, recupera all'interno dell'Array contenente tutti gli indirizzi solo quelli compatibili col *pattern* di ricerca e li trasferisce alla *Table View* che li mostra. Una volta mostrati, la ricerca può continuare più nel dettaglio aggiungendo lettere alla stringa di ricerca, oppure può essere selezionato l'indirizzo di interesse. Se premuto, appare un bottone per confermare la scelta, prima è possibile aggiungere il numero civico come mostrato in figura. Impostato l'indirizzo desiderato, riappare il View Controller principale e viene inserito un

nuovo pin sulla mappa di colore verde se abbiamo impostato la partenza altrimenti grigio per la destinazione. Si è deciso di introdurre il bottone nella conferma della scelta per obbligare la ricerca solamente tra le vie presenti nello stradario della città, con la pressione del tasto di conferma presente nella tastiera non viene ripresentato il View Controller principale, evitando di impostare la partenza e la destinazione con indirizzi non presenti in città.

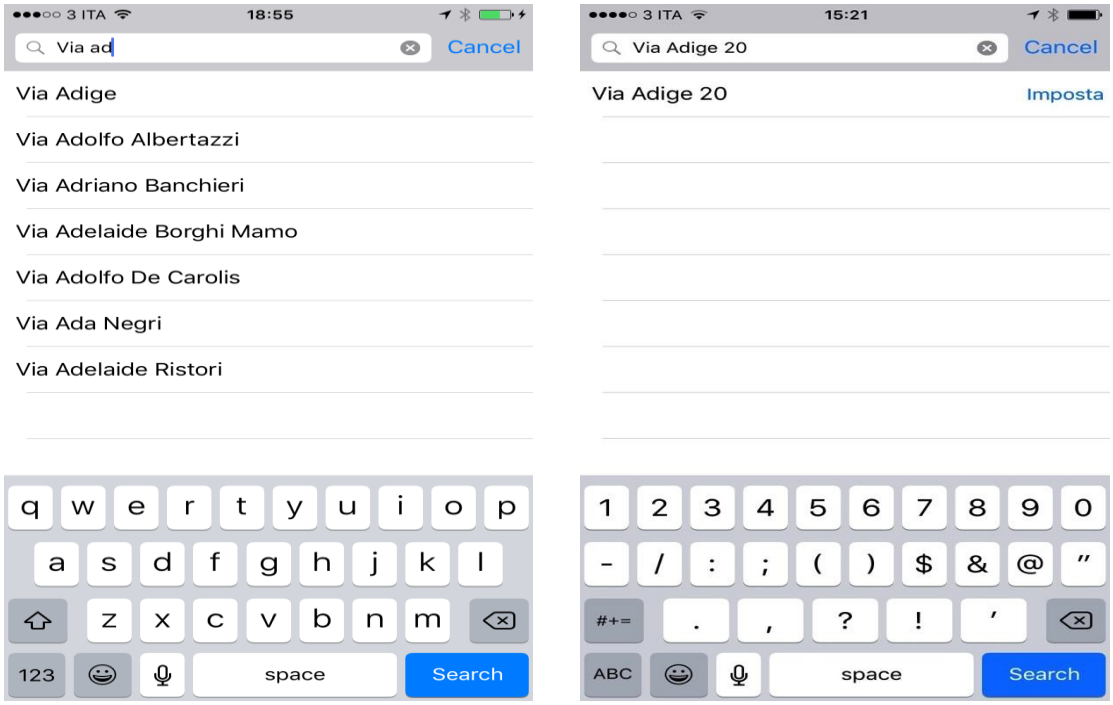


Figura 3.10: Ricerca indirizzo

3.3.3 Itinerario View Controller

Definita la partenza e la destinazione, può essere svolta la ricerca dell'itinerario. Viene visualizzato sul View Controller principale il risultato di questa compreso di dettagli importanti. Se ritenuto soddisfacente, può essere selezionato da mostrare sulla mappa oppure può essere effettuata una nuova ricerca. Quando viene selezionato appare un nuovo View Controller che si dedica alla visualizzazione del percorso nei dettagli. In figura 3.11 sono mostrati due itinerari, quello a sinistra è il

risultato di una ricerca avvenuta utilizzando la modalità dei trasporti pubblici invece quello a destra utilizzando quella per i veicoli elettrici.

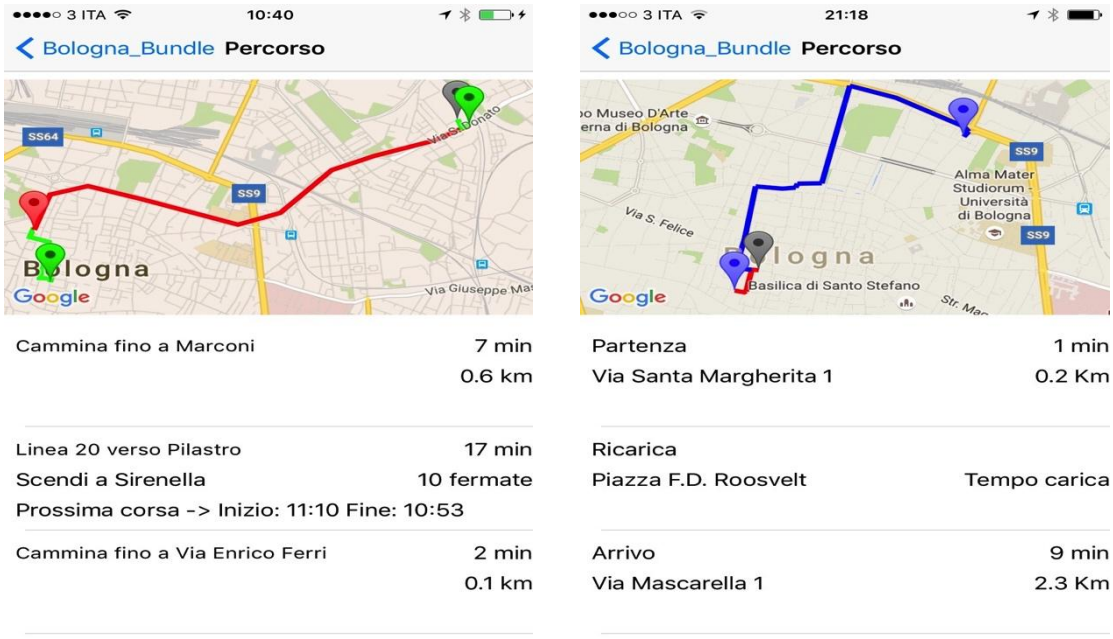


Figura 3.11: Dettagli sugli itinerari

All'interno del View Controller è presente una mappa, gestita utilizzando sempre le sdk di Google, dove vengono mostrati solo i punti di interesse riguardanti l'itinerario. Questo è suddiviso in vari *step*, ognuno di questi è rappresentato visivamente da un overlay colorato che unisce due pin e, da una cella della Table View, presente sotto la mappa, con all'interno tutti i dettagli necessari. Nelle celle che rappresentato gli step da effettuare in autobus è visualizzata la linea da prendere, il verso di percorrenza, l'ora in cui passerà il prossima autobus e quella di fine tragitto. Inoltre viene indicato il numero di fermate da effettuare. Nella modalità di ricerca dei trasporti pubblici gli overlays colorati di rosso rappresentato un pezzo del tragitto da effettuare in autobus, le zone verdi si devono percorrere camminando. In quella riferita ai veicoli elettrici il tratto rosso rappresenta la strada da percorrere per arrivare alla colonnina di ricarica. Se è impostata un'autonomia alta, viene visualizzata sulla mappa solamente la partenza e la destinazione collegate da un unico overlay di colore blu.

Capitolo 4

Utilizzo reale dell'applicazione

Sfruttando la novità introdotta nella settima versione dell'ambiente di sviluppo, quella di poter installare ed utilizzare in modo gratuito un'applicazione direttamente in un dispositivo fisico senza utilizzare il simulatore presente nell'IDE, è stata eseguita una prova di utilizzo di entrambe le modalità all'interno dell'area urbana della città. Nel primo paragrafo viene presentata la simulazione effettuata utilizzando la modalità riguardante la ricerca di un percorso sfruttando i trasporti pubblici, invece il secondo presenta la ricerca di un itinerario trovandosi a bordo di un veicolo elettrico.

4.1 Modalità trasporti pubblici

Nel paragrafo viene presentata la simulazione di utilizzo dell'applicazione all'interno della città, sfruttando i mezzi pubblici per giungere a destinazione. Dopo aver parcheggiato l'autovettura all'interno del Parcheggio Tanari situato nel centro della città, avviando l'applicazione viene mostrata la schermata iniziale mostrata in figura 4.1. Vicino alla posizione del parcheggio è presente sulla mappa una postazione per il prelievo di biciclette, mostrata con il pin giallo, e una fermata degli autobus rappresentata dal pin rosso. Si è scelto di utilizzare

quest'ultima per impostare la partenza, premendo sull'annotazione e successivamente sul bottone desiderato per impostarla come tale, contenuto nella *Callout View* apparsa dopo la pressione del pin come mostrato in figura 3.1.

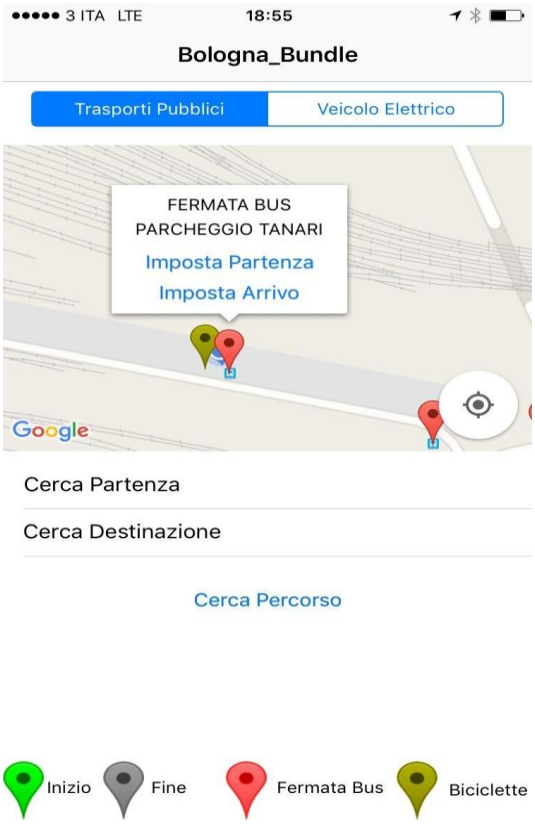
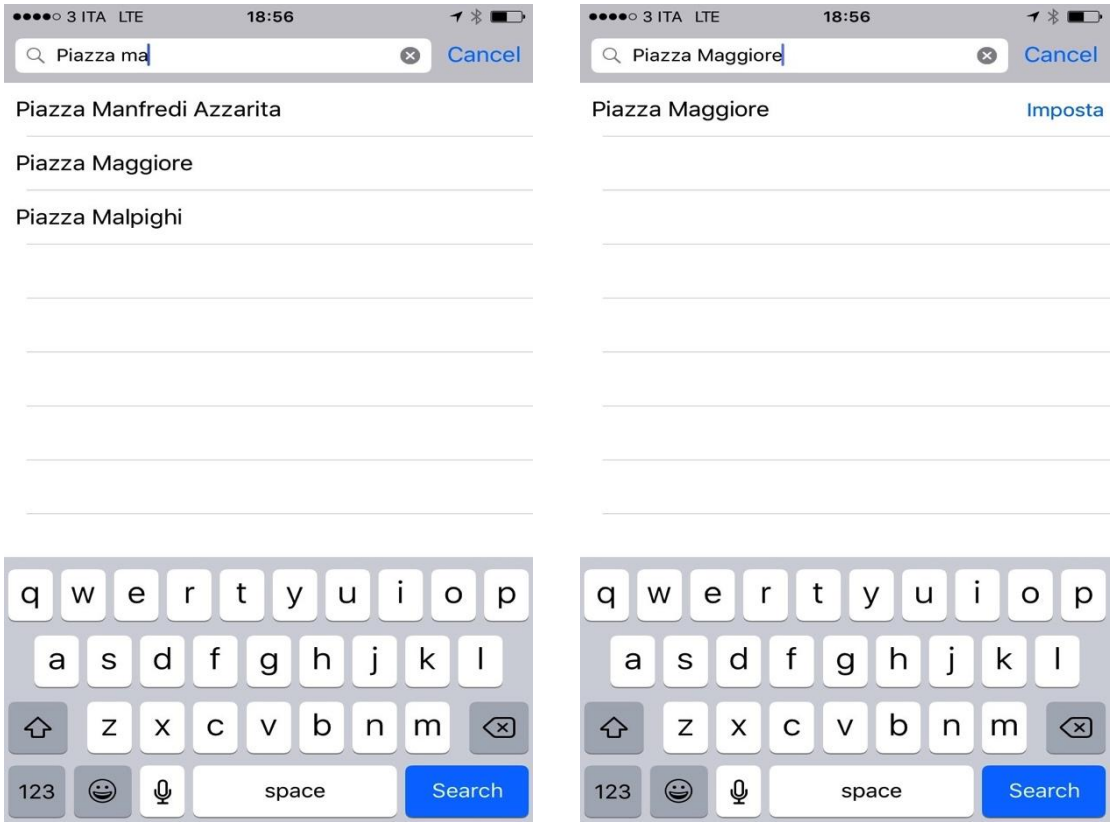


Figura 4.1: Impostazione partenza

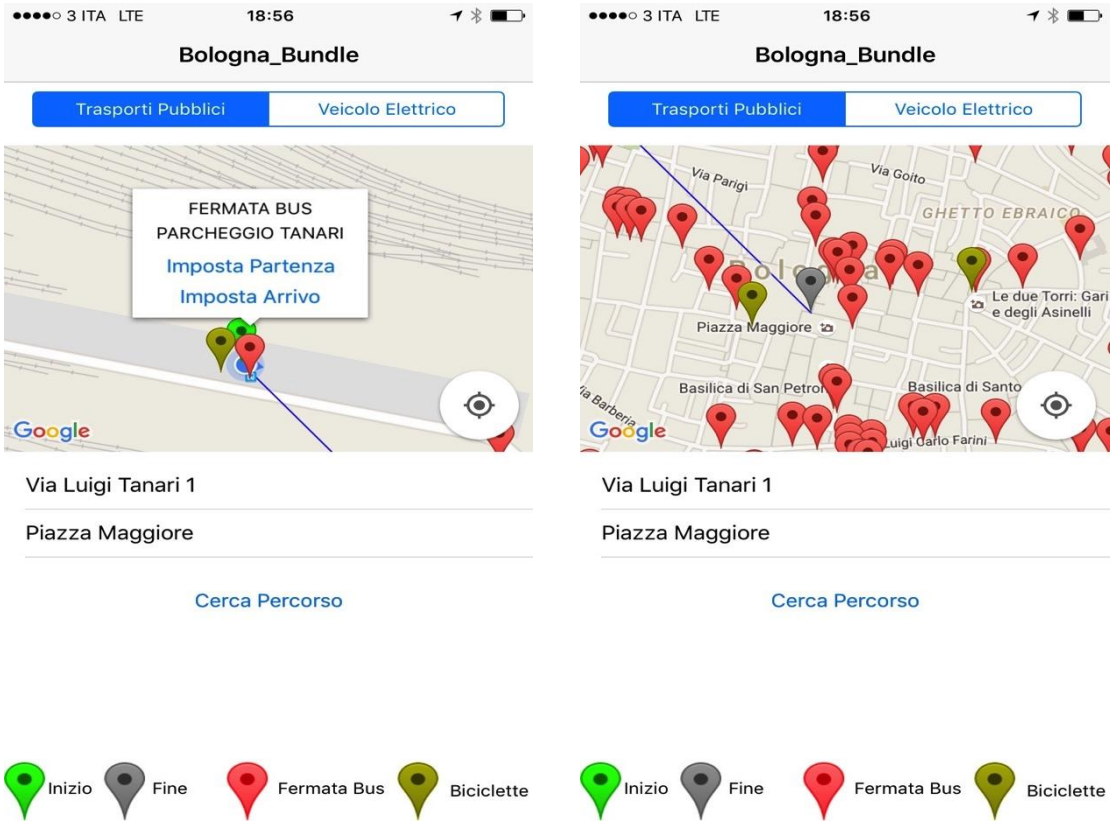
La destinazione, invece, è stata impostata inserendo l'indirizzo da ricercare in formato human-readable. Nella Table View presente sotto la mappa è stata premuta la cella dedicata alla presentazione del view controller che gestisce la ricerca della destinazione tra gli indirizzi contenuti nello stradario della città, come si può vedere in figura 3.2. E' stata scelta come destinazione Piazza Maggiore, nel pieno centro storico della città. Inserita la stringa per ricercare l'indirizzo, nella nostra simulazione *Piazza ma*, sono apparsi nella tabella sottostante al Search Controller quelli compatibili con la ricerca, come mostrato in figura 3.2 (a). Trovato l'indirizzo desiderato, è stata premuta la cella che lo contiene e successivamente il bottone apparso al suo interno, come mostrato in figura 3.2 (b), per ritornare alla schermata principale con l'indirizzo desiderato impostato come destinazione.



(a) (b)

Figura 4.2: Selezione dell'indirizzo di destinazione

La figura 4.3 mostra la schermata principale con i due indirizzi impostati. Il *pin* verde che visualizza la partenza, mostrata in figura 4.3 (a), è stato aggiunto alla mappa utilizzando le sdk di Google recuperando le informazioni di latitudine e longitudine per il posizionamento direttamente dal modello gestito da Core Data contenenti i dettagli delle fermate degli autobus. In figura 4.3 (b) è visualizzato il pin grigio rappresentante la destinazione, questo è stato inserito dopo aver recuperato le informazioni necessarie utilizzando il servizio di geocoding offerto da Google. Le due annotazioni sono unite da un overlay per facilitarne la visualizzazione, specialmente nelle zone dove sono presenti svariate fermate degli autobus.



(a) (b)

Figura 4.3: Visualizzazione della partenza e destinazione

Impostata la partenza e la destinazione, premendo il bottone presente al centro del view controller viene eseguita la ricerca. Per recuperare l'itinerario ottimale, l'applicazione utilizza le Directions API che interrogano i server di Google.

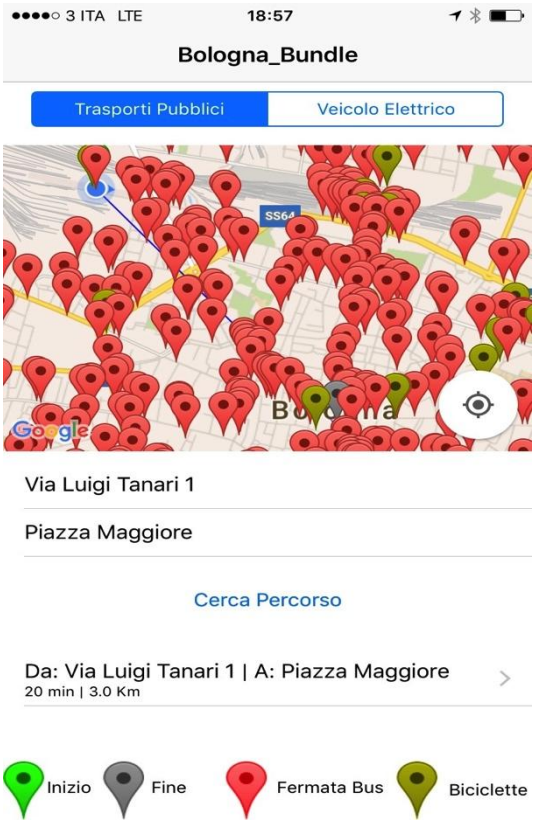
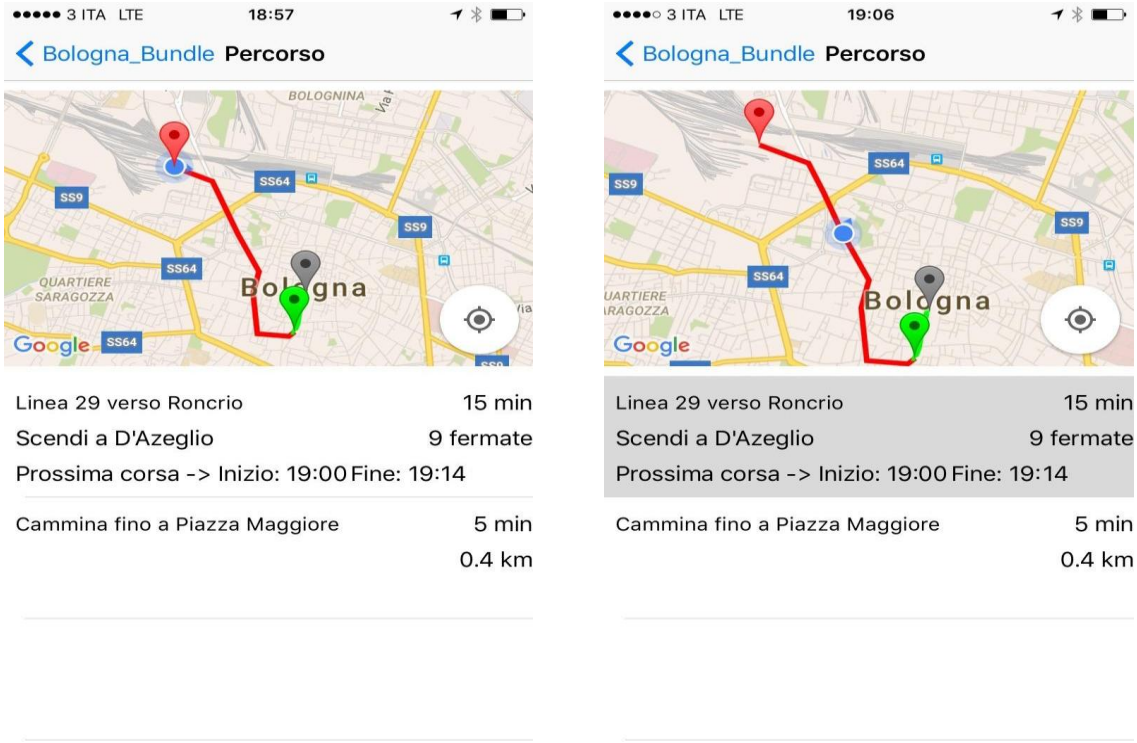


Figura 4.4: Ricerca dell'itinerario

Ottenuto il percorso finale, appare una nuova Table View formata da un unica cella che visualizza le informazioni riguardanti l'itinerario trovato: la partenza, la destinazione, il tempo e la distanza necessarie per ultimarlo. Selezionando la cella viene presentato il view controller, descritto nel paragrafo 3.3.3, che si occupa di visualizzare i dettagli del percorso come mostrato in figura 4.5 (a). Le informazioni visualizzate in tabella ci mostrano che per raggiungere destinazione è consigliato prendere la linea numero 29 in direzione Ronchio, scendere dal mezzo dopo nove fermate, a D'azeglio. La prossima corsa passa alle 19:00 e dura quindici minuti. La

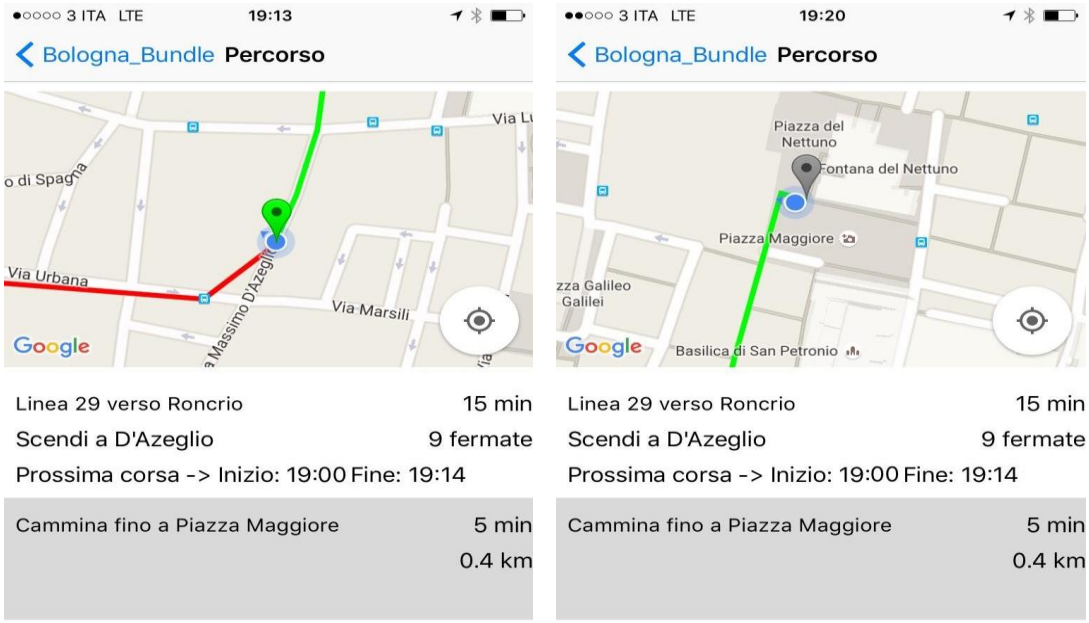
salita sull'autobus è avvenuta alle 18:57, dopo tre minuti è partito, allo stesso orario visualizzato nei dettagli presenti nell'applicazione. Nella figura 4.5 (b) è mostrata la posizione in cui mi trovavo alle 19:06, dopo che l'autobus ha effettuato la quarta fermata.



(a) (b)

Figura 4.5: Visualizzazione dettagli dell'itinerario

Alle 19:13, il mezzo ha raggiunto la fermata D'azeglio, un minuto prima dell'orario visualizzato, come mostrato in figura 4.6 (a). Dopo aver ultimato la corsa, è necessario camminare 0.4 Km nel tempo calcolato di cinque minuti per raggiungere destinazione, come descritto nella seconda cella della Table View. Alle 19:20 si è giunti a destinazione in Piazza Maggiore come mostra la figura 4.6 (b).



(a) (b)

Figura 4.6: Ultimo step dell'itinerario

4.2 Modalità veicolo elettrico

Per quanto riguarda la modalità disponibile per i veicoli elettrici, si è ricercato un percorso impostando un'autonomia bassa per fare in modo che la ricerca potesse comprendere la sosta in una colonnina di ricarica.

Passando dalla modalità descritta nel paragrafo precedente a questa, viene visualizzato un Alert Controller come mostrato in figura 4.7 il quale avverte di impostare l'autonomia. Selezioniamo l'opzione di *Autonomia Alta*, in questo momento si vuole visionare il percorso senza preoccuparsi di questa, viene reimpostata in un secondo momento.

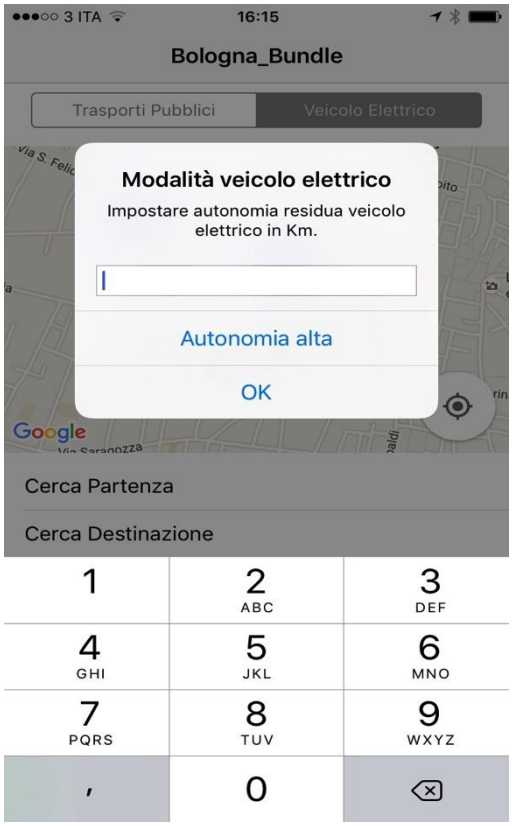


Figura 4.7: Alert Controller

Nella figura 4.8 viene mostrata sulla mappa la posizione attuale del veicolo identificata dal pallino blu e svariati pin di colore blu che rappresentato le colonnine per la ricarica elettrica.

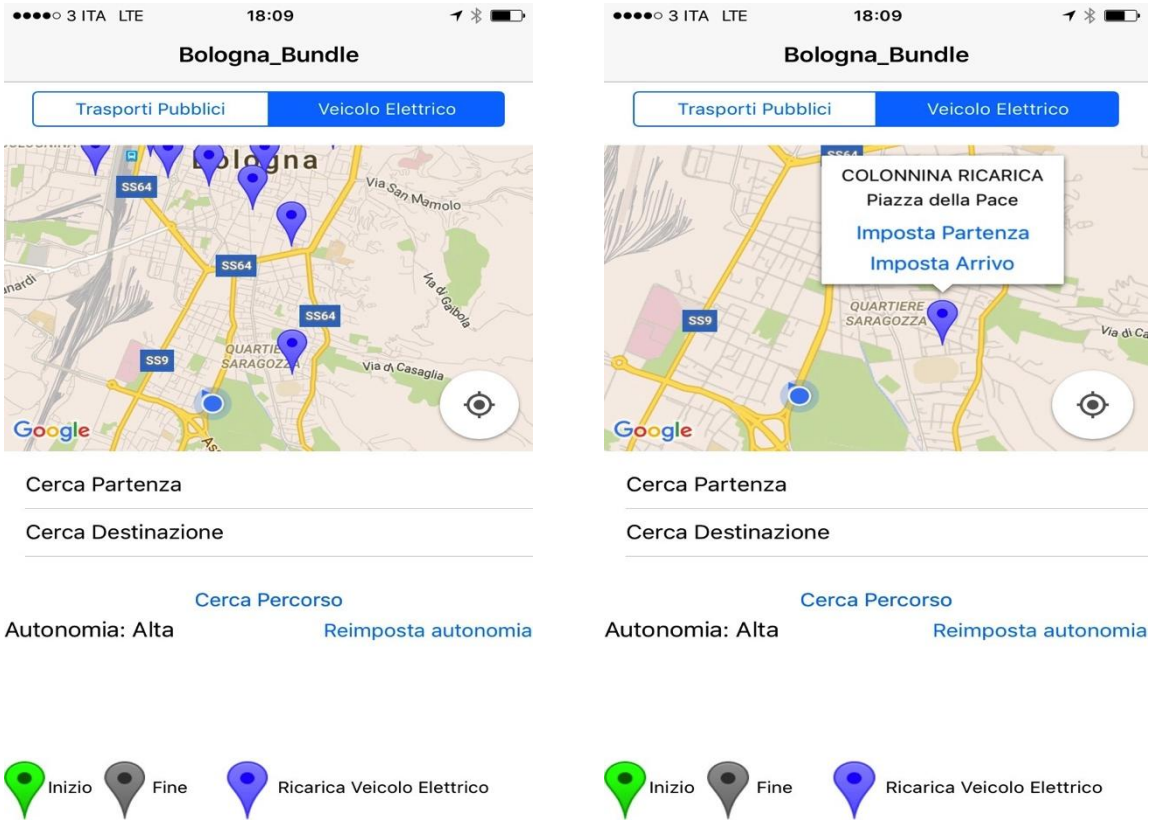


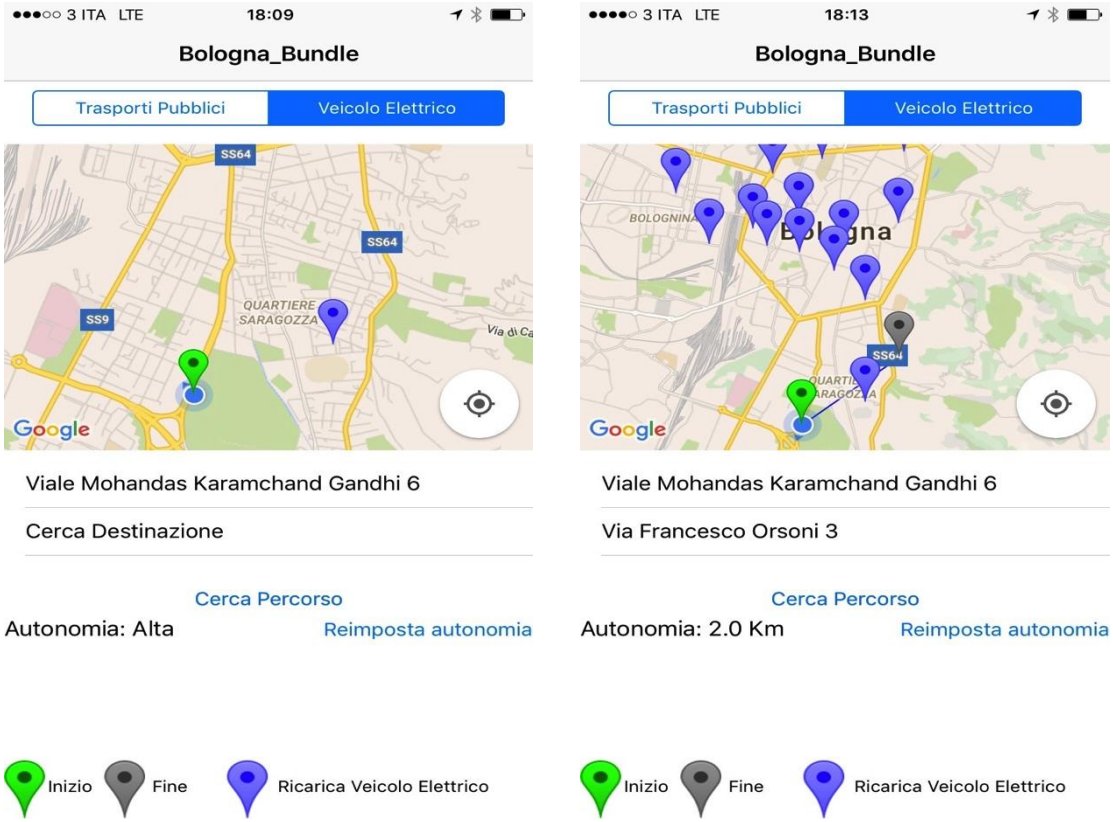
Figura 4.8: Visualizzazione colonnine di ricarica per veicolo elettrico

La partenza è stata impostata sulla posizione attuale del dispositivo, opzione disponibile effettuando uno swipe a sinistra sulla cella desiderata e scegliendo l'opzione **Me**. Per inserire l'annotazione di colore verde l'applicazione utilizza Core Location per recuperare le coordinare geografiche del dispositivo e successivamente utilizza il servizio di reverse geocoding offerto da Google per convertirle in formato human-readable. L'indirizzo ottenuto dalla conversione viene visualizzato nella cella apposita all'interno della Table View sotto la mappa, come mostrato in figura 4.9 (a).

Per impostare la destinazione, si è utilizzata l'opzione **Pin**, sempre disponibile utilizzando lo swipe, che inserisce al centro della mappa visualizzata un'annotazione. Apparso il pin, è stato trascinato sulla mappa dopo aver eseguito una pressione prolungata su di esso, utilizzata per attivare lo spostamento. In

figura 4.9 (b) è visualizzata la posizione finale della destinazione, annotazione di colore grigio. Anche quando si utilizza l'opzione **Pin** e ogni volta che si conclude lo spostamento di un'annotazione all'interno della mappa, viene utilizzato il reverse geocoding per recuperare l'indirizzo in formato human-readable da visualizzare all'interno della Table View.

Infine premendo il bottone *Reimposta autonomia* al centro dello schermo, viene presentato il controller mostrato in figura 4.7, questa volta viene settata un autonomia residua di 2 Km.



(a) (b)

Figura 4.9: Impostazione partenza e destinazione

Inserita la partenza e la destinazione, viene ricercato il percorso ma, come si può vedere in figura 4.10 (a), servono minimo 3,7 Km di autonomia all'interno del veicolo ma ne ha disponibili solamente due. Appare un altro alert controller per

informare del problema riscontrato e consigliare di ricalcolare il percorso inserendo nel tragitto la sosta alla colonnina di ricarica più vicina. Si è scelto di ricalcolare il percorso, in figura 4.10 (b) viene mostrato l'itinerario risultate dalla seconda ricerca. Per includere la sosta di ricarica nel tragitto, il percorso si è allungato passando dai 3,7 Km ai 4,3 Km come si può vedere dalla informazioni apparse all'interno della cella che indica l'itinerario trovato. Viene mostrato anche che il tempo totale per completare l'itinerario è di, sei minuti per raggiungere la ricarica più il tempo necessario per ricaricare il veicolo più sette minuti per raggiungere destinazione dalla colonnina.

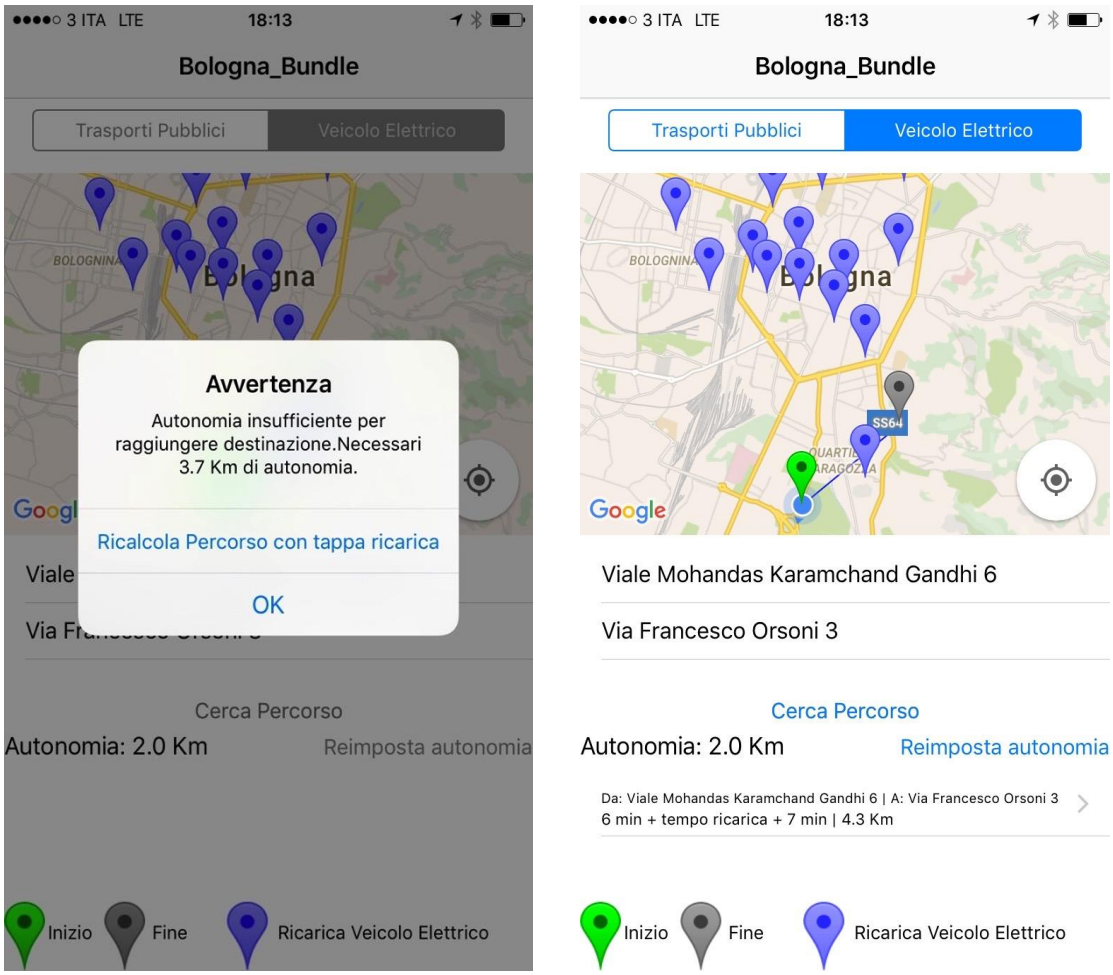


Figura 4.10: Ricalcolo percorso causa autonomia insufficiente

Se l'autonomia residua non è sufficiente neanche per arrivare alla colonnina di ricarica più vicina, l'applicazione avverte il conducente di utilizzare i trasporti

pubblici per giungere a destinazione. Dopo aver selezionato il percorso, appare il controller mostrato in figura 4.11 utilizzato per visualizzare i dettagli dell'itinerario.

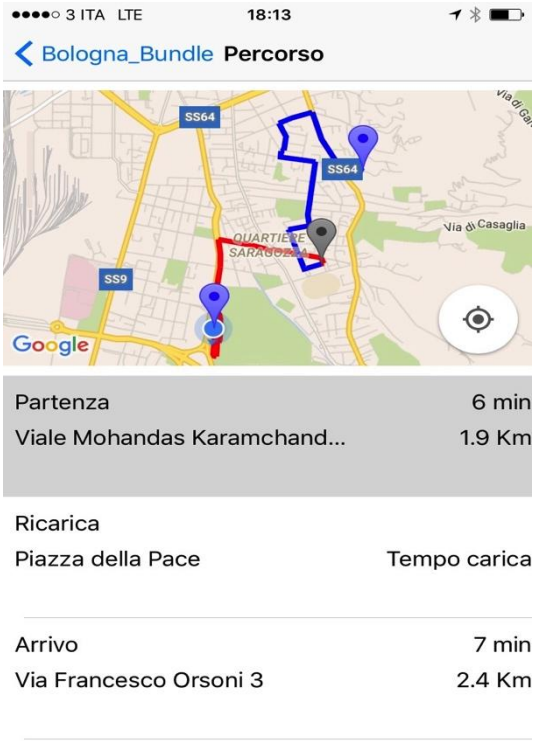


Figura 4.11: Dettagli del percorso

La figura 4.11 mostra il veicolo sulla posizione di partenza, avvenuta alle 18:13, per giungere alla colonnina di ricarica in Piazza della Pace è necessario percorrere 1,9 Km, di poco inferiori all'autonomia del veicolo impostata a 2 Km. Dopo nove minuti, tre in più di quelli necessari secondo le informazioni recuperate, alle 18:24 il veicolo si trova davanti alla colonnina di ricarica, com'è mostrato in figura 4.12. Dopo aver ricaricato in una delle due colonnine disponibili, il tragitto è proseguito per altri sette minuti, fino a destinazione in Via Francesco Orsoni 3 come si può vedere in figura 4.13.

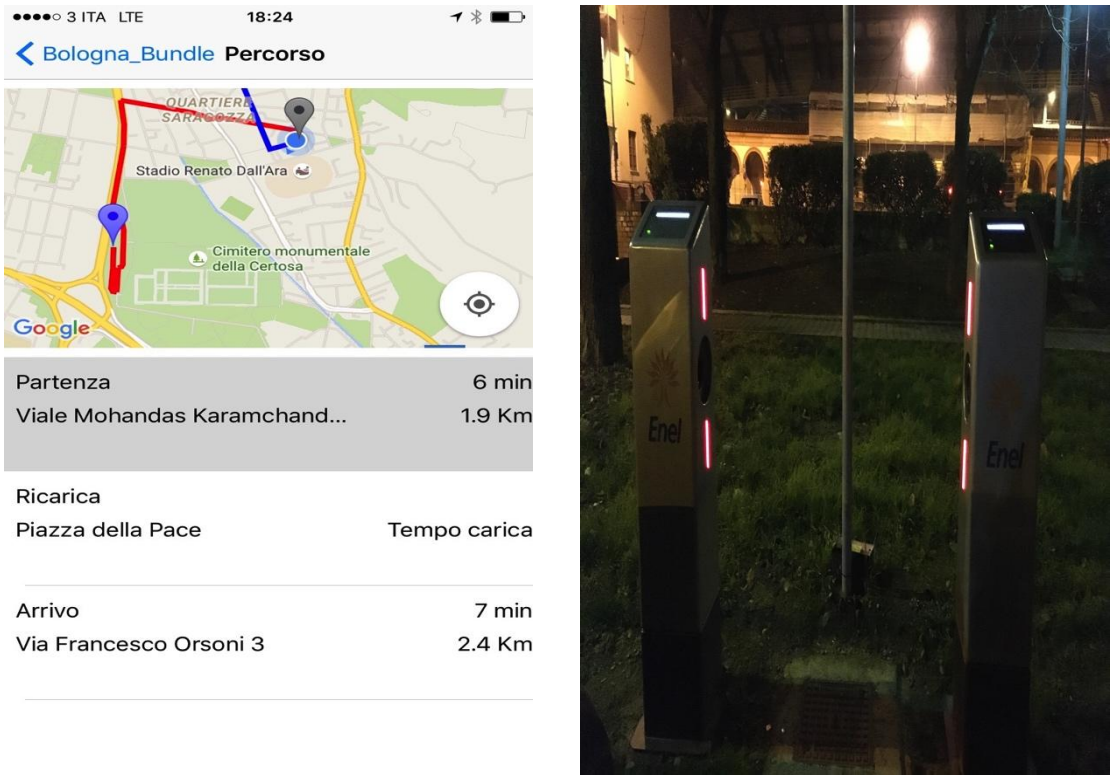


Figura 4.12: Postazione di ricarica per veicoli elettrici

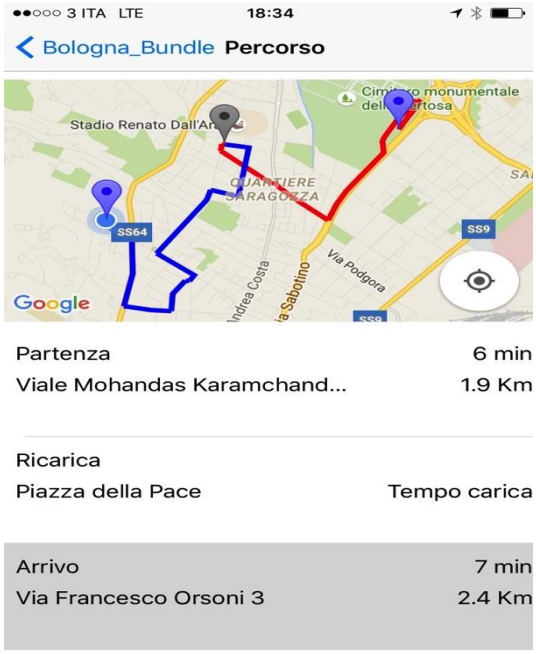


Figura 4.13: Fine dell'itinerario

Capitolo 5

Conclusione e Sviluppi Futuri

Il progetto è stato testato molte volte durante la scrittura del codice, all'interno dell'ambiente di sviluppo utilizzando il simulatore integrato. Conclusa, è stata testata anche su dispositivo fisico. In entrambi i test non si sono verificati particolari malfunzionamenti del software, ogni tanto si sono presentati alcuni bug utilizzando le sdk di Google Maps. Per ovviare a questo problema si sarebbe potuto utilizzare il framework Map Kit nativo del linguaggio in sostituzione a quello di Google ma nelle mappe di Apple sono presenti minori dettagli.

L'applicazione da noi creata fa uso di vari servizi offerti da Google, le Directions API vengono utilizzate per assicurare l'efficiente calcolo del percorso più breve tra due punti sulla mappa e il servizio di geocoding converte coordinate geografiche in indirizzi espressi in formato comprensibile all'utente e viceversa. L'utilizzo di questi in modo gratuito purtroppo è limitato ad un massimo di 2500 richieste al giorno per entrambi, pagando una quota ogni 1000 richieste aggiuntive esiste la possibilità di ampliare queste fino a 100.000 al giorno. Il numero di richieste

giornaliere è stato più che sufficiente per testare il progetto in tutte le sue funzionalità ma il problema potrebbe essere di fondamentale importanza se in un futuro si volesse richiedere il rilascio sull'App Store dell'applicazione da noi creata; il limite più grave sarebbe quello di non riuscire a rispettare il massimo delle richieste se questa iniziasse ad essere utilizzata da un bacino d'utenza abbastanza ampio. Per ovviare a questo problema si dovrebbe creare un servizio personalizzato per il calcolo dei percorsi più brevi, nello specifico per calcolare itinerari da completare utilizzando mezzi pubblici. Prima della scelta di utilizzare le Directions API, si è pensato di creare un algoritmo che restituisca il percorso più breve per giungere a destinazione tramite l'utilizzo di trasporti pubblici. Dopo aver visionato svariate informazioni online e progetti open source non si è trovato niente di intuito da cui cominciare e soprattutto che non facesse perdere più tempo di quello definito per la realizzazione dell'intero progetto.

All'interno dell'applicazione, oltre alla possibile realizzazione di un algoritmo di routing personalizzato, esistono svariati sviluppi e margini di miglioramento che possono essere attuati in futuro. Il sito web gestito da TPER offre ulteriori dati da ispezionare relativi alle fermate degli autobus, aggiungendoli all'interno dell'applicazione aumenterebbero le informazioni disponibili per ogni fermata. Con la pressione di un'annotazione sulla mappa, oltre al nome della fermata e all'ubicazione, potrebbero essere visualizzate tutte le linee passanti da questa. Inoltre c'è la possibilità di importare tutti gli orari feriali e festivi delle relative linee, grazie a questo potrebbe essere realizzato un menù compreso di tutte le linee e selezionando una tra queste verrebbe visualizzato sulla mappa tutto il percorso che svolge la linea, visualizzando in primo piano solo le fermate appartenenti a questa, unite tra di loro utilizzando degli overlay. Questa opzione sarebbe molto utile per l'utente che si volesse informare sugli orari delle varie linee, magari per pianificare un tragitto futuro. Potrebbe essere migliorata anche la modalità riguardante i veicoli elettrici, oltre alla visualizzazione delle colonnine di ricarica potrebbero essere rappresentati sulla mappa servizi di car sharing, se presenti, che utilizzano veicoli ecologici, soprattutto per sfruttare la condivisione di una macchina per entrare nelle zone a traffico limitato (ZTL).

Bibliografia

[1] The Swift Programming language (Swift 2 edition), Apple inc.

[2] Developing iOS 8 Apps with Swift, Stanford iTunes u course, Core Location and Map Kit chapter.

[3] Google Maps iOS sdk Developer website:
https://developers.google.com/maps/documentation/ios-sdk/?hl=en

[4] Location and Maps Programming Guide, iOS Developer Library:
https://developer.apple.com/library/ios/navigation/#section=Frameworks&topic=MapKit

[5] Google Maps Directions API:
https://developers.google.com/maps/documentation/directions/

[6] Google Geocoding API:
https://developers.google.com/maps/documentation/geocoding/intro

[7] Core location Guide, iOS Developer Library:
https://developer.apple.com/library/ios/navigation/#section=Frameworks&topic=CoreLocation

[8] Core Data Guide, iOS Developer Library:
https://developer.apple.com/library/ios/navigation/#section=Frameworks&topic=CoreData

[9] Sezione open data TPER:
http://www.tper.it/tper-open-data

[10] Pubblicazione dataset Bologna:
http://www.dati.comune.bologna.it

[11] servizio “mobi” comune di bologna:
http://www.comune.bologna.it/trasporti/servizi/2:3026/5843/