ITIS “E. Mattei”

ANNO SCOLASTICO 2015/2016

CLASSE 5^D/E

INDIRIZZO INFORMATICA

SARplay

Un’applicazione nativa per iPad che mostra la deformazione del terreno sulla base di dati satellitari

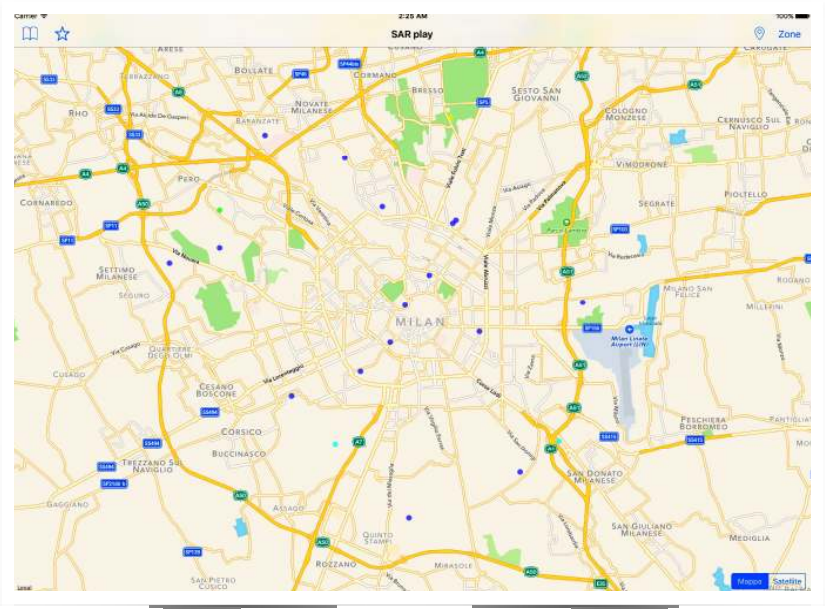
Sommario

Introduzione

SARplay nasce grazie a due eventi che si sono svolti presso il nostro istituto, durante l'anno scolastico 2015/2016:

1. la collaborazione della scuola con la software house "Visuality srl", la quale, in cerca di una sede a Morbegno e di nuovo personale, si è offerta di istruire e trasmettere la sua esperienza nel mondo Apple verso gli studenti interessati;

2. il coinvolgimento dell'ex professore di Informatica Sabatino Buonanno, il quale, dopo aver insegnato per alcuni anni presso l'I.T.I.S. di Sondrio e il Liceo Scientifico "C. Donegani", è tornato a Napoli, dove attualmente lavora per il CNR (Consiglio Nazionale delle Ricerche), e si occupa di elaborazione di immagini satellitari, riguardanti la deformazione della crosta terreste nel tempo.



L'istituto IREA (Istituto per il Rilevamento Elettromagnetico dell'Ambiente, dipartimento del CNR) ha il compito di raccogliere ed elaborare i dati dei satelliti, i quali vengono poi rappresentati su una mappa, in maniera che essa possa contenere tutti i punti analizzati. La mappa viene messa a disposizione chiedendo l'autorizzazione al CNR, ed è consultabile tramite un web server; tuttavia, la fluidità dell'interfaccia web dipende molto dalla connessione di cui si dispone ed inoltre è parecchio complicata da utilizzare; pertanto un utente medio non sarebbe in grado di farne un buon uso. L'unione di queste due

problematiche ci ha spinti a pensare ad un'app per iPad semplice ed intuitiva, da realizzare insieme a Visuality Software e con la collaborazione dell'IREA.

Conformazione dell'app

L'app è formata dai seguenti elementi:

- connettore, ovvero una classe che si interpone tra server e applicazione; il suo compito è quello di scaricare i dati delle deformazioni e renderli disponibili in modo leggibile alla app, tramite una serie di richieste http
- database, contenente tutti i punti scaricati dal server tramite il connettore. È formato dalle entità "point", "zone" e "deformation"
- mappa, contenuta nella view principale, nella quale sono visibili i punti di deformazione in base alla zona selezionata. È possibile scegliere due tipi di mappa diversi (satellitare e carta politica)
- posizione corrente, in modo che la mappa visualizzi i punti di deformazione della zona in cui si trova l'utente
- menu delle zone disponibili, costituito da una tabella contenente tutte le zone messe a disposizione dal connettore
- menu dei preferiti, costituito anch'esso da una tabella contenente tutte le zone visitate maggiormente dall'utente. Per poter inserire una zona tra i preferiti, si può utilizzare il bottone apposito, rappresentato da una stella

Raccolta, elaborazione e visualizzazione dei dati lato web

Per poter visualizzare la deformazione del terreno nelle zone interessate, i dati devono essere raccolti, elaborati e visualizzati:

Il processo si suddivide in cinque fasi:

1. l'acquisizione dei dati è affidata al satellite, il quale esegue i rilevamenti tramite l'uso di radar SAR.
   I sensori SAR sono associati a specifiche bande dello spettro elettromagnetico. Nelle applicazioni InSAR le bande comunemente utilizzate sono la banda L (frequenza 1-2 GHz, λ ~24 cm), la banda C (frequenza 5-6 GHz, λ ~6 cm) e la banda X (frequenza 8-12 GHz, λ ~3 cm). Il principio di funzionamento di questi sensori è il seguente: un'antenna trasmittente propaga nello spazio un' onda elettromagnetica che, incidendo sulla superficie terrestre, subisce un fenomeno di riflessione. Una parte del campo diffuso torna verso la stazione trasmittente, equipaggiata per la ricezione, dove vengono misurate le sue caratteristiche.
   Il dispositivo è in grado di individuare il bersaglio elettromagnetico (funzione di detecting) e, misurando il ritardo temporale tra l'istante di trasmissione e quello di ricezione, valutare la distanza a cui è posizionato, localizzandolo in modo preciso lungo la direzione di puntamento dell'antenna (direzione di range).Il segnale radar relativo ad un bersaglio è caratterizzato da due valori: ampiezza e fase.Questi valori permettono di realizzare due immagini. La fase in particolare racchiude l'informazione più importante ai fini delle applicazioni interferometriche;

2. dopo aver raccolto i dati questi verranno inviati ai centri di trasmissione mediante l'uso di antenne satellitari. Queste faranno da intermediari con i server del CNR ai quali saranno inviate tutte le informazioni in formato immagine in scala di grigi ancora da elaborare;

3. il compito dei server è quello di elaborare le immagini satellitari ed estrapolare dati concreti.
   L'elaborazione consiste nel creare un'immagine di interferometria sovrapponendo due o più immagini in scala di grigi fornite dal satellite. L'interferometria è la misurazione delle variazioni della fase del segnale SAR tra due acquisizioni distinte. Questo intero processo, totalmente automatizzato tramite l'uso di programmi in C, è realizzato tramite l'utilizzo di una architettura hardware per elaborazioni in parallelo, chiamata CUDA (Compute Unified Device Architecture), sviluppata da NVIDIA. Essa permette di raggiungere alte prestazioni di computing grazie alla potenza di calcolo delle GPU. Nel caso analizzato vengono utilizzate schede video tesla k20 in un cluster di 23 computer;

4. i dati elaborati vengono quindi caricati in formato compatibile con gli standard OGC (Open Geospatial Consortium) su di un database collegato a

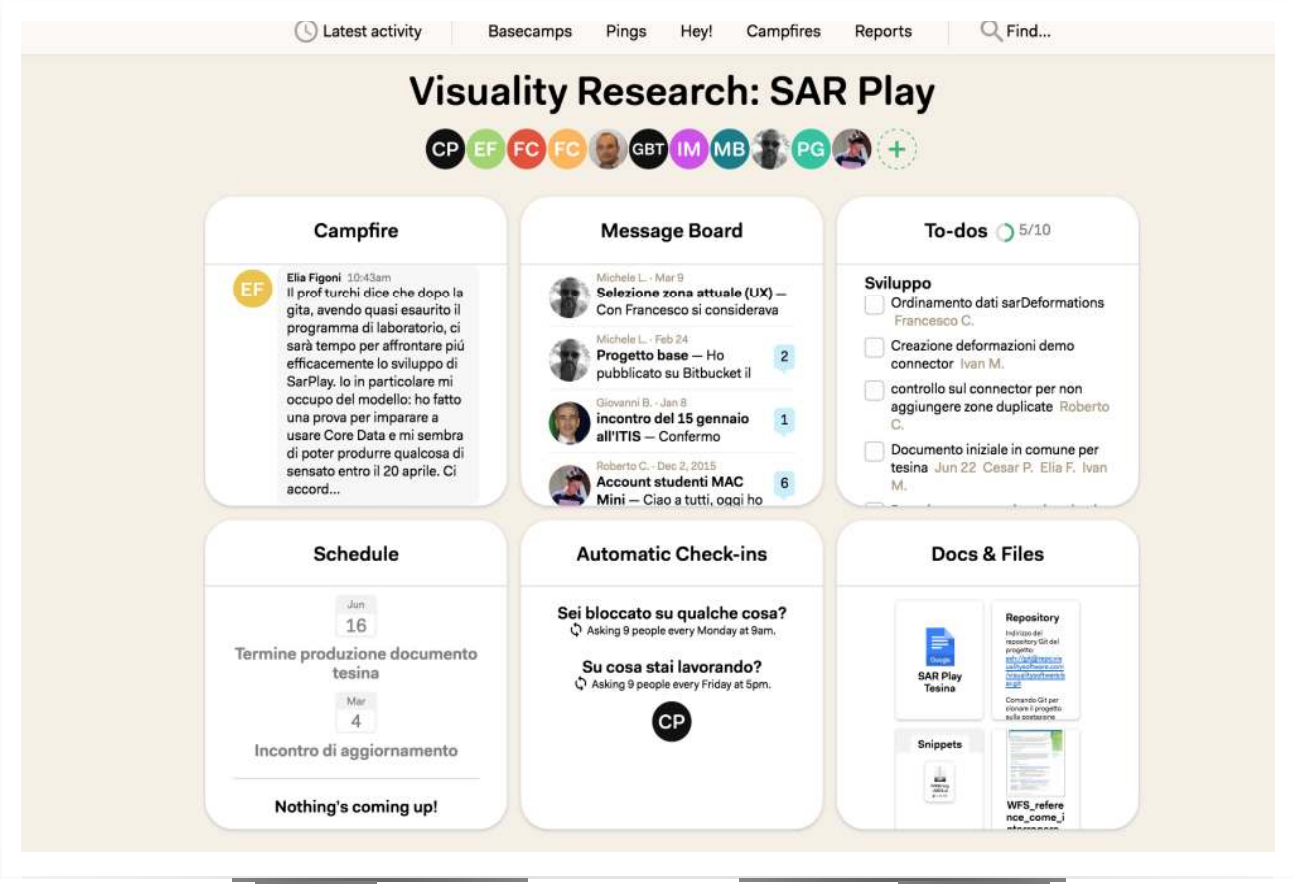
un’applicativo web chiamato GeoServer. Quest’ultimo non fa nient’altro che porsi come interfaccia tra i dati e l’utente;

5. tramite internet e con l’utilizzo di un browser si è in grado di collegarsi al GeoServer gestito dal CNR e visualizzare così su una mappa i dati sotto forma di punti di deformazione. Tramite diverse gradazioni di colore si può capire il movimento del terreno, ed è possibile ispezionare lo storico di deformazione di uno specifico punto.

Metodologia e ambiente di lavoro

Basecamp - Collaborazione e organizzazione del team

Dal momento che SARplay nasce come progetto di gruppo, sin da subito è nata la necessità di trovare un modo per rimanere in contatto. Oltre ai numerosi incontri tenuti durante l'anno, sia con i membri di Visuality, sia tra noi ragazzi, permaneva comunque il problema di tenersi aggiornati frequentemente,utilizzando mezzi che non fossero le email. Nel nostro caso, Visuality software usa da tempo Basecamp (basecamp.com): un potente strumento web per lavorare in team. Pertanto è stato creato uno spazio apposito per il progetto (un basecamp, appunto).



Le principali funzionalità di basecamp sono:

- Ping: sono dei messaggi tra due o più utenti, molto comodi per chiarire, specificare o chiedere concetti in maniera immediata; il funzionamento è lo stesso di ogni programma di chat;
- Hey: è una schermata contenente tutte le notifiche;
- Campfire: è una chat tra tutti i membri del gruppo;
- To-dos: è una schermata che contiene tutte le cose da fare. Ogni utente può creare dei to-do, e delegare ad un altro utente il compito; una volta terminato il compito, è possibile archiviarlo;
- Schedule: contiene tutte le scadenze dei to-dos. È possibile visualizzare tutti i to-dos di tutti i mesi futuri, oltre a quelli già scaduti;
- Check-ins: sono delle domande programmate che vengono inviate all'utente costantemente. Una domanda può essere "Sei bloccato su qualcosa?", oppure "Su cosa stai lavorando?"; in base alla risposta inviata, si possono ricevere chiarimenti, oppure si può organizzare ancora meglio il lavoro rimanente;
- Docs & files: è uno spazio nel quale vengono caricati file utili per il progetto corrente.

Tuttavia, il codice non viene caricato su basecamp, in quanto è scomodo e quasi impossibile tenere traccia di tutti gli aggiornamenti: per questo Visuality usa BitBucket (http://bitbucket.org) come servizio di source control & versioning. BitBucket, fratello di GitHub, è un repository hosting basato su git.

Git & BitBucket - Source control & versioning

Git è in assoluto il sistema di versioning più utilizzato da parecchio tempo. Ma cosa vuol dire versioning?
Per versioning si intende un sistema in grado di tenere traccia di tutti i cambiamenti avvenuti ad uno o più files nel tempo, così da poterne recuperare una versione precedente in qualunque momento, capire come è mutato un progetto durante il suo corso, sapere chi ha modificato qualcosa e quando.
Lavorare appoggiandosi a git significa che qualsiasi errore o malfunzionamento introdotto dal programmatore può essere ripristinato in pochi secondi.
Per usare questo sistema si può scegliere di usare il terminale e dare i vari comandi operativi tramite CLI oppure di appoggiarsi a vari programmi che mettono a disposizione una GUI di facile utilizzo.

L'esempio riportato raffigura Tower. Sulla sinistra i branch e lo spazio di lavoro, al centro la lista di tutti i commit e sulla destra il dettaglio delle modifiche relative al commit selezionato.

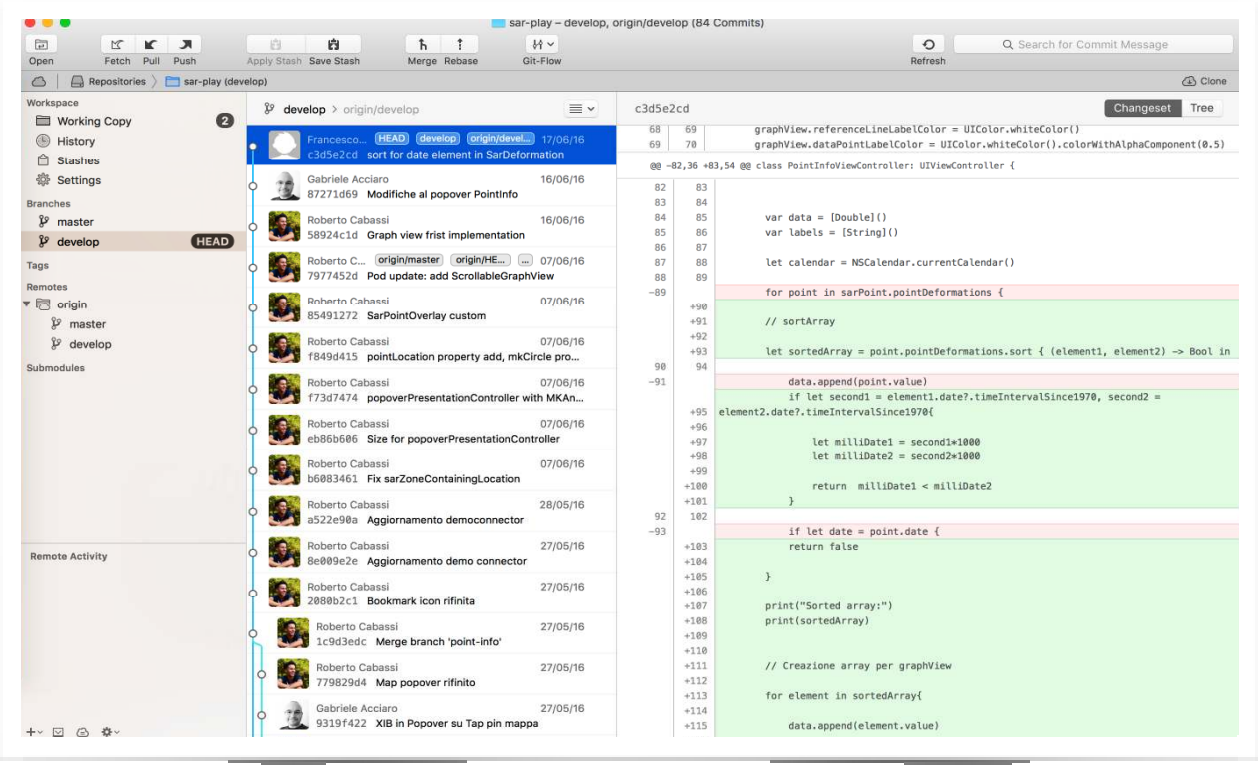
Git considera i propri dati come una serie di istantanee (snapshot) di un mini filesystem. Ogni volta che l'utente effettua un commit, git salva lo stato del proprio progetto: fondamentalmente fa un'immagine di tutti i file con modifiche presenti in quel momentosalvando un riferimento allo snapshot. Se alcuni file non sono stati modificati, git non li clona ma crea un collegamento agli stessi della versione precedente.

Un progetto git è composto dai seguenti elementi:

- Working dir o directory di lavoro che contiene i file appartenenti alla versione corrente del progetto sulla quale l'utente sta lavorando;
- Index o Stage che contiene i file in transito, cioè quelli candidati ad essere committati;
- Head che contiene gli ultimi file committati.

É possibile inizializzare unprogetto git in due modi:

- definire un progetto preesistente come gitrepository;
- clonare un repository git esistente da un server.



Storico dei commit, il comando log

Mediante il comando log è possibile visualizzare l'elenco degli ultimi commit effettuati. Ciascun commit è contrassegnato da un codice SHA-1 univoco, la data in cui è stato effettuato e tutti i riferimenti dell'autore. Il comando, lanciato

senza argomenti, mostra i risultati in ordine cronologico inverso, quello più recente è mostrato all'inizio.
Naturalmente sono disponibili numerosi argomenti opzionali utilizzabili con il comando log che permettono di filtrare l'output.

```
>> git log
```

Stato dei file, il comando status

Mediante il comando status, è possibile analizzare lo stato dei file. Git ci indicherà quali sono i file modificati rispetto allo snapshot precedente e quali quelli già aggiunti all'area STAGE.

```
>> git status
```

Il modus operandi più diffuso è il seguente:

- fare pull (git pull) per ottenere le ultime modifiche dal server
- aggiornare il progetto effettuando le modifiche desiderate
- fare il commit delle modifiche (git commit -m "commento")
- fare push per pubblicare le proprie modifiche e renderle disponibili per chiunque (git push)

Oltre alle funzionalità base qui elencate, git offre moltissime altre features; per maggiori informazioni si rimanda al sito di supporto di git (www.git-scm.org).

Xcode - Integrated Development Environment

Apple, per tutti i suoi dispositivi ha un solo IDE, Xcode per l'appunto.
Esso contiene - come ci si aspetta da un IDE - tutti gli strumenti per lo sviluppo, il testing e la distribuzione del software.
Xcode permette la creazione di applicazioni per iPhone, iPad, Mac, appleWatch e appleTV.

In particolare vi sono due grandi famiglie di framework su cui Xcode permette di lavorare:

- Cocoa
- Cocoa touch

Cocoa è costruito per macOS e Cocoa touch per iOS, tvOS e watchOS.
Siccome un approfondimento di Xcode sarebbe lungo oltremodo, per le specifiche si rimanda a Apple Developer (developer.apple.com)

Swift - Il linguaggio di programmazione

Come per ogni progetto software, anche noi per la realizzazione di SARplay abbiamo dovuto scegliere un linguaggio di programmazione.
Avendo scelto Apple come ambiente di sviluppo a priori le scelte ricadono su Objective-C e Swift.
Obj-C è nato negli anni ’80 ed è sempre stato utilizzato da Apple come unico linguaggio di programmazione proveniente dal C. Nel 2014 è nato Swift, questa volta completamente e interamente progettato da Apple per essere veloce, moderno, interattivo e sicuro.

Le novità introdotte da Swift sono così tante che è quasi impossibile elencarle tutte; è da notare come Swift erediti caratteristiche da tantissimi linguaggi di programmazione prendendo il meglio da ognuno di essi.
Un esempio eclatante è che in Swift un errore del tipo NullPointerException che tutti i programmatori Java conoscono (e odiano) non esiste. Questo perché effettuare una chiamata su di un oggetto nullo non è possibile. Non perché gli oggetti non saranno mai nulli, ma semplicemente perché è impossibile scrivere del codice che possa effettuare una chiamata di questo tipo.
Altro caso è quello delle Extension: questo tipo di peculiarità unica nel suo genere permette di aggiungere funzionalità ad una classe, una struttura, una enumerazione o un protocollo. Questo include la possibilità di estendere tipi per il quale non si ha accesso al codice sorgente originale (questa pratica viene chiamata retroactive modeling).
Per le specifiche si rimanda a swift.org e a Apple Developer.

A seguito della suddivisione del lavoro, mi è stata assegnata la realizzazione del connettore, parte fondamentale in quanto è responsabile di mettere a disposizione i dati da consultare.

SARplayConnector

Il connettore, come accennato in precedenza si occupa di scaricare i dati dai server del CNR e di passarli al modello che in seguito li salverà nel database interno all'applicazione. Per arrivare alla realizzazione dello stesso ho dovuto affrontare diversi passi:

- capire come accedere al GeoServer
- studiare lo standard WFS (Web Feature Service)
- trovare un formato dati fornito dalle API di GeoServer compatibile e il più semplice da "parsare"
- studiare le tecnologie messe a disposizione da Apple per lo scaricamento di dati in background
- codificare il connettore

GeoServer - Accesso ai dati

GeoServer (applicativo web utilizzato dal CNR per la visualizzazione dei dati) utilizza due protocolli per funzionare:

- WMS: Web Map Service
- WFS: Web Feature Service

WMS è il protocollo il cui compito è quello di scaricare e rappresentare la mappa vera e propria, sia essa di tipo satellitare, politica, fisica, geologica ecc. In sostanza scarica le tiles che compongono la mappa e le dispone nello spazio di visualizzazione in maniera georeferenziata. E' immaginabile come una sorta di agente che si occupa di comporre un puzzle per poi mostrare il risultato finale incorniciato in un quadretto.

WFS invece è il protocollo che si occupa di scaricare e posizionare sulla mappa punti, immagini, polilinee, geofence, cerchi, forme di vario genere e tutto ciò che di visibile si possa sovrapporre a una mappa.

WFS - Web Feature Service

Le deformazioni del terreno su GeoServer vengono rappresentate tramite un dodecagono con contorno colorato per capire il tipo di spostamento. Questi dodecagoni sono subset di feature. Questo vuol dire, informaticamente parlando, che un dodecagono eredita da feature e implementa nuove caratteristiche e proprietà.

Questo processo avviene per tutti gli elementi grafici rappresentabili con WFS. WFS si interfaccia con un database Postgres da cui ottiene i dati necessari per costruire graficamente la feature e posizionarla sulla mappa. Questo avviene tramite una richiesta HTTP asincrona verso il GeoServer che restituirà un documento XML mediante la tecnologia AJAX.
La richiesta è del tipo:

```
http://server.com/geoserver/wfs?
  service=wfs&
  version=2.0.0&
  request=GetFeature&
  typeNames=namespace:<Nome della zona>
```

Questa query può essere usata per ottenere tutte le features contenute in una determinata zona specificando l'identificatore della zona stessa. Si può inoltre specificare il formato dei dati se non lo si desidera in XML.
Altri formati supportati sono:

1. JPEG image
2. PDF document
3. PNG image
4. Zipped Shapefile
5. GML 2.0 Format
6. GML 3.1.1 Format
7. CSV Format
8. Excel Format
9. GeoJSON Format
10. KML Format
11. Google Earth Format
12. Tiles Format

A priori ho scartato tutto a parte GML 2.0, GML 3.1.1 e GeoJSON. GML è un derivato dell' XML e GeoJSON è un derivato del JSON. Questi ultimi sono i due formati più semplici da elaborare e interpretare.
Di questi ho scelto il formato GeoJSON (www.json.org per le specifiche) e ho scaricato l'area di Roma per effettuare dei test. Il file pesa 417 MB.

Ecco un esempio del file (ovviamente tagliato):

```
// Begin JSON Document
{
"type":"FeatureCollection",
"totalFeatures":2,
"features":
       [
              {
                     "type":"Feature",
                     "id":"csk_roma.fid-68b99caf_1522fa8eedd_-faf",
                     "geometry":
                     {
                            "type":"Polygon",
                            "coordinates":
                            [
                                   [
                                          [12.2461841605744,41.7362017017566],
                                          [12.2460957779932,41.7361792358623],
                                          [12.2460042395264,41.736192879839],
                                          [12.2459340727034,41.7362389778141],
                                          [12.2459040786496,41.7363051779241],
                                          [12.2459222943581,41.7363737419347],
                                          [12.2459838390735,41.7364262981546],
                                          [12.2460722219571,41.7364487641356],
                                          [12.246163760776,41.7364351201044],
                                          [12.2462339276487,41.736389021988],
                                          [12.2462639214001,41.7363228217914],
                                          [12.2462457053395,41.7362542578353],
                                          [12.2461841605744,41.7362017017566]
                                   ]
                            ]
                     },
                     "geometry_name":"the_geom",
                     "properties":
                     {
                            "id":1,
                            "plot":"<a href=\"#\"
onClick=\"window.open('http://IP_ADDRESS/static/plot.php?id=1&table=csk_roma_0','PLOT','w
idth=700, height=400, location=no','_blank')\">Plot data<\/a>",
                            "east":270977.84,
                            "north":4624165.5,
                            "lat":41.736313,
                            "lon":12.246084,
                            "azimuth":354,
                            "range":0,
                            "coherence":0.844166,
                            "velocity":0.158807,
                            "elevation":5.09221,
                            "z2010_5115":0,
                            "z2010_5334":-0.0446284,
                            "z2010_5581":-0.349941,
                            "z2010_5773":-0.00378048,
                            "z2010_5976":0.175714,
                            "z2010_6195":0.0349877,
                            "z2010_6864":0.233866,
                            "z2010_7275":-0.113367,
                            "z2010_7752":-0.221647,
                            "z2010_8191":-0.265989,
                            "z2011_2248":0.312297,
                            "z2011_2670":0.0925604,
                            "z2011_3108":-0.280589,
                            "z2011_3558":-0.20474,
                            "z2011_3997":-0.194254,
                            "z2011_4419":0.210427,
                            "z2011_4857":0.423169,
                            "z2011_5307":0.41748,
                            "z2011_6606":0.408881,
                            "z2011_7028":-0.207416,
```

```
                        "z2011_7467":0.0135972
                }
        },

        {
                "type":"Feature",
                "id":"csk_roma.fid-68b99caf_1522fa8eedd_-fae",
                "geometry":
                {
                        "type":"Polygon",
                        "coordinates":
                        [
                                [
                                        [12.3665579379854,41.7292305959677],
                                        [12.3664695228108,41.7292082235154],
                                        [12.3663780201262,41.7292219642424],
                                        [12.3663079478732,41.7292681363521],
                                        [12.3662780818378,41.7293343681167],
                                        [12.3662964247108,41.7294029128199],
                                        [12.366358061663,41.7294554039439],
                                        [12.3664464771406,41.7294777764826],
                                        [12.3665379801768,41.7294640357007],
                                        [12.3666080524784,41.7294178634497],
                                        [12.3666379182108,41.7293516315988],
                                        [12.3666195749861,41.7292830869504],
                                        [12.3665579379854,41.7292305959677]
                                ]
                        ]
                },
                "geometry_name":"the_geom",
                "properties":
                {
                        "id":1,
                        "plot":"<a href=\"#\"
onClick=\”window.open('http://IP_ADDRESS/static/plot.php?id=1&table=csk_roma_1','PLOT','w
idth=700, height=400, location=no','_blank')\">Plot data<\/a>",
                        "east":280965.5,
                        "north":4623078,
                        "lat":41.729343,
                        "lon":12.366458,
                        "azimuth":276,
                        "range":279,
                        "coherence":0.972302,
                        "velocity":-0.00610888,
                        "elevation":9.30568,
                        "z2010_5115":0,
                        "z2010_5334":0.00768796,
                        "z2010_5581":0.0592088,
                        "z2010_5773":0.0775075,
                        "z2010_5976":0.0660014,
                        "z2010_6195":-0.274582,
                        "z2010_6864":0.235149,
                        "z2010_7275":0.00518118,
                        "z2010_7752":-0.54853,
                        "z2010_8191":-0.954134,
                        "z2011_2248":-0.222902,
                        "z2011_2670":-0.00346928,
                        "z2011_3108":0.0973639,
                        "z2011_3558":0.128939,
                        "z2011_3997":-0.195534,
                        "z2011_4419":0.0562914,
                        "z2011_4857":0.156633,
                        "z2011_5307":-0.173955,
                        "z2011_6606":-0.330861,
                        "z2011_7028":-0.308144,
                        "z2011_7467":0.0955062
                }
        }
    ]
}

// End JSON Document
```

Si ha quindi una collezione di features: ogni features come già accennato corrisponde a un pallino sulla mappa.

Ogni pallino ha dei dati che lo caratterizzano:

- type: indica il tipo di oggetto, che è una feature
- id: identifica la feature in maniera univoca
- geometry: è un oggetto che contiene:
  - type: è sempre Polygon a quanto ho potuto vedere ed è il luogo geometrico della feature (infatti si può osservare che in realtà non sono circonferenze quelle visualizzate sulla mappa, bensì dodecagoni come precedentemente accennato)
  - coordinates: è l'insieme dei punti che definisce il poligono
- geometry_name: è il nome della geometria che nello standard GeoJSON è sempre "the_geom"

- properties: è un oggetto che contiene:
  - id: è ciò che identifica l'oggetto properties, ma visto che ogni feature ha solo un oggetto properties è sempre 1
  - plot: è l'indirizzo URL del grafico di deformazione del terreno che mostra l'andamento nel tempo
  - east: riferimento in gradi rispetto all'origine X della zona
  - north: riferimento in gradi rispetto all'origine Y della zona
  - lat: indica la latitudine della feature
  - lon: indica la longitudine della feature
  - azimut
  - range
  - coherence: è un indice di affidabilità dei dati
  - velocity: è la velocità di spostamento misurata
  - elevation: è l'elevazione AMSL della feature
  - una serie di campi zAAAA_xxxx. AAAA indica l'anno in 4 cifre, xxxx sono le ore passate dall'inizio dell'anno indicato. Sono i dati utilizzati per costruire il plot

Un esempio di plot rappresentante i dati della prima feature nel documento JSON qui sopra:

JSON parsing

Dopo aver interpretato la struttura del file GeoJSON, la prosecuzione naturale dello sviluppo è stata quella di scrivere del codice che fosse in grado di analizzare il file, che nient'altro è se non una stringa, attraverso le API JSON di Apple e successivamente creare delle entità per il database.

Il file dopo essere stato scaricato viene salvato in maniera temporanea nella Document directory della applicazione per poi essere serializzato. Di seguito uno snippet di codice che mostra la funzione che carica il file JSON in maniera uncached:
Questo tipo di lettura permette una velocità maggiore in quanto il file non verrà caricato totalmente in memoria per poi essere serializzato, bensì verrà letto un carattere alla volta e trasformato in un NSDictionary del tipo [String: AnyObject].

Ottenuto il dizionario, questo viene analizzato e confrontato con un pattern. Per pattern si intende una struttura in cui viene indicato il tipo di dati che un'implementazione della stessa deve contenere. E' possibile paragonarla alla dichiarazione di una struttura in C mediante typedef struct {}.
Se il confronto con il pattern corrisponde vuol dire che è possibile creare un oggetto di tipo feature dal record contenuto nel dizionario e pertanto esso viene creato e inserito in un array che successivamente verrà passato al modello.

```
private func loadJSON() -> [String: AnyObject]? {

    do {

        guard let url = NSBundle.mainBundle().URLForResource("loadedZone", withExtension: "json")

        else {
                print("Error loading json file from resource")
                return nil
        }

        let data = try NSData(contentsOfURL: url, options: .DataReadingUncached)

        let object = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments)

        if let dict = object as? [String: AnyObject] {

            return dict
        }
    }
    catch {
        print("Error parsing json: \(error)")
    }

    return nil
}
```

Scaricamento dati in background

L'ultimo problema della lista e forse il più importante, è stato scaricare effettivamente i files delle zone sul dispositivo. Data la dimensione media di una zona (~500MB) è chiaro come la app debba necessariamente effettuare questo scaricamento in maniera asincrona sfruttando un thread separato e soprattuto che il download possa proseguire anche dopo che la app sia passata dallo stato di foreground a quello di background.
IOS a differenza di Android, quando una app passa in background la congela istantaneamente. Questa operazione viene effettuata per ovvi motivi di risparmio energetico. E' difatti questo comportamento uno dei motivi per il quale un iPhone ha la stessa autonomia di un terminale Android di fascia alta che però monta una batteria con capacità doppia.
Apparentemente non sembrerebbe possibile effettuare operazioni in background in quando la app è congelata. Apple ha deciso di introdurre delle eccezioni

chiamate Background Modes. Queste funzionalità permettono di eseguire le più svariate operazioni in una coda a bassa priorità del sistema operativo senza interferire con altre operazioni che l'utente sta effettuando.
In SARplay ci serviamo della Background Mode chiamata Background fetch. Quest'ultima ci permette di scaricare il file da internet ed eseguire le operazioni di parsing precedentemente descritte.

Viene dunque configurata una NSBackgroundSessionConfiguration che permettere di eseguire una NSURLSession asincrona e in background:
Una volta configurata la sessione, il procedimento continua analogamente ai download con app in foreground, cioè facendo partire la sessione e utilizzando un Delegate per intercettare e gestire l'evento di fine download.
Alla fine del download, la NSURLSession chiamerà dunque il Delegate che a sua volta si preoccuperà di prendere il file JSON scaricato sotto forma di NSData e salvarlo nella Document directory.

```
let sessionIdentifier =  "url_session_background_download"

var configuration = NSURLSessionConfiguration.backgroundSessionConfigurationWithIdentifier
    (sessionIdentifier)
```

Database & Persistent storage

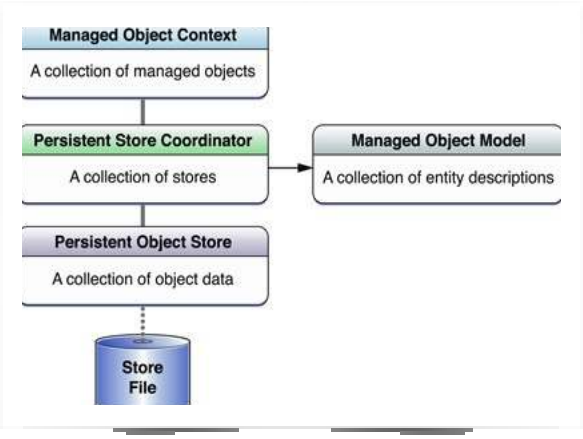
Core Data

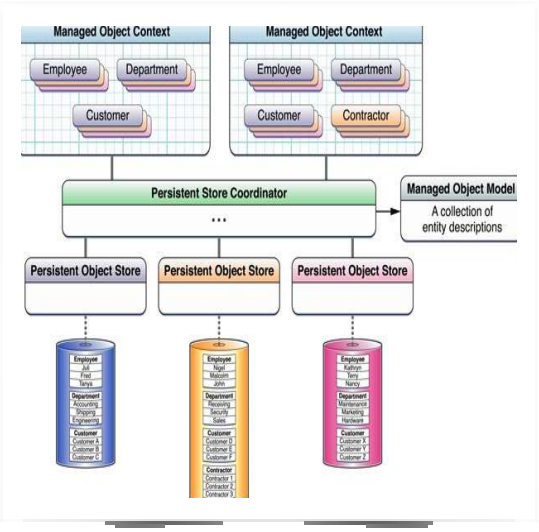
Core Data è un framework di Apple che permette la gestione del ciclo di vita e della persistenza degli oggetti/entità di un'applicazione. É basato su SQLite, ma non è un database relazionale, infatti **è uno strumento intermedio che fa da tramite tra l'applicazione e la memorizzazione dei dati in memoria che può avvenire in tre modi:** XML, binario o SQLite. Per la memorizzazioneCore Data utilizza di default un database di tipo Relazionale(SQLite).

Organizzazione di core data

Una delle caratteristiche principali di Core Data è la sua organizzazione a livelli che va a definire quello che è comunemente chiamato Core Data Stack. Esso è composto da:

1. Context
2. ManagedObjects
3. Store Coordinator
4. Object Store





NSManagedObjectContext

L'NSManagedObjectContext **si occupa della gestione delle informazioni contenute nella memoria** dell'applicazione, salvate tramite le funzionalità del Core Data. Esso si può pensare come un pacchetto che contiene tutte le istanze dei vari NSManagedObjects. Inoltre il Context

mantiene lo status degli oggetti e gestisce le loro relazioni fino a quando l'utente non specifica di salvare i cambiamenti permanentemente.

```
lazy var managedObjectContext: NSManagedObjectContext = {


        let coordinator = self.persistentStoreCoordinator

        var managedObjectContext = NSManagedObjectContext(concurrencyType:
.MainQueueConcurrencyType)
        managedObjectContext.persistentStoreCoordinator = coordinator

        return managedObjectContext
    }()
```

Questa istruzione permette di richiamare il context da qualsiasi classe dell'applicazione

```
let context = SARplayModel.sharedInstance.managedObjectContext
```

Managed Objects

Gli Object sono istanze della classe NSManagedObject e possono essere rappresentati come dei records in un tabella di un database relazionale. Essi sono simili agli NSManagedObject di Swift ma con proprietà specifiche per Core Data.
Il Context permette di effettuare su di essi operazioni di ricerca, inserimento, modifica ed eliminazione.

Store Coordinator

Lo Store Coordinator è l'elemento che si occupa di gestire una collezione di negozi(Store).
In parole povere è il responsabile della coordinazione degli accessi a multipli "Persistent Object Store".

ObjectStore

Rappresenta lo strato più basso della pila di Core Data, nel quale i dati vengono memorizzati. Vengono supportate le codifiche XML, binaria e SQLite. Lo sviluppatore non interagirà mai direttamente con questo oggetto.
Nella maggior parte dei casi si ha solo un ObjectStore.
Esistono due tipi di "Negozi": uno di questi è un negozio in memoria, dove è possibile utilizzare Core Data completamente in memoria. Il secondo è il negozio persistente, istanza di un oggetto NSPersistentStore, e che rappresenta il negozio reale su disco.

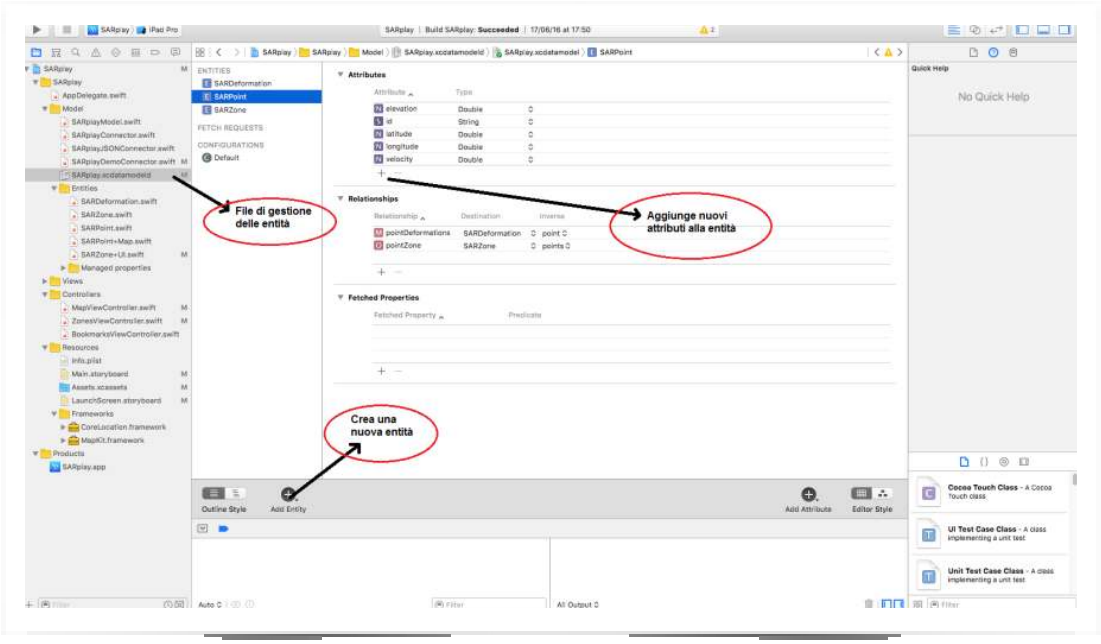
Componenti di Core data

Core Data nella gestione della memorizzazione dei dati necessita della definizione di due elementi fondamentali: le entità e le relazioni. Per la loro gestione crea automaticamente un interfaccia, ossia il file .xcdatamodeld, che

permette di definirli e organizzare quello che è il modello del database su cui poi andrà ad effettuare le operazioni da noi specificate.

Entità

Un Model, o meglio conosciuto come entità(Entity), è un prototipo di ciò che verrà inserito e salvato nella memoria del dispositivo grazie a Core Data. Una Entity si può considerare come una classe astratta che serve a raggruppare oggetti aventi caratteristiche comuni. Ogni entità possiede i propri attributi, ossia gli elementi che permettono di definirla e di distinguere due istanze della stessa.
Nell'immagine sotto rappresentata troviamo gli elementi che costituiscono un'entità e le modalità per crearla.



Le entità che verranno definite in Core Data, per essere tramutate in istanze di una determinata classe, necessitano di essere allocate tramite la creazione di una classe che implementa un costruttore per i suoi attributi. Quella che si andrà a realizzare non sarà che una sottoclasse della NSObjectClass.

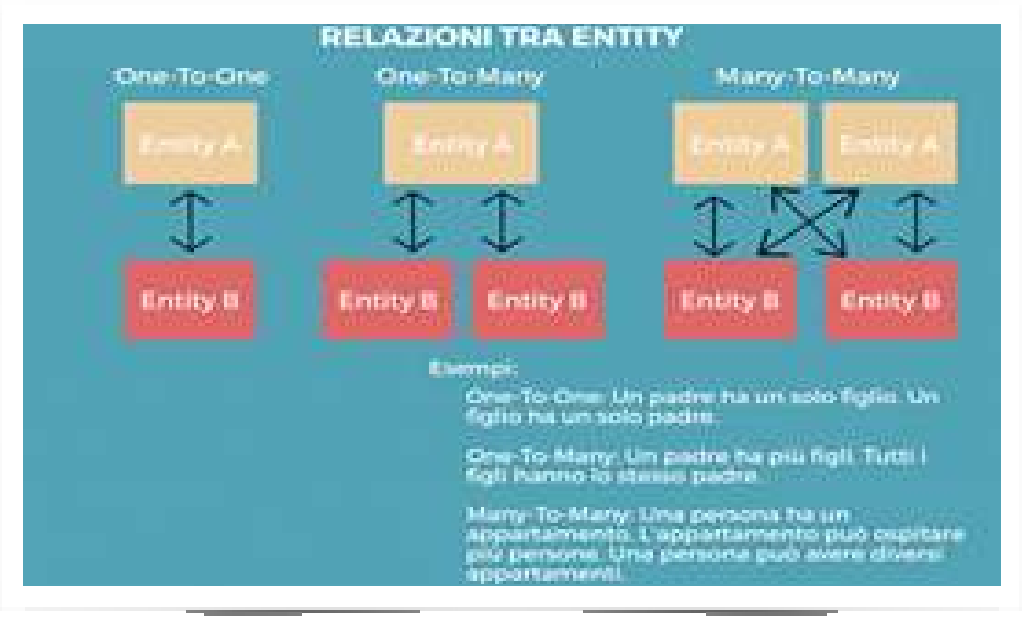
```
import Foundation
import CoreData
import CoreLocation


class SARPoint: NSManagedObject
{

    class func pointWithId(id: String, coordinate: CLLocationCoordinate2D,
        elevation: Double, velocity: Double,
        inContext context: NSManagedObjectContext, createIfNeeded: Bool = true)

        let point = NSEntityDescription.insertNewObjectForEntityForName("SARPoint",
inManagedObjectContext: context) as! SARPoint

        point.id=id
        point.latitude = coordinate.latitude
```

```
        point.longitude = coordinate.longitude
        point.elevation = elevation
        point.velocity = velocity

        return point
}
```

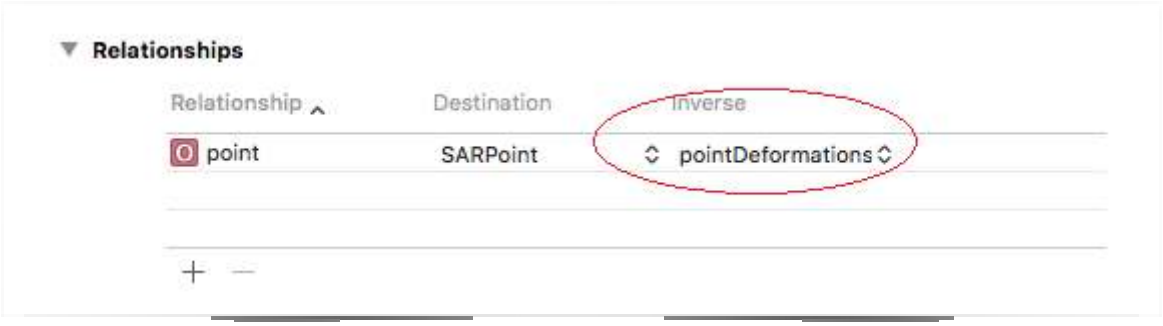
Relazioni

Per relazione si intende il metodo per associare due entità provenienti da classi diverse.
Consiste in un vero e proprio collegamento che permette di stabilire una parentela tra questi oggetti. Le relazioni in base alla loro cardinalità possono essere:

- ⊕ One to One
- ⊕ One to Many
- ⊕ Many to Many



Parlando di relazioni in Core Data dobbiamo sottolineare un aspetto importante ossia la possibilità che essa sia inversa, e quindi bidirezionale, oppure semplicemente unidirezionale. L'invertibilità permette, in modo automatico, di creare una relazione opposta per la classe coinvolta nella relazione. Questo fa si che noi possiamo richiamare il collegamento da entrambe le entità, cosa che non accadrebbe senza questa clausola.



Operazioni con Core data

Come per ogni qualsiasi gestore di dati Core Data implementa delle funzioni apposite per l'inserimento oppure per la lettura dei dati dalla memoria. Per quanto riguarda l'inserimento i dati verranno passati direttamente al Contex che si occuperà del loro salvataggio mentre per la lettura verranno utilizzate la funzione NSFetchRequest.

Inserimento

Le operazioni di inserimento di dati nella memoria tramite l'uso di Core Data avvengono assegnando l'oggetto istanziato al contex. Questo con la funzione save() memorizzerà tutto ciò che gli è stato associato sulla memoria.
L'oggetto verrà inizializzato grazie ad un costruttore. Per associarlo al context oltre che ai valori dei vari attributi a questo va passato per riferimento il context che gestisce la nostra applicazione. Implementata la funzione di costruttore andremo a definire il nostro oggetto, acui dovremo ancora assegnare le varie variabili, come un NSEntityDescription. Tramite la funzione insertNewObjectForEntityForName(), a cui passeremo il nome della classe entity dell oggetto e il context, avremo stabilito la relazione con il contex. Seguiranno poi tutte le assegnazioni relative ai vari attributi.
In questo modo al salvataggio del contex qualunque oggetto della nostra sottoclasse verrà memorizzato.

```
import Foundation
import CoreData
import CoreLocation


class SARPoint: NSManagedObject {

    class func pointWithId(id: String, coordinate: CLLocationCoordinate2D,
        elevation: Double, velocity: Double,
        inContext context: NSManagedObjectContext, createIfNeeded: Bool = true)


let point = NSEntityDescription.insertNewObjectForEntityForName("SARPoint",
inManagedObjectContext: context) as! SARPoint

        point.id=id
        point.latitude = coordinate.latitude
        point.longitude = coordinate.longitude
        point.elevation = elevation
```

```
        point.velocity = velocity

        return point
}
```

Tramite questa funzione l'oggetto verrà memorizzato.

```
func saveContext () {

        if managedObjectContext.hasChanges {

            do {

                try managedObjectContext.save()

                print("Context saved!")
            } catch {

                let nserror = error as NSError
                NSLog("Unresolved error \(nserror), \(nserror.userInfo)")
                abort()
            }
        }
    }
```

Un'altra possibile soluzione è quella di assegnare il contex successivamente dopo aver definito l'oggetto. Avremo però un'applicazione spezzata in blocchi non generalizzati e quindi difficili da comprendere e molto più dispendiosi a livello computazionale.

Estrazione/Lettura dei dati

Una volta memorizzati i dati le operazioni necessarie alla loro estrazione per modifica o per la visualizzazione del contenuto sono realizzate tramite l'uso delle FetchRequest. Questa funzione a cui è associato un predicate permettono di avere come risultato una serie di oggetti salvabili all'interno di variabili. La NSFetchRequest necessita come parametro il nome identificativo dell'entità di cui vogliamo prelevare le istanze. Oltre alla request come detto bisogna specificare il predicate che rappresenta in sisntesi i valori di ritorno. Pensando a un Database Relazionale corrisponde ai valori specificati di seguito alla funzione SELECT. La funzione che esegue le varie operazioni nel loro complesso è la executeFetchRequest. Questa permette di effettuare le operazioni indicate.

```
let request = NSFetchRequest(entityName: "SARPoint")
        request.predicate = NSPredicate(format: "id == %@", id)

        let points = (try? context.executeFetchRequest(request)) as? [SARPoint]
```

Realm

Realm è un database cross-platform realizzato come soluzione alla persistenza dei dati, progettato per le applicazioni mobile.
Esso è incredibilmente veloce e facile da usare. L'applicazione sarà realizzata con solo un paio di righe di codice, indipendentemente da operazioni di lettura o la scrittura sul database.

Vantaggi Realm

Facilità di installazione: L'installazione Realm è più facile. Con un semplice comando Cocoapods, siamo pronti a lavorare con Realm.
Velocità: Realm è una libreria incredibilmente veloce per eseguire operazioni con il database. Realm è più veloce di SQLite e CoreData e i benchmarks sono la prova migliore per questo (benchmark è la determinazione della capacità di un software di svolgere più o meno velocemente, precisamente o accuratamente, un particolare compito per cui è stato progettato).
Cross Platform: i file del database di Realm sono cross-platform ossia possono essere condivisi tra iOS e Android.
Scalabilità: la scalabilità è molto importante da considerare durante lo sviluppo di app per dispositivi mobili specialmente se gestisce un numero enorme di record. Realm è progettato per la scalabilità e il lavoro con i dati di grandi dimensioni in pochissimo tempo.
Una buona documentazione e supporto: la squadra di Realm ha fornito una documentazione leggibile, ben organizzata e ricca su Ream.

Installazione di Realm

La squadra di Realm ha fornito un plugin molto utile per Xcode che verrà utilizzato per la generazione di modelli Realm. Per installare il plugin si può far uso di Alcatraz . Alcatraz è un gestore di pacchetti open source per l'installazione automatica di plugin, modelli o colori in Xcode senza dover eseguire configurazioni manuali . Per installare Alcatraz semplicemente incollare il seguente comando nel terminale e quindi riavviare il Xcode:
curl -fsSL
https://raw.githubusercontent.com/supermarin/Alcatraz/master/Scripts/install.sh | s
Una volta riavviato si avrà una nuova voce nella window di Xcode denominata Packagement Manager. La sua apertura aprirà una finestra in cui potremo cercare il framework realm e aggiungerlo alla app.

Creazione di un database Realm

In modo analogo a Core Data Realm deve essegere gestito come un NSContext. Esso è infatti il context del database stesso. Questo fa sì che non vi sia una strutturazione, come nel caso dello Stack di Core Data, e quindi vi sia un salvataggio diretto contattando realm.

Per creare un oggetto Realm si utilizza il seguente codice che richiede una gestione delle eccezioni nel caso in cui l'operazione non andasse a buon fine.

```
import Foundation
import RealmSwift

extension Realm {

    static func getInstance() -> Realm {

        var realmInstance: Realm?

        do {
            realmInstance  = try Realm()

        }
        catch {
            print("Error in realm init \(error)")
        }

        // Se non crea realm è come se non ci fosse il contesto, quindi è
ragionevole che la app vada in crash

        return realmInstance!
```

Componenti di Realm

Gli elementi principali che compongono un database di Realm come per Core Data sono le entità e le relazioni. Esse possiedono però caratteristiche diverse poiché in questo caso non avremo una schermata/file che ci permette di gestirli con un'interfaccia. Dovremo invece realizzarle tramite codice, implementando delle classi per ognuna entità.

Entità

Come per Core Data le entità rappresentano sia le nostre classi di oggetti, sia gli elementi del nostro database.
Si può pensare quindi che non vi sia una differenza tra Core Data e Realm poiché anche nell'altro caso bisognava implementare le varie classi per la definizione degli oggetti veri e propri. In questo caso però le classi oltre che a permettere la generazione delle istanze, definiscono l'oggetto del database. Non dobbiamo andare ad organizzare nulla come invece dovevamo fare con Core Data. Realm si occupa della gestione del database e di organizzarlo in base alla nostra definizione delle entità che avviene a riga di codice nelle classi.

```
import Foundation
import RealmSwift

class SARDeformation: Object {

    // MARK: Properties

    dynamic var date: NSDate?
    dynamic var value: Double = 0.0
```

```
    // MARK: Methods

    static func createDeformation(
        date: NSDate,
        value: Double,
        inRealm realm: Realm) -> SARDeformation {

        let deformation = SARDeformation()

        deformation.date = date
        deformation.value = value


        return deformation
    }

}
```

Relazioni

Le relazioni in consistono nelle associazioni tra istanze di classi diverse. Nel nostro caso abbiamo due tipi di relazioni binarie: quelle che associano le zone ai punti e quelle che associano i punti alle deformazioni. Il tipo di relazione utilizzata è quella One to Many. Questo significa che ad una istanza sono legate N istanze di una diversa classe. Nel nostro caso è semplice intuire la relazione 1-N tra le zone e i punti. Una stessa zona può contenere più punti.
La definizione delle relazioni in Realm avviene indicando la cardinalità dell'associazione e la classe di riferimento.
La differenza in realm tra le relazioni 1-1 o 1-N è il fatto che nelle 1-N abbiamo come riferimento una lista di istanze anziché una unica.
Nel caso della nostra applicazione la definizione di una relazione avviene in questo modo:

```
let points = List<SARPoint>()
```

Operazioni Con Realm

Come ogni database Realm implementa delle funzioni apposite per l'inserimento oppure per la lettura dei dati dalla memoria. Per quanto riguarda l'inserimento, i dati verranno inviati direttamente a Realm che si occuperà del loro salvataggio mentre per la lettura verranno utilizzate delle query.

Inserimento

Gli oggetti da salvare vengono assegnati a Realm durante la loro creazione. Realm esegue operazioni in modo analogo a Core Data quando questo salva gli oggetti nel context. Una volta che un'instanza viene passata a Realm, tramite una funzione add() l'oggetto in questione viene memorizzato in memoria. Per realizzare questo si devono tenere conto di due operazioni:
La prima riguarda la definizione di Realm all'interno di ogni entità. In fase di creazione dell'oggetto tramite uso di un costruttore, bisogna specificare l'istanza di Realm utilizzata nell'applicazione:

```
static func createDeformation(
    date: NSDate,
    value: Double,
    inRealm realm: Realm) -> SARDeformation {

    let deformation = SARDeformation()
```

La seconda parte consiste invece nell'operazione di salvataggio vera e propria su memoria:

```
guard let demoPoints = demoZone["points"] as? [[String:AnyObject]] else {continue}
let realm = Realm.getInstance()
let points = List<SARPoint>()

for demoPoint in demoPoints {

    guard let pointId: String = demoPoint["id_point"] as? String else {continue}
    guard let latitude: Double = demoPoint["latitude"] as? Double else {continue}
    guard let longitude: Double = demoPoint["longitude"] as? Double else {continue}
    guard let elevation: Double = demoPoint["elevation"] as? Double else {continue}
    guard let velocity: Double = demoPoint["velocity"] as? Double else {continue}

    let sarPoint = SARPoint.pointWithId(pointId,
                                        coordinate: CLLocationCoordinate2D(latitude: latitude,longitude:
                                            longitude),
                                        elevation: elevation,
                                        velocity: velocity,
                                        pointDeformations: List<SARDeformation>(),
                                        inRealm: realm)


    points.append(sarPoint)
```

Istanza di Realm

L'istanza viene passata al costruttore

Il salvataggio avviene mediante queste istruzioni:

```
try! realm.write {

      realm.add(zone)
  }
```

Grazie alle associazioni unidirezionali tra zone, punti e deformazioni, l'inserimento dalle deformazioni negli appositi punti, e dei punti nelle specifiche zone, renderà sufficiente aggiungere a Realm il vettore di Zone e salvarlo. Automaticamente, grazie alle associazioni, tutto verrà memorizzato correttamente.

Estrazione/Lettura dei Dati da Realm

Le operazioni di lettura dal database sono relativamente semplici e avvengono tramite l'uso delle query. Queste restituiscono un'istanza che contiene una raccolta di oggetti. La raccolta è organizzata come un Array composto solo da oggetti di una singola classe. I risultati di una query non sono però copie dei dati: modificandoli si modificheranno i dati su disco direttamente. Questo è positivo poiche con una sola query posso svolgere operazioni di cancellazione,

modifica e lettura. Un ulteriore aspetto positivo è che essendo i dati reali si può attraversare il grafico delle relazioni direttamente dai ottenuti senza dover eseguire query di ricerca. Nel nostro caso avendo un punto senza ulteriori query potevamo sapere la zona di appartenenza o le deformazioni relative a quel punto.

```
let points = realm.objects(SARPoint.self) // ritornano tutti gli oggetti punto memorizzati in Realm
```

Questa funzione permette tramite query di ottenere tutte le zone contenute in Realm

```
    static func allZonesInRealm() -> Results<SARZone> {

        let realm = Realm.getInstance()

        return realm.objects(SARZone)
    }
```

Oltre che alle semplici query si possono applicare anche filtri in base a determinati valori dell'oggetto analizzato. Nel caso dell'esempio qui sotto viene applicato un filtro per avere come output le zone preferite, ossia con valore di bookmarked_=true.

```
static func allBookmarkedZones() -> Results<SARZone> {

        let realm = Realm.getInstance()

        return realm.objects(SARZone).filter("bookmarked_ = true")
    }
```

Mappa

Introduzione a MapKit

Per lo sviluppo dell'interfaccia che comprende la mappa è stato utilizzato il framework proprietario di Apple denominato MapKit.

MapKit permette, come anche Google Mapes, di visualizzare la mappa in due modalità: satellitare, dove si può vedere la mappa fisica, oppure in modalità classica. Le due modalità possono essere scelte grazie a un selettore posto in basso a destra dell'applicazione.

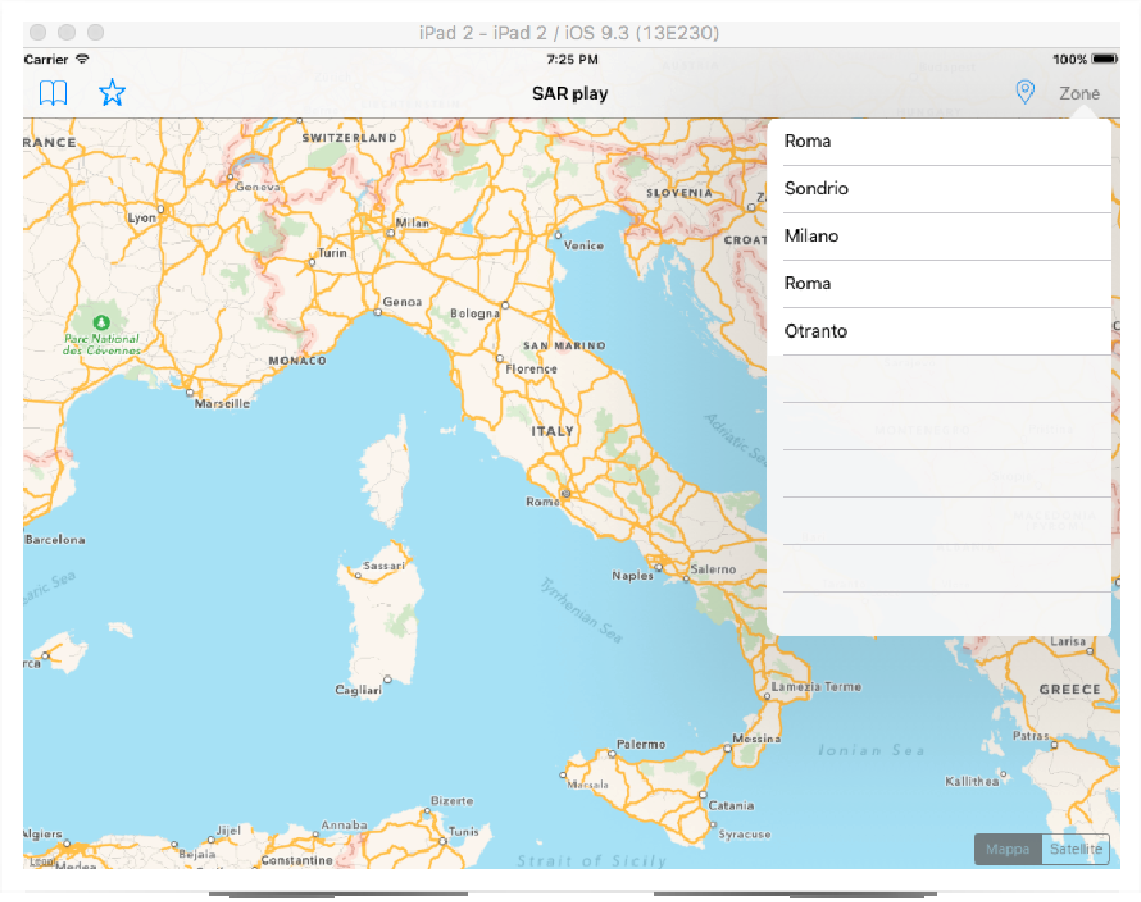
Le classi utilizzate in SARplay sono le seguenti:

- **MKAnnotation** e le sue sottoclassi:
  - MKCircle: è la classe che permette la costruzione del punto fisico;
  - MKCircleRender: è la classe che permette di visualizzare il punto sulla mappa e dargli un colore;

- **CLLocationManager**: è la classe che permette di implementare la funzione GPS;

Modalità manuale

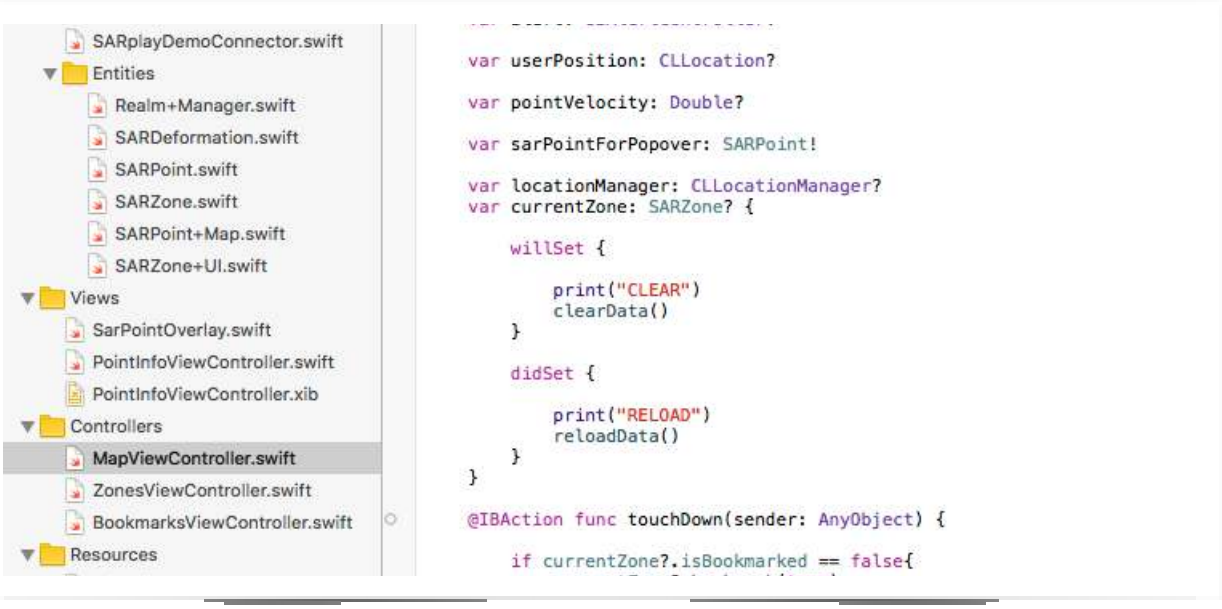
La modalità manuale è quella che permette all'utente di selezionare una zona tra quelle presenti nel database, che si trova nel menu in alto a destra "Zone";



Di seguito il meccanismo che permette, una volta selezionata la zona, di visualizzare i punti sulla mappa:
La classe MapViewController.swift contiene un attributo **currentZone** di tipo **SARZone**(entità da noi creata)
Quando questo attributo si setta, ovvero si cambia il suo valore, dapprima viene azionato il metodo **clearData** e poi il metodo **reloadData**. Questo è possibile grazie a **willSet** che permette di eseguire operazioni mentre l'attributo viene settato, e **didSet** che esegue operazioni una volta che l'attributo è stato settato.

Quando si preme una zona nel menu, scatta questo meccanismo, infatti come si nota nell'immagine, viene settato currentZone della classe MapView, e viene modificato nella zona scelta.

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {

    mapView?.currentZone = allZones[indexPath.row]

    dismissViewControllerAnimated(true, completion: nil)// chiudi la finestrella
}
```



Come già detto in precedenza, una volta che il meccanismo viene attivato, vengono chiamati i due metodi **clearData** e **reloadData** che permettono di togliere e mettere fisicamente i punti.

```
// MARK: - Methods
func clearData() {

    self.mapView.removeOverlays(self.mapView.overlays)
}
func reloadData() {

    guard let zone = self.currentZone else {return}
    let points = zone.points


    print("ZONA")
    print(zone.title)

    print("ZONA SCARICATA: \(zone)")

    // Spostamento della mappa

    let lowerLeftCorner = MKMapPointForCoordinate(zone.lowerLeftCorner)
    let upperRightCorner = MKMapPointForCoordinate(zone.upperRightCorner)

    let zoomMapRect = MKMapRectMake(lowerLeftCorner.x, upperRightCorner.y, upperRightCorner.x - lowerLeftCorner.x,
        lowerLeftCorner.y - upperRightCorner.y);

    self.mapView.setVisibleMapRect(zoomMapRect, animated: true)


    for point in points {

        print("Punto(\(point.id) [\(point.latitude), \(point.longitude)])")

        pointVelocity = point.velocity


        dispatch_async(dispatch_get_main_queue(), {

            let overlay = SarPointOverlay.sarPointOverlayWithPoint(point)


            self.mapView.addOverlay(overlay)
        })
    }
```

clearData*:** come si vede nell'immagine precedente questo metodo ha il semplice compito di togliere i punti dalla mappa, grazie al metodo ***removeOverlay.

reloadData: questo metodo è un po' più complesso, esso permette di visualizzare i punti sulla mappa in base alla zona.
Dapprima si controlla se currentZone non è nulla grazie all'uso di **guard**, e se questa condizione è vera, grazie all'istruzione **zone.points** viene creata una query sul DB che ritorna un vettore carico di tutti i punti della zona. A questo punto mentre i punti vengono caricati nel vettore interno al metodo, spostiamo la locazione della mappa in modo da poter vedere i puntini quando saranno caricati.
Questo si fa grazie a **lowerLeftCorner** e **upperRightCorner** che sono due attributi della zona; essi rappresentano:

- **lowerLetfCorner**: oggetto di tipo MKMapPointForCoordinate che contiene le coordinate del punto in basso a sinistra, come si vede nella foto;
- **upperRightCorner**: oggetto del medesimo tipo che contiene invece le coordinate del punto in alto a destra

Questi oggetti saranno poi utilizzati dalla classe MKMapRect per creare il rettangolo da visualizzare. A questo punto con il metodo **setVisibleMapRect** della mapView, a cui si passa l'oggetto precedentemente creato, la mappa si sposterà.

L'ultima cosa che manca è quella di caricare i punti sulla mappa:

```
for point in points {
    print("Punto(\(point.id) [\(point.latitude), \(point.longitude)])")
    pointVelocity = point.velocity

    dispatch_async(dispatch_get_main_queue(), {
        let overlay = SarPointOverlay.sarPointOverlayWithPoint(point)

        self.mapView.addOverlay(overlay)
    })
}
```

Si scorre il vettore **points** che contiene i punti della zona corrente; per ogni punto, in un thread diverso, viene creato un oggetto di tipo **SarPointOverlay** che è una sottoclasse di **MKCircle** da me realizzata per poter arricchire di proprietà il punto.
A questo punto si aggiunge alla mappa l'overlay. Per fare ciò il metodo **addOverlay** ha bisogno di "contattare" il CircleRender che disegna il punto sulla mappa.
Ogni volta il metodo qui riportato a sinistra prenderà un MKCircle e ritornerà un Overlay colorato in base alle proprietà del punto stesso.

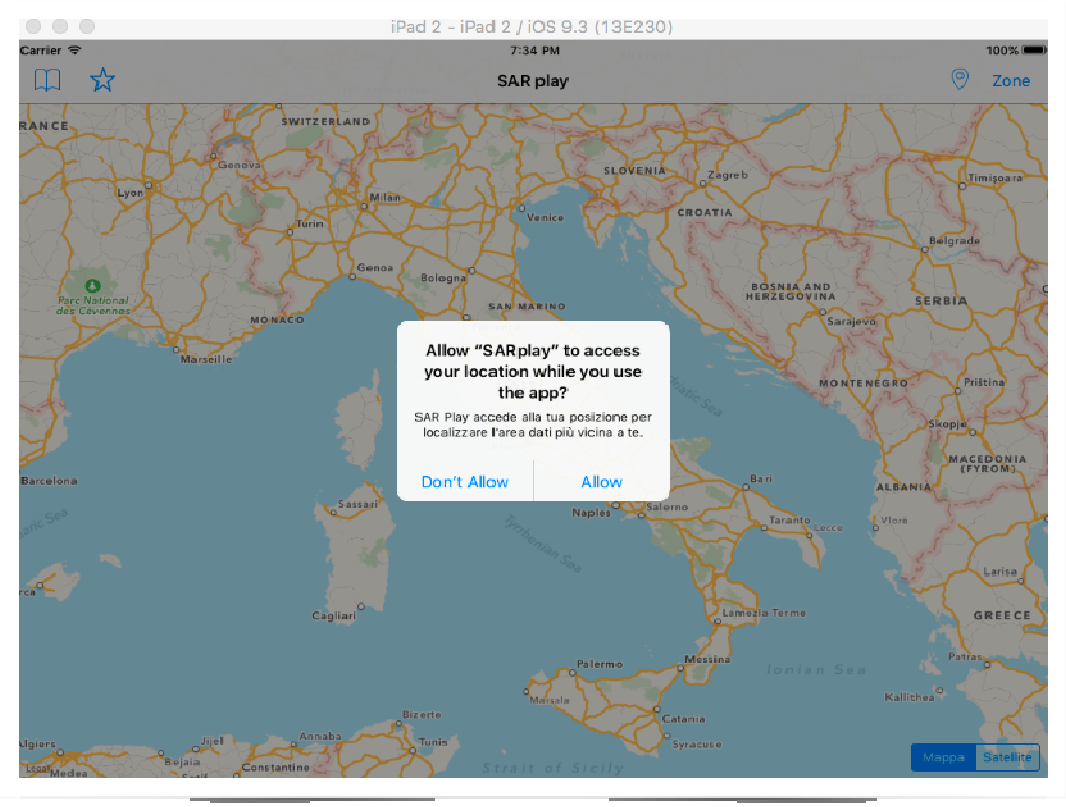
Il colore viene dato chiamando il metodo
colorForVelocity che riceve un valore decimale e ritorna il colore corrispondente secondo le specifiche del CNR.
Alla fine del ciclo tutti i punti saranno visualizzati sulla mappa.

```
func mapView(mapView: MKMapView, rendererForOverlay overlay: MKOverlay) -> MKOverlayRenderer {
    if let overlay = overlay as? SarPointOverlay, velocity = overlay.point?.velocity {
        let circleRenderer = MKCircleRenderer(circle: overlay)
        let color = colorForVelocity(velocity)
        circleRenderer.strokeColor = color
        circleRenderer.alpha = 0.5

        return circleRenderer
    }
    else {
        return MKOverlayRenderer(overlay: overlay)
    }
}
```

```
func colorForVelocity(value: Double) -> UIColor {
    // Colori secondo le specifiche CNR
    let viola = (1.07143...Double.infinity)
    let blu = (0.64286...1.07143)
    let azzurro = (0.21429...0.64286)
    let verde = ((-0.21429)...0.21429)
    let giallo = ((-0.64286)...(-0.21429))
    let arancio = ((-1.07143)...(-0.64286))
    let rosso = ((-Double.infinity)...(-1.07143))

    if viola.contains(value) {
        return UIColor.purpleColor()
    }
    if blu.contains(value) {
        return UIColor.blueColor()
    }
    if azzurro.contains(value) {
        return UIColor.cyanColor()
    }
```

Modalità automatica (GPS)

In generale il funzionamento che permette di visualizzare i punti è lo stesso visto in precedenza, l'unica cosa che cambia in questa modalità è come settare **currentZone**, infatti ora la zona sarà modificata non più con il click della zona nel menu a destra, ma premendo il bottone della modalità automatica a fianco del precedente.
La prima volta che l'app viene avviata, viene visualizzata la seguente schermata:



Premendo su **Allow**, sarà concesso all'applicazione di usare il GPS del nostro dispositivo.

Premendo quindi sul bottone  , verrà chiamato il metodo **automaticType** della classe MapViewController.swift.

Il metodo automatico, dapprima setta l'immagine del bottone premuto con la modalità manuale, così da far capire che ora si è in automatica, poi dopo di che

controlla se il GPS funziona correttamente e la variabile **userPosition** contiene delle coordinate. Se ciò è vero
sposta la mappa nella zona dove siamo usando le coordinate della **userPosition,** grazie al **setRegion** della MapView.

A questo punto manca solo di settare la zona;
Dato che abbiamo però solo le coordinate di dove siamo, ci necessita capire in che zona siamo per poter caricare i punti: questo avviene attraverso lo **zoneContainingLocation** qui sopra riportato, che avendo in input un CLLocation, che corrispondono a due coordinate, restituisce la SARZone corrispondente. Il metodo non è complesso, infatti, controlla se il "rettangolo" delle zone contiene la nostra localizzazione. Se la localizzazione è contenuta, allora currentZone sarà settata e secondo il metodo descritto precedentemente i punti verranno caricati, altrimenti verrà visualizzato un popover che farà capire che la zona non è presente nell'archivio.

```
func automaticType()
{
    // TODO: capire come gestire il bottone quando e' in modalita GPS

    print("AUTOMATIC TYPE")

    TypeLocation.image = UIImage(named:"manuale")

    if let userPosition = self.userPosition {

        let span = MKCoordinateSpanMake(0.5, 0.5)
        let region = MKCoordinateRegion(center: userPosition.coordinate, span: span)
        mapView.setRegion(region, animated:true)


        currentZone = SARZone.zoneContainingLocation(userPosition)
        print("settint zone \(currentZone)")

        if currentZone == nil{

            self.presentViewController(alert, animated: true, completion: nil)
        }

    }

}
```

```
class func zoneContainingLocation(location: CLLocation) -> SARZone? {

    let point = MKMapPointMake(location.coordinate.latitude, location.coordinate.longitude)

    let zones = SARZone.allZonesInRealm()

    for zone in zones {

        if MKMapRectContainsPoint(zone.mapRect, point) {

            return zone
        }
    }

    return nil
}
```

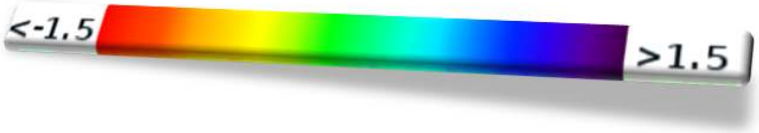
Interpretazione della feature su SARplay

Il pallino sulla mappa

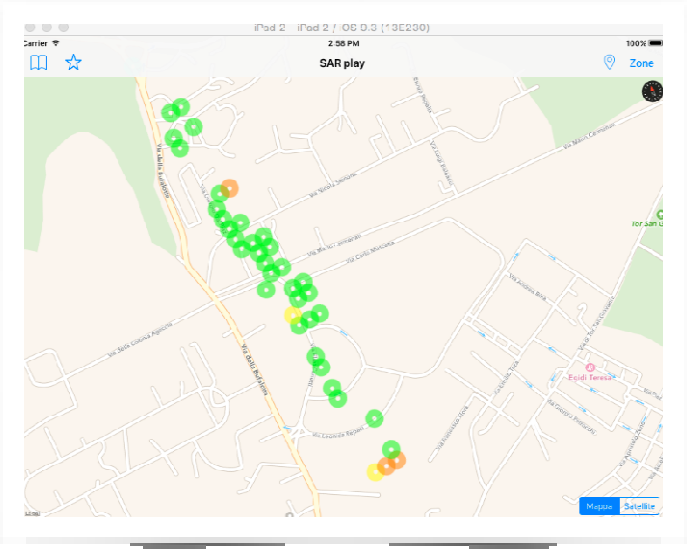
Le features, così sono chiamati i punti della mappa, permettono all'utente di capire di quanto il terreno si è alzato o abbassato rispetto alla misurazione avvenuta una settimana prima. Qui sotto sono posti alcuni screen che mostrano l'app funzionante.

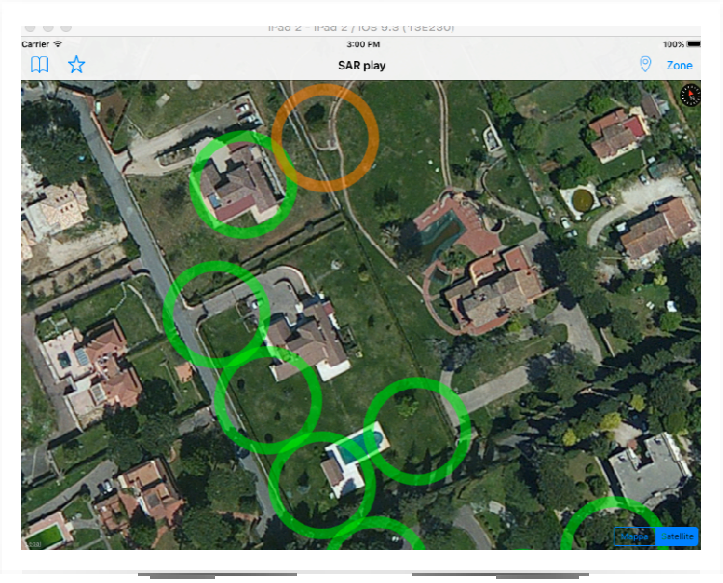
Lo spostamento del terreno varia di solito da -1.5mm e 1.5 mm, in quanto in una settimana è difficile che il terreno si sposti di più.
Qui sotto è presentata la scala dei colori dei punti.

ROSSO: -1,5mm / -1,071mm
ARANCIONE: -1,071mm / -0,64mm
GIALLO: -0,64mm / -0,21mm
VERDE: -0,21mm / 0,21mm
AZZURRO: 0,21mm / 0,64mm
BLU: 0,64mm / 1,071mm
VIOLA: 1,071mm / 1,5mm

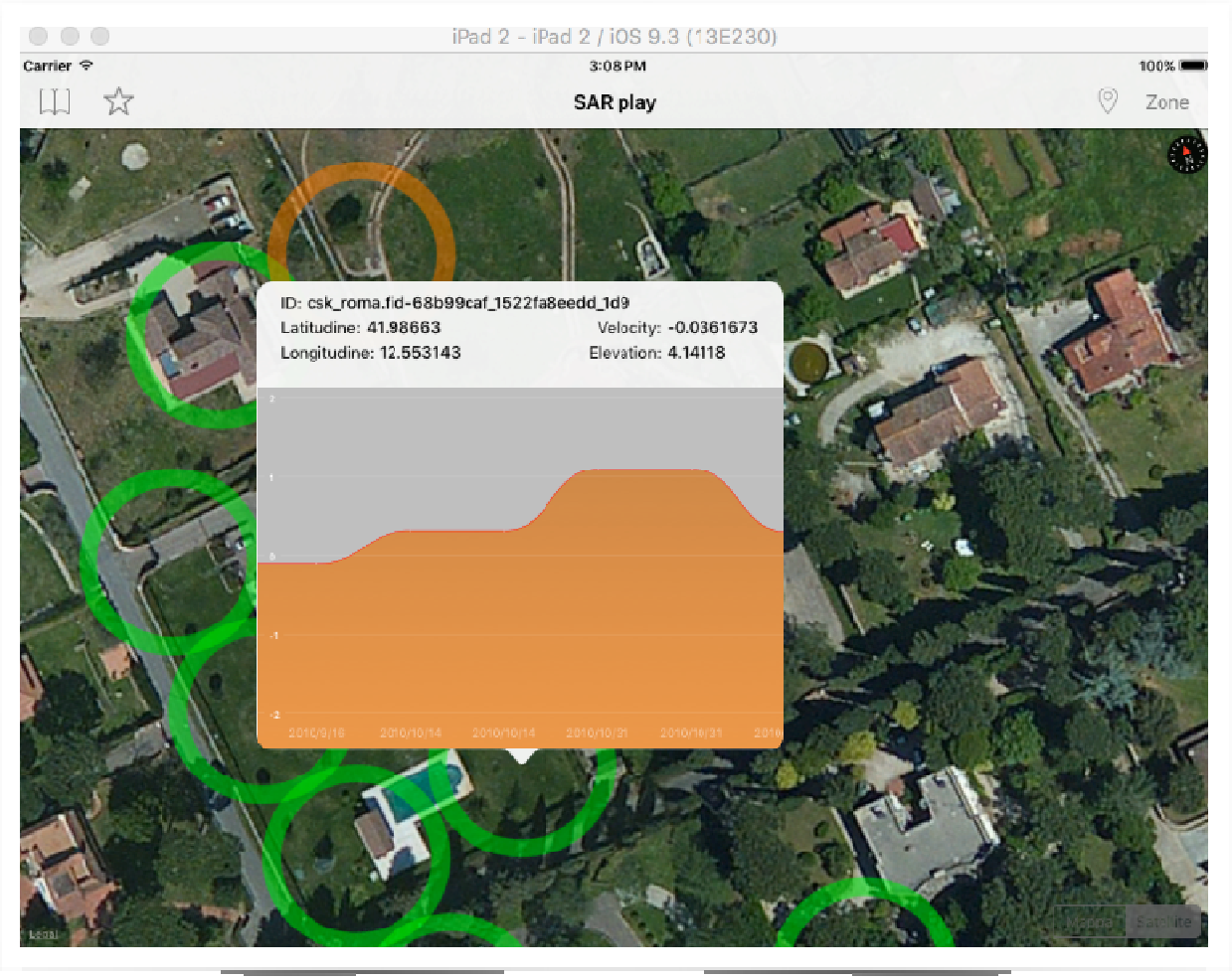


Informazioni dal grafico della feature



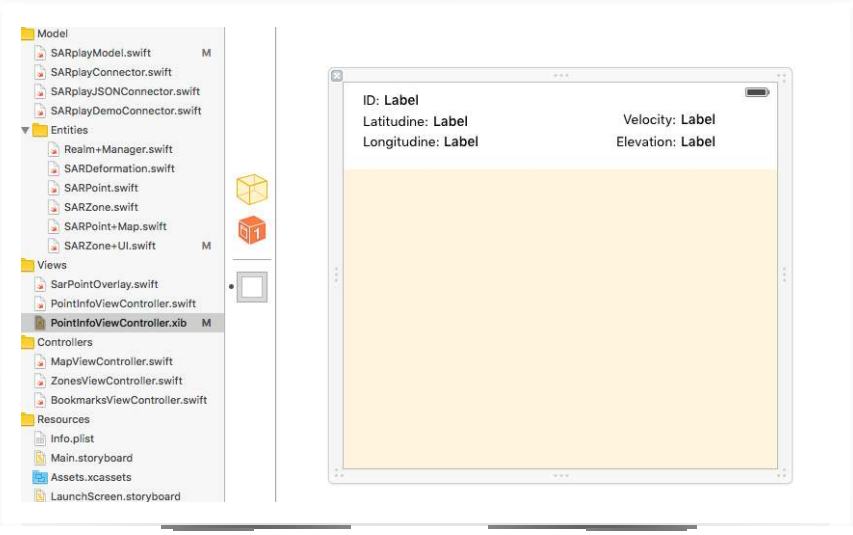


Una volta che le features si sono presentate sulla mappa, premendo sopra di esse, viene visualizzato un popover, che conterrà le principali caratteristiche del punto:



Come si vede qui sopra, il popover contiene l'**ID** del punto, valore univoco nella tabella del DB, la **latitudine** e la **longitudine**, l'**elevation** che rappresenta l'altitudine, e la **velocity** che è il parametro espresso in millimetri che ci permette di capire di quanto si è alzato il terreno, e quindi di dare un colore alla features.

Vediamo ora com'è possibile costruire un popover e dargli delle caratteristiche:
Innanzitutto è necessario che si possa interagire con l'overlay con il tap di un dito; è stato utilizzato dunque un**TapGestureRecognizer**che abbinato ad una**MKAnnotation** permette di catturare gli eventi sullo schermo. Ora vediamo come il popover viene creato:




```
print(sarPoint)

guard let sarPoint = self.sarPoint else {return}

idLabel.text = sarPoint.id
latitudeLabel.text = "\(sarPoint.latitude)"
longitudeLabel.text = "\(sarPoint.longitude)"
velocityLabel.text = "\(sarPoint.velocity)"
elevationLabel.text = "\(sarPoint.elevation)"

self.preferredContentSize = CGSizeMake(450, 400)
```

Dapprima è necessario creare un file**xib**, esso rappresenta la view che il popover presenterà al suo interno, e sarà necessario collegarlo a un'altra classe **Swift** per poterne modificare le proprietà.

Quando un overlay viene premuto, viene istanziato un oggetto del tipo PointInfoViewController a cui passeremo un punto, cosi da poter usare le sue proprietà per la costruzione dell'interfaccia.

Dapprima si controlla che il punto non sia nil, con l'istruzione **guard**, se è vero, si settano le varie label dello xib sopra presentato con i valori del punto corrente.
Per ultima cosa fissiamo la grandezza dello xib:
450x400 è la misura ideale considerando i dispositivi su cui SARplay viene eseguito.

Il grafico

Il grafico contenuto nel popover rappresenta i dati contenuti nella lista pointDeformations che è un attributo del punto SARPoint. Essa è formata da:

- value: che contiene alcuni valori di innalzamento precedenti a quello corrente;
- date: che contiene la data corrispondente del valore sopra;

Ora vediamo come si costruisce il grafico reale.
Innanzitutto è necessario importare una libreria, la ScrollableGraphView che ci permette di costruire grafici complessi e migliori esteticamente.
Dapprima s'istanza un oggetto del tipo ScrollableGraphView e gli si passa la finestra in cui dovrà adattarsi. Dopodiché è necessario modificare i colori delle parti del grafico a nostro piacimento. Come ultima cosa è necessario passare i dati al grafico, si suddivide quindi la lista in due array e si passano al grafico con il metodo setData.

Durante la prototipazione del grafico, si è notato che i dati nella lista non erano in ordine di tempo; è stato quindi necessario riordinarli. Questo è stato facilmente possibile grazie al metodo sort della lista, che permette di ordinarla con due semplici istruzioni.

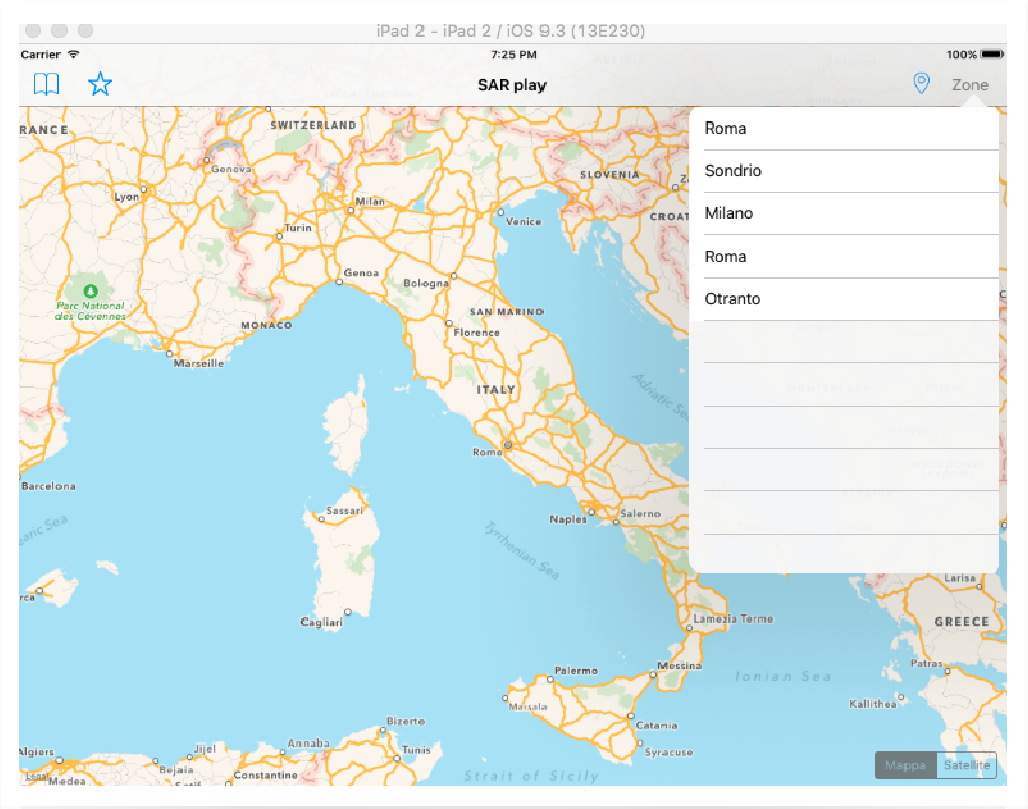
```
// Creazione Grafico
graphView = ScrollableGraphView(frame: graphV.frame)

// Posizionamento del Grafico
graphView.frame = CGRectMake(0, 0, graphV.frame.size.width, graphV.frame.size.height)

graphView.backgroundFillColor = UIColor(red:0.75, green:0.75, blue:0.75, alpha:1.0)
  graphView.layer.cornerRadius = 7

graphView.rangeMin = -1.5
graphView.rangeMax = 1.5

graphView.lineWidth = 0.5
graphView.lineColor = UIColor.redColor()
graphView.lineStyle = ScrollableGraphViewLineStyle.Smooth
graphView.shouldFill = true
graphView.fillType = ScrollableGraphViewFillType.Gradient
graphView.fillColor = UIColor.blackColor().colorWithAlphaComponent(0.8)
graphView.fillGradientType = ScrollableGraphViewGradientType.Linear
graphView.fillGradientStartColor = UIColor.brownColor().colorWithAlphaComponent(0.8)
graphView.fillGradientEndColor = UIColor.orangeColor().colorWithAlphaComponent(0.6)
graphView.dataPointSpacing = 80
graphView.dataPointSize = 0.5
graphView.dataPointFillColor = UIColor.whiteColor()

graphView.referenceLineLabelFont = UIFont.boldSystemFontOfSize(8)
graphView.referenceLineColor = UIColor.whiteColor().colorWithAlphaComponent(0.2)
graphView.referenceLineLabelColor = UIColor.whiteColor()
graphView.dataPointLabelColor = UIColor.whiteColor().colorWithAlphaComponent(0.5)

let graphData = array(forPoint: sarPoint)

print(graphData)

graphView.setData(graphData.data, withLabels: graphData.labels)

self.graphV.addSubview(graphView)
```

Menù zone e preferiti

Tra le varie funzionalità di SARplay, vi è la possibilità di scegliere la zona su cui operare, oltre alla possibilità di poter contrassegnare le zone visitate maggiormente come “preferiti” e di vederle comparire in un menu a parte. Scegliendo una zona, si viene rimandati sulla regione di mappa interessata, senza dover inserire coordinate o riferimenti geografici ulteriori.
I menu sono formati dai seguenti elementi:

- Bottone
- Tableview
- Segue

Il bottone ha il compito di fare da collegamento tra la mappa e il nostro menu: infatti, cliccandovi sopra, si aprirà la nostra tableView, contenente l’elenco delle zone; ciò che si interpone tra bottone e tableView, si chiama segue; nelle pagine seguenti verrà illustrato tutto il meccanismo che si cela dietro a questi tre elementi



Il menù delle zone

La classe di riferimento per il menu delle Zone è la ZonesViewController.swift, di tipo UITableViewController, ed ha i seguenti attributi:

- mapView, di tipo MapViewController, che non è altro che la mappa principale dell'app, sulla quale ci si sposta con la scelta della zona;
- allZones, che è un array, contenente tutte le zone tutte le zone disponibili, ottenute interrogando il database locale;
- tableview, che è il menu di selezione delle zone. La scritta @IBOutlet sta a significare che la variabile in questione è un collegamento diretto ad un elemento presente nel Main.storyboard, ovvero la sezione dell'ambiente di sviluppo dove sono collocati gli elementi grafici ed i vari collegamenti ad essi.

Struttura della classe ZonesViewController

```
//
//  ZonesViewController.swift
//  SARplay
//
//  Created by Michele Longhi on 24/02/16.
//  Copyright © 2016 Visuality srl. All rights reserved.
//

import UIKit

class ZonesViewController: UITableViewController {

    var mapView: MapViewController?
    let allZones:[SARZone] = Array(SARZone.allZonesInRealm())

    @IBOutlet var tableview: UITableView!
```

La classe zonesViewController è formata dai seguenti metodi:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 1
}
```

```
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows

    return allZones.count
}
```

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell:UITableViewCell = self.tableView.dequeueReusableCellWithIdentifier(cellReuseIdentifier) as UITableViewCell!

    cell.textLabel?.text = allZones[indexPath.row].title as String

    return cell

}
```

Il metodo numberOfSectionsInTableView è uno dei molti metodi nativi, ovvero che va ridefinito in base agli usi che lo sviluppatore ne farà; questo metodo in particolare ritorna il numero di sezioni che il menu avrà. Di default, tutte le tableView ne hanno solo una, ed è così anche nel nostro caso.

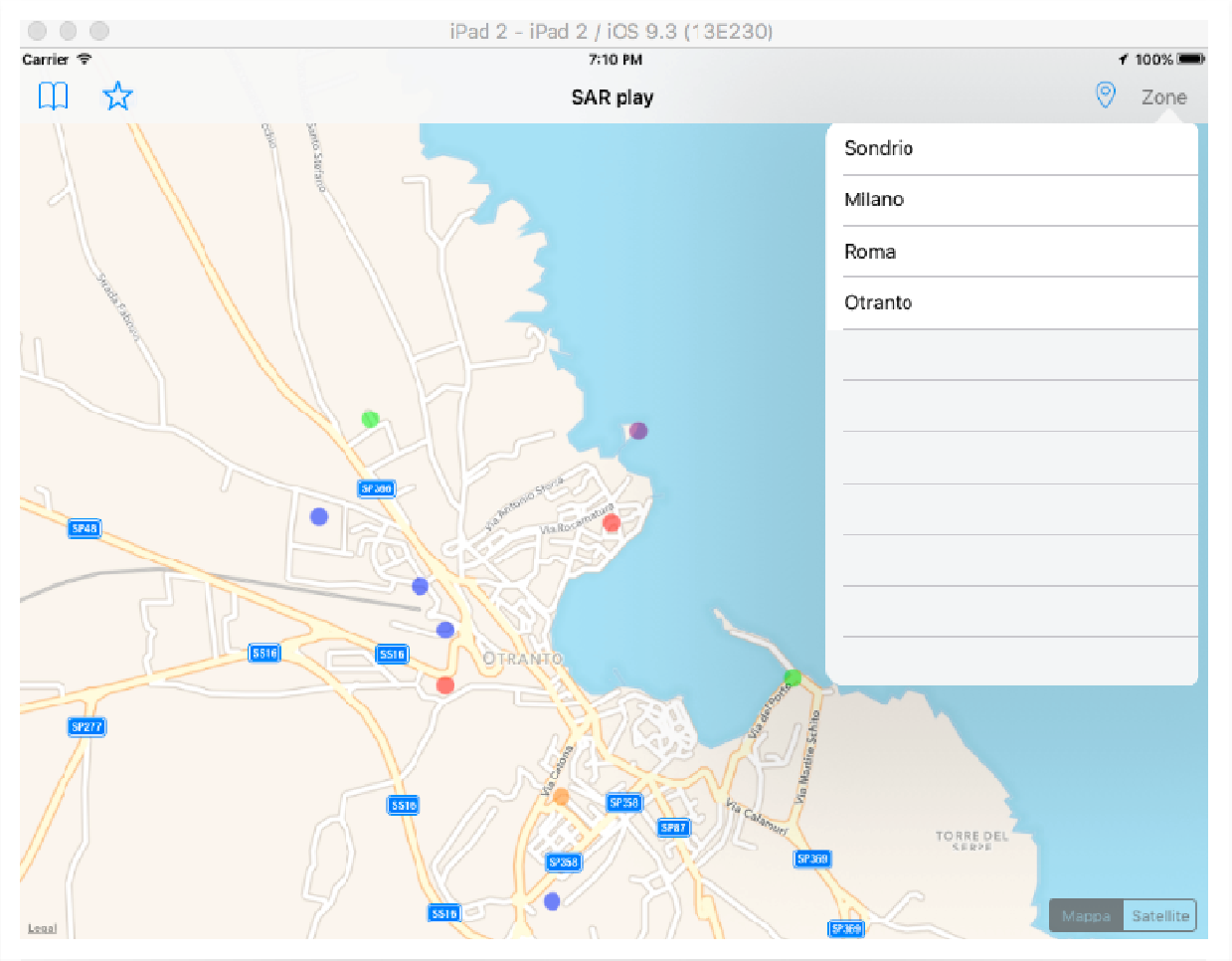
Il metodo tableView viene ridefinito molte volte: in questo caso, gli viene passato l'elemento "tableView", simboleggiante la tabella contenente le varie zone, ed ha il compito di definire il numero di righe che il menu avrà; per far ciò basta contare il numero di elementi che ha il vettore allZones.

Anche in questo caso, al metodo tableView viene passata la stessa tableView, ma la funzione del metodo è totalmente diversa: il metodo viene richiamato tante volte quante sono le celle (vedi metodo precedente), e per ciascuna di esse viene modificata la propria label col nome di una zona.

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    mapView?.currentZone = allZones[indexPath.row]
    dismissViewControllerAnimated(true, completion: nil)// chiudi la finestrella
}
```

L'ultimo metodo di questa classe viene invocato ogni volta che si verifica una selezione di un elemento del nostro menu: nel nostro caso si vuole che ci si sposti su una determinata zona, pertanto si va a modificare l'attributo "currentZone" (zona corrente in cui ci si trova) dell'oggetto mapView con l'elemento da noi scelto, cercandolo all'interno del vettore allZones. Una volta selezionato, la schermata del menu viene chiusa, in modo che l'utente possa usare la mappa per intero e senza menu aperti.

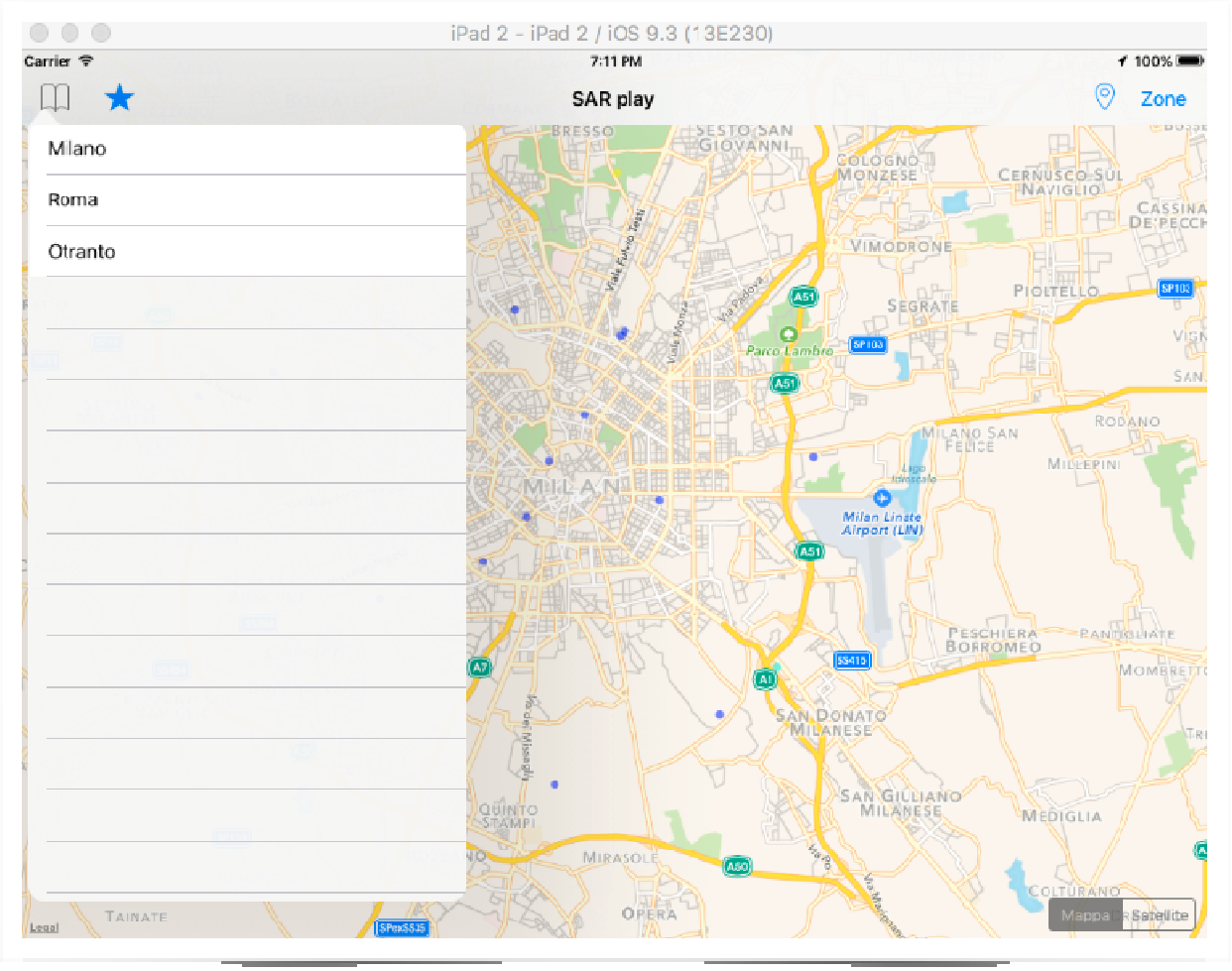
Il risultato ottenuto è il seguente:



Il menù dei preferiti

Per una maggiore comodità e semplicità di utilizzo, è stata pensata una zona preferiti, in maniera che l'utente possa salvare le zone che visita maggiormente senza il bisogno di doverle cercare ogni volta tra tutte le zone disponibili. Il funzionamento è molto simile al menu dei preferiti, infatti il menu dei preferiti è formato dagli stessi elementi del menu precedente, però in più vi è il pulsante per aggiungere/rimuovere la zona dei preferiti, rappresentato da una stella piena, quando la zona è tra i preferiti, e una stella vuota quando la zona non è tra i preferiti.

Il risultato è il seguente:



Inserire una zona tra i preferiti

Il frammento di codice riportato sotto viene richiamato quando viene premuto il pulsante di inserimento/rimozione della zona corrente dai preferiti.

```
@IBAction func touchDown(sender: AnyObject) {

    if currentZone?.isBookmarked == false{
        currentZone?.bookmark(true)
        setBookmark.image = UIImage(named: "bookmark.png")
    }
    else{
        currentZone?.bookmark(false)
        setBookmark.image = UIImage(named: "not-bookmark.png")

    }
}
```

Ogni oggetto di tipo SARZone ha come attributo un valore denominato "bookmarked_" di tipo boolean e il metodo .isBookmarked(), che ritorna "TRUE" se la zona è tra i preferiti, e ritorna false in caso contrario.
Qualora la zona non sia tra i preferiti, l'immagine del bottone "aggiungi/rimuovi" è una stella vuota: in caso di pressione, bisogna innanzitutto aggiungere la zona ai preferiti, utilizzando il metodo .bookmark(true), e in seguito cambiare l'immagine, scegliendo la stella piena.
Nel caso che la zona sia già tra i preferiti, bisogna fare il procedimento opposto, ovvero rimuovere la zona corrente dai preferiti, usando il metodo .bookmark(false), ed impostare l'immagine del bottone con la
Stella vuota

Struttura della classe BookmarksViewController

Anche in questo caso, la classe BookmarksViewController è di tipo UITableViewController, dato che nell'app è presente un menù contenente tutte le zone preferite. Gli attributi di questa classe, sono molto simili a quelli della ZonesViewController, infatti sono:

- zones, ovvero un vettore contenente tutte le zone che hanno il valore "bookmarked_" impostato come "true";
- mapview, di tipo MapViewController, ovvero la variabile che si riferisce alla mappa della schermata principale

La classe bookmarksViewController è inoltre formata dai seguenti metodi:

Come nel caso precedente, è il metodo che definisce il numero di sezioni di una tableView, e, come nel caso precedente, ritorna 1.

```swift
//
//  BookmarksViewController.swift
//  SARplay
//
//  Created by Michele Longhi on 24/02/16.
//  Copyright © 2016 Visuality srl. All rights reserved.
//

import UIKit

class BookmarksViewController: UITableViewController {

    var zones:[SARZone] = Array(SARZone.allBookmarkedZones())
    var mapView: MapViewController?
```

```swift
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
    return 1
}
```

```swift
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return zones.count
}
```

Anche in questo caso, il metodo riportato sopra ha la funzione di definire il numero di righe che il menu dovrà contenere; per far ciò, è sufficiente contare tutti gli elementi contenuti del vettore “zones”, poiché ciascun elemento di esso occuperà una cella del menu.

```swift
override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell:UITableViewCell = self.tableView.dequeueReusableCellWithIdentifier(cellReuseIdentifier) as UITableViewCell!
    // Configure the cell...
    cell.textLabel?.text = zones[indexPath.row].title as String

    return cell
}
```

Questo metodo ridefinito ha il compito di dare un nome a ciascuna cella del menu, andando a prelevarlo dal nome di ciascuna zona contenuto nel vettore “zones”; come nel caso precedente, questo metodo viene invocato ad ogni apertura del menu a tendina tante volte quante sono le zone contenute nell’array “zones”.

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {

    mapView?.currentZone = zones[indexPath.row]

    dismissViewControllerAnimated(true, completion: nil)// chiudi la finestrella
}
```

La funzione in questione viene richiamata ogni qualvolta viene registrato un evento di pressione su una qualunque cella del menu: nel nostro caso, ci si vuole spostare sulla zona da noi scelta, pertanto si imposta il valore "currentZone" della mappa, contenuta nella schermata principale, con il valore dell'array nella posizione da noi scelta.

Tuttavia, sia nel caso precedente, sia in questo, una volta premuta la cella contenente la zona desiderata non succederebbe nulla, se non ci fossero i "segue": più avanti verrà spiegata la loro funzione.

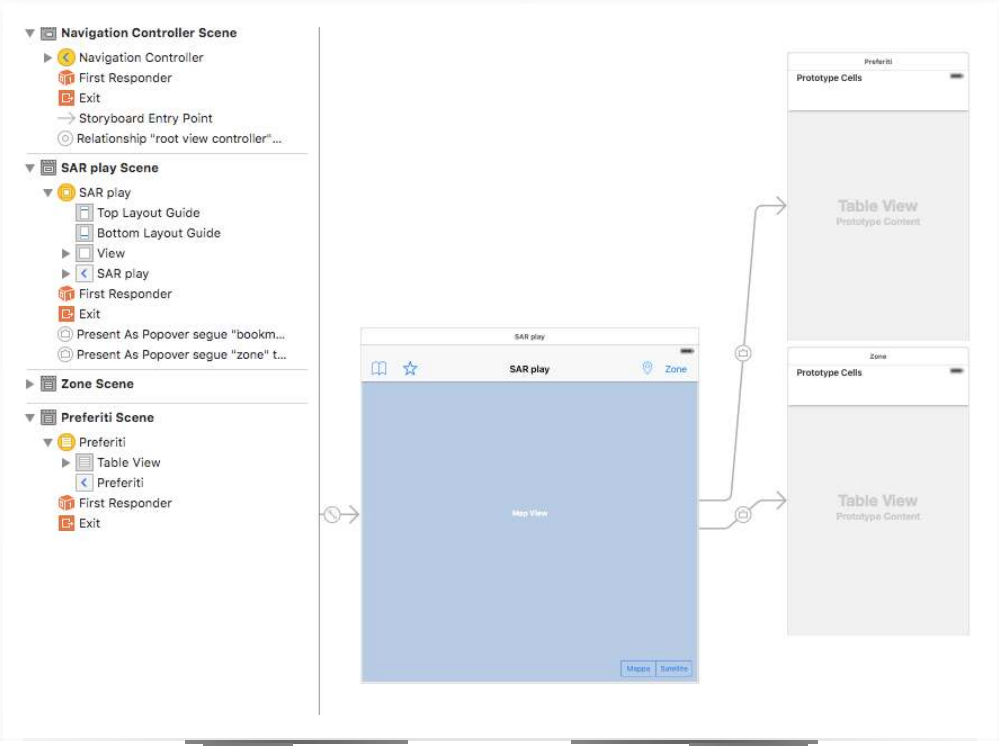
Segue

Un Segue è l'operazione che ti permette di spostarsi da un ViewController A ad uno B che si trova in una posizione successiva rispetto alla partenza. Oltre a spostarsi semplicemente, i segue permettono anche di trasferire dati da una view ad un'altra.
Nel nostro caso i segue sono due, e collegano i bottoni "Zone" e "Preferiti" alle rispettive tableView, come in figura:

I due segue hanno un nome identificativo, in maniera che siano univoci e che non vi possano essere ambiguità qualora essi vengano richiamati.

Il frammento di codice sopra riportato viene richiamato ogni qualvolta si cerca di modificare l'attributo "currentZone", poiché esso si trova in una view diversa da quella corrente.

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {

    if let newController = segue.destinationViewController as? ZonesViewController {

        newController.mapView = self
        TypeLocation.image = UIImage(named:"automatico")
    }


    if let newController = segue.destinationViewController as? BookmarksViewController {

        newController.mapView = self
        TypeLocation.image = UIImage(named:"automatico")
    }
```



Librerie di terze parti

Per la realizzazione di SARplay è stato necessario appoggiarsi a software scritto da altre persone in quanto realizzare una soluzione di persistenza (database) e la creazione di grafici sarebbe stato inutile data la moltitudine di librerie già presenti e la loro elevata qualità.

CocoaPods - Dependency manager

CocoaPods è un dependency manager per progetti in Swift e Objective-C. Ha più di 18.000 librerie ed è utile in quanto è facilmente scalabile e sopratutto facile da mantenere. Se infatti una libreria viene aggiornata sarà sufficiente eseguire un comando sulla CLI per aggiornare la libreria in locale.
CocoaPods è realizzato in Ruby ed è facilmente installabile sul proprio Mac aprendo il terminale e dando il seguente comando:

```
>> sudo gem install cocoapods
```

Per abilitare CocoaPods nel nostro progetto, sempre tramite CLI sarà sufficiente recarsi nella cartella del progetto e dare il comando:

```
>> pod init
```

Pod creerà dunque un file chiamato “Podfile”. Qui è riportato quello del nostro progetto.

```
platform :ios, '9.0'

use_frameworks!

target 'SARplay' do

pod 'RealmSwift'
pod 'ScrollableGraphView'

end
```

Una volta creato e salvato il Podfile, per completare l’installazione delle librerie si esegue il comando:

```
>> pod install
```

Una volta terminata la procedura, basterà aprire il progetto e tutte le librerie saranno disponibili e funzionanti.

RealmSwift

Realm è una soluzione di persistenza nata come progetto interno a Zynga (una softwarehouse famosa per la moltitudine di giochi pubblicati su Facebook) e in seguito resa open source e disponibile a tutti gratuitamente.
Inizialmente SARplay utilizzava CoreData che è un database ORM basato su SQLite sviluppato da Apple. Purtroppo una volta testato il codice ci siamo accorti che CoreData è troppo lento per gestire una moltitudine di dati e relazioni come quelli presenti nella app. Siamo dunque giunti alla decisione di utilizzare Realm come database, che oltre a essere più veloce di circa l’80% è molto più semplice e richiede molte meno righe di codice per ottenere lo stesso risultato.

ScrollableGraphView

ScrollableGraphView è una libreria per grafici adattivi nata, come Realm, per supportare un progetto privato e in seguito resa open source tramite GitHub.

Il suo uso è davvero semplice e richiede pochissime righe di codice.

```
let graphView = ScrollableGraphView(frame: someFrame)
let data: [Double] = [4, 8, 15, 16, 23, 42]
let labels = ["one", "two", "three", "four", "five", "six"]
graphView.setData(data, withLabels: labels)
```

```
someViewController.view.addSubview(graphView)
```

Si crea la vista contenente il grafico e la si aggiunge al ViewController