I.T.I.S. “E.MATTEI”
Indirizzo di Informatica

CLASSE 5E

SARPLAY - UN’APP PER IPAD CHE MOSTRA LE DEFORMAZIONI DEL TERRENO SULLA BASE DI DATI SATELLITARI

ELABORATO DI: Elia Figoni

Anno scolastico 2015/2016

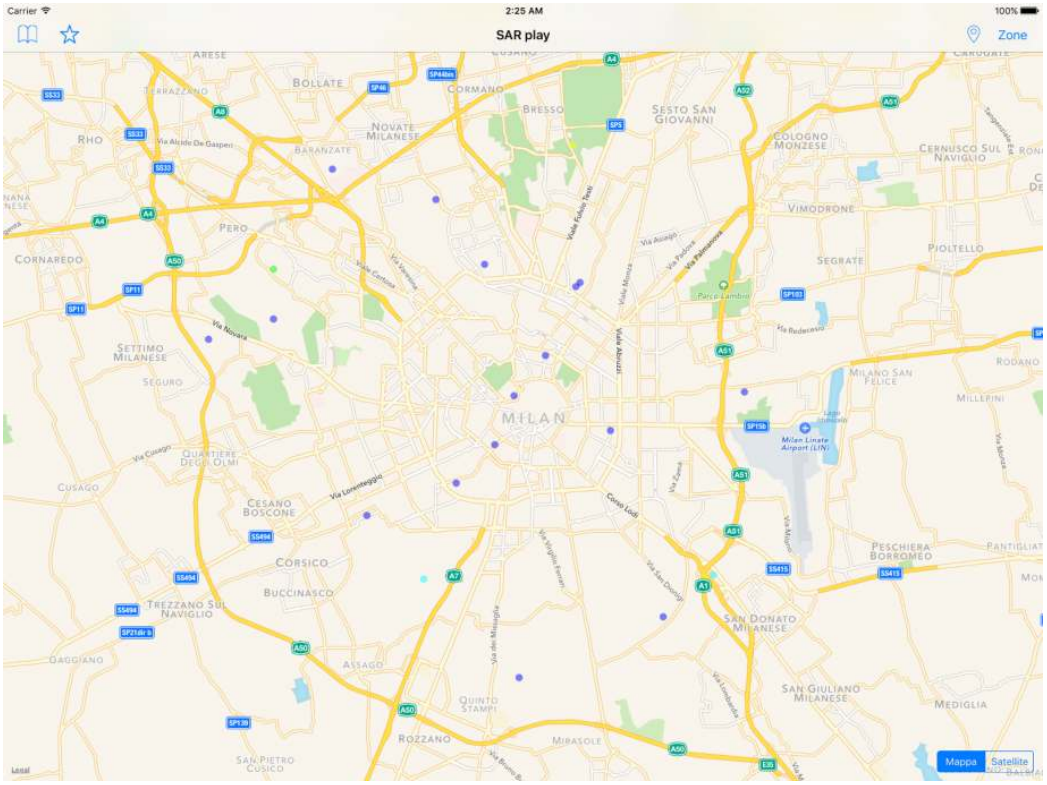
Indice dei contenuti

CAPITOLO 1
INTRODUZIONE

L'applicativo "SARplay" nasce grazie a due eventi che si sono svolti presso l'I.T.I.S. "E.Mattei" di Sondrio, durante l'anno scolastico 2015-2016:

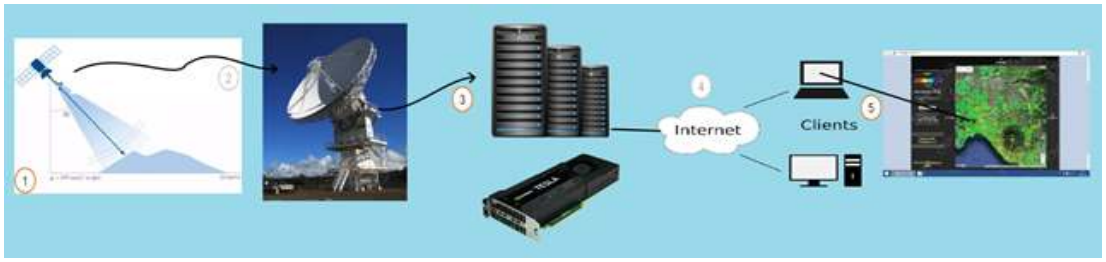
- il primo riguarda la collaborazione della nostra scuola con la software house "Visuality Software srl", la quale, in cerca di una sede a Morbegno e di nuovo personale, si è offerta di istruire e trasmettere la sua esperienza nel mondo Apple verso gli studenti interessati;
- il secondo evento vede coinvolto l'ex professore di Informatica Sabatino Bonanno, 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 edintuitiva, da realizzare insieme alla Visuality Software e con la collaborazione dell'IREA.


Carrier
2:25 AM
100%
SAR play
Zone
MILAN
Mappa
Satellite

CAPITOLO 2
RACCOLTA, ELABORAZIONE E VISUALIZZAZIONE DEI DATI LATO WEB

Per poter visualizzare il movimento del territorio nelle zone specificate e messe a disposizione dal CNR i dati per essere raccolti, elaborati e visualizzati richiedono una serie di passaggi schematizzabili con lo schema sotto rappresentato.



Dallo schema si evidenziano le 5 fasi che il processo richiede:

1. La fase di 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, lunghezza d'onda ~24 cm), la banda C (frequenza 5-6 GHz, lunghezza d'onda ~6 cm) e la banda X (frequenza 8-12 GHz, lunghezza d'onda ~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 vari dati questi verranno inviati a delle antenne satellitari. Queste faranno da intermediari con i server del CNR ai quali saranno inviate tutte le informazioni in formato stringa e ancora da elaborare.
3. Il compito dei server è quello di prendere le immagini satellitari ed elaborarle. L'elaborazione consiste nel rappresentare un immagine detta interferometria. L'interferometria è la misurazione delle variazioni della fase del segnale SAR tra due acquisizioni distinte. Una seconda elaborazione consiste invece nella trasformazione dell'interferometria in formato stringa, in modo da rendere semplice la gestione dell'immagine. Questo intero processo, totalmente automatizzato tramite l'uso di programmi in C, è realizzato tramite l'utilizzo di una architettura hardware 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.
4. I server oltre che ad occuparsi dell'elaborazione dei dati svolgono anche ruolo di WEB Server e si occupano di gestire una porzione del sito del CNR con il compito di mostrare i dati raccolti.
5. Tramite internet e con l'utilizzo di un browser si è in grado di collegarsi col CNR e visualizzare così su una mappa google i dati sotto forma di punti. Tramite diverse graduazioni di colore si può capire il movimento del terreno.

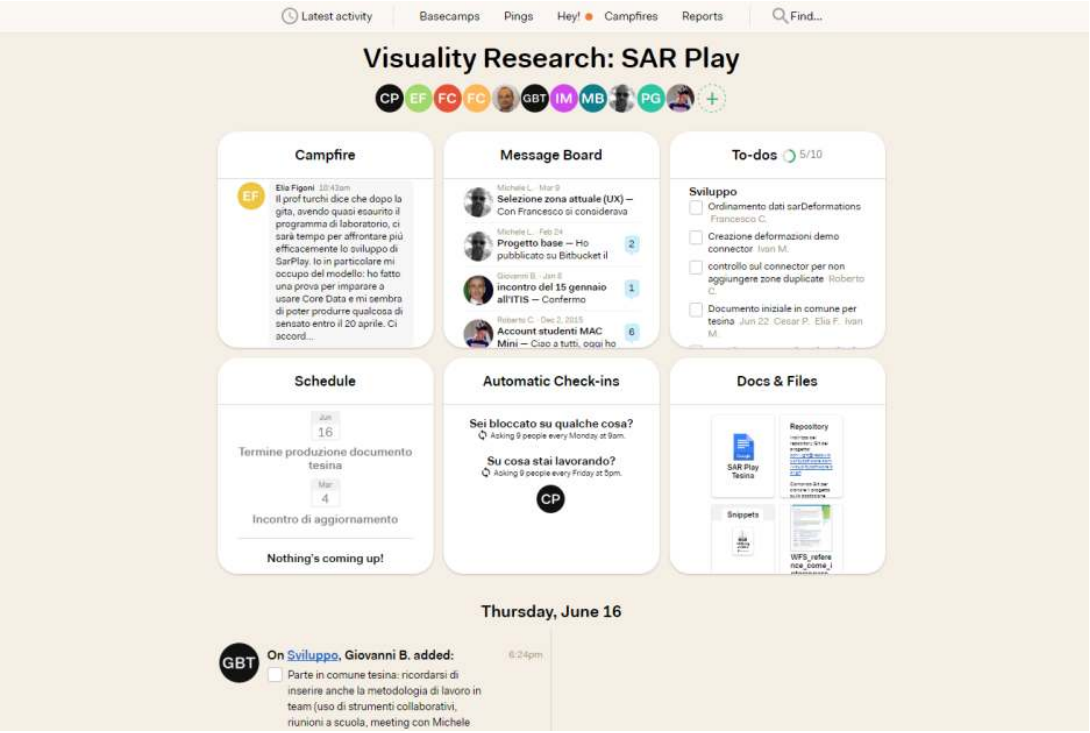
CAPITOLO 3
METODOLOGIA DI LAVORO IN TEAM

Dal momento che SARplay nasce come progetto di gruppo, sin da subito c'è stata la necessità di trovare un modo per rimanere in contatto tra tutti e di poter condividere tra tutti il progetto di lavoro. Oltre ai numerosi incontri tenuti durante l'anno, sia con i membri della Visuality, sia solo tra noi ragazzi, rimaneva comunque il problema della condivisione del materiale, con mezzi che non fossero le e-mail. Gli sviluppatori di software utilizzano strumenti come LinkedIN per poter rimanere in contatto facilmente e per poter condividere il materiale tramite le apposite repository. Nel nostro caso, la Visuality software usa da tempo Basecamp (https://3.basecamp.org/), pertanto ha creato uno spazio apposito per il progetto (un basecamp, appunto).

Tuttavia, il progetto vero e proprio contenente 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.

3.1 Basecamp

Il titolo di terzo livello è usato per identificare un sottoparagrafo. È consigliabile non utilizzare ulteriori suddivisioni dei sottoparagrafi. Eccezionalmente è disponibile un titolo di quarto livello privo di numerazione automatica.

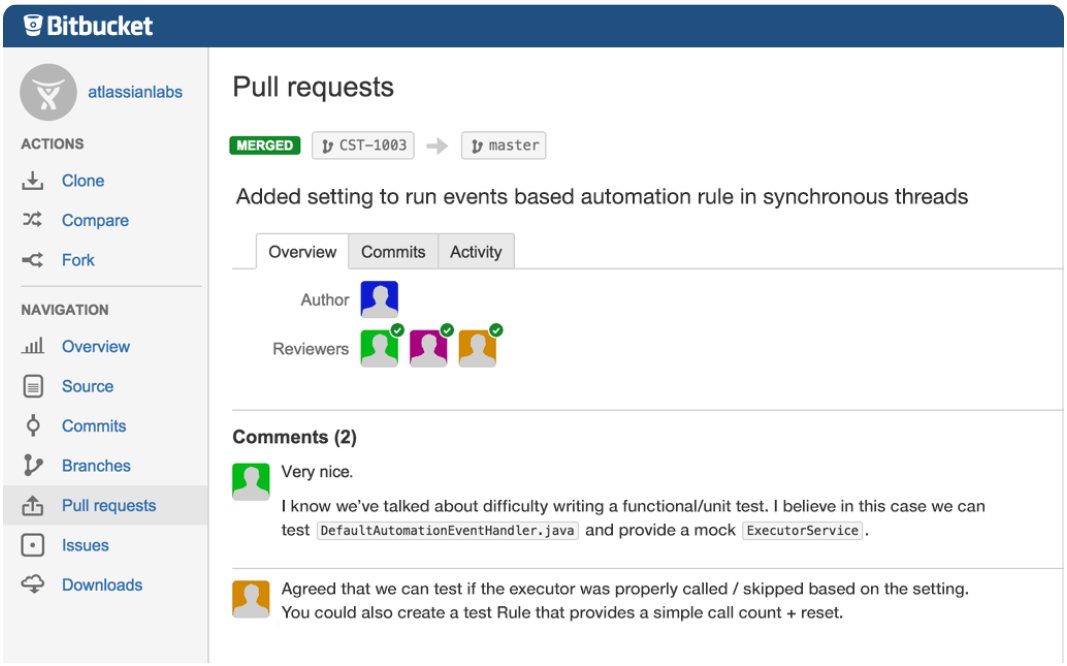


Basecamp è un valido strumento di gestione del progetto utilizzato per tenere sotto controllo i progetti non solo interni alla propria impresa/azienda, ma anche con clienti e partner. Basecamp è un grande strumento dotato di to-do list, la condivisione di file, blog, messaggistica in stile wiki e l'integrazione con il gruppo di eccellenti prodotti chat Campfire.
Le principali funzionalità di basecamp sono:

1. 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;
2. Hey: è una schermata contenente tutte le notifiche
3. Messages: è una chat tra tutti i membri del gruppo
4. 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.
5. 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

6. 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
7. Docs & files: è uno spazio nel quale vengono caricati file utili per il progetto corrente

3.2 Bitbucket



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 nel tempo, 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.

Git, considera i propri dati come una serie di istantanee (snapshot) di un mini filesystem. Ogni volta che l'utente effettua un commit, o salva lo stato del proprio progetto,

fondamentalmente fa un'immagine di tutti i file presenti in quel momento, salvando un riferimento allo snapshot. Se alcuni file non sono stati modificati, Git non li clona ma crea un collegamento agli stessi file 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 un nuovo progetto Git in due modi:

- Definire un nostro progetto preesistente come GIT Repository.
- Clonare un repository Git esistente da un altro server.

3.2.1 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.

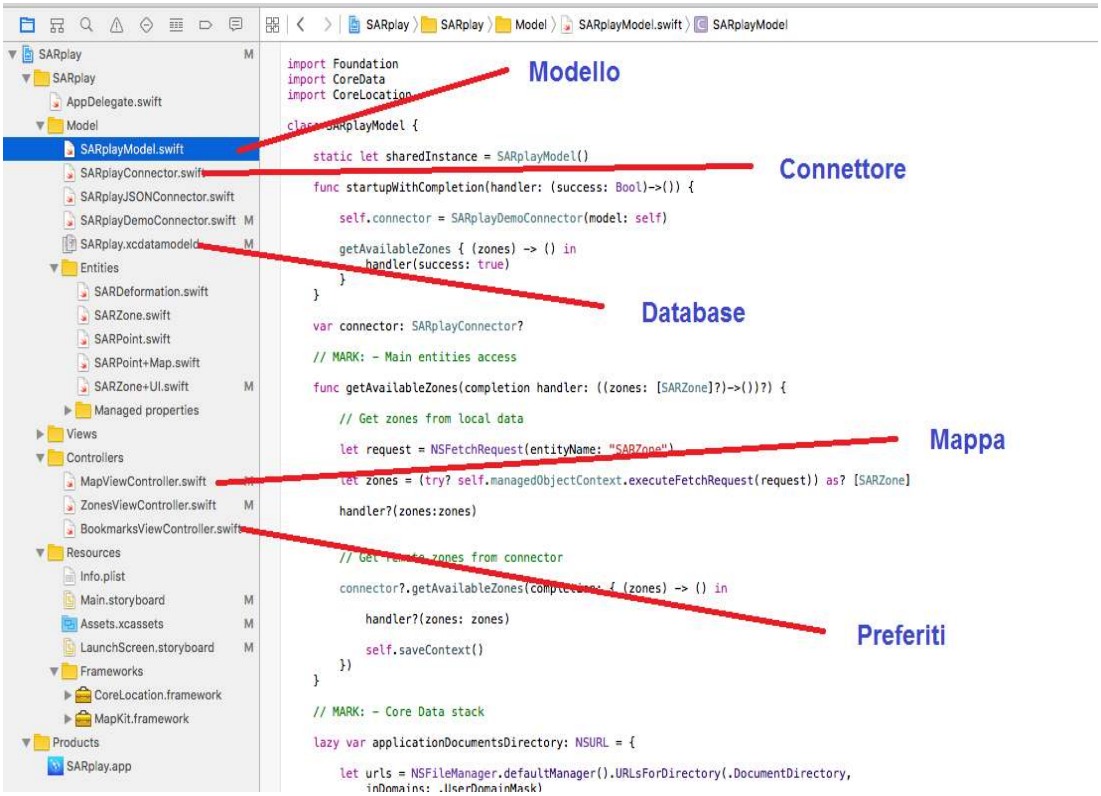
>> git status
Il modus operandi più diffuso è il seguente:

- 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")
- Per il push per pubblicare le proprie modifiche e renderle disponibili per chiunque

CAPITOLO 4
SCHEMA 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 un file json dal server, tramite una richiesta 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 saranno visibili i punti in base alla zona selezionata. È possibile scegliere due layout diversi (satellitare e mappa semplice)
- 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, raffigurato da una stella


Modello
Connettore
Database
Mappa
Preferiti

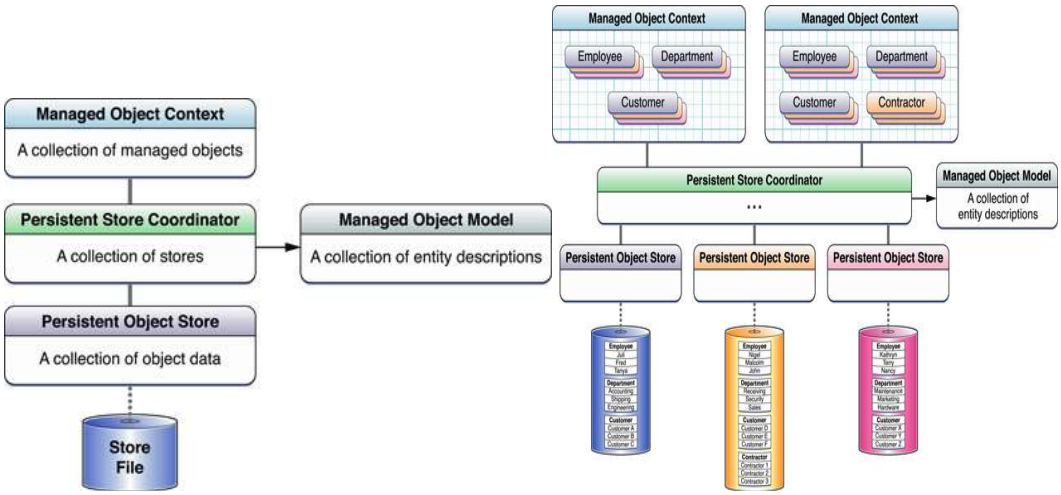
CAPITOLO 5
CORE DATA

Core Data consiste in 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 la tua applicazione iOS e la memorizzazione dei dati in memoria che puo essere secondo tre tipologie:** XML, binario o SQLite**. Per la memorizzazioneCore Data utilizza di default un database di tipo Relazionale(SQLite).**

5.1 Organizzazione di core data

Una tra le 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



5.1.1 NSManagedObjectContext

L'NSManagedObjectContext **si occupa della gestione delle informazioni contenute nella memoria** dell'applicazione, salvate tramite le funzionalità del Core Data. Esso si puo pensare come un pacchetto che contiene tutte le istanze delle varie NSManagedObject. 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 funzione permette di richiamare il context da qualsiasi classe dell'applicazione

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

5.1.2 Managed Objects

Gli Object consistono in delle istanze della classe NSManagedObject e possono essere rappresentate 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.

5.1.3 Store Coordinator

Lo Store Coordinato 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".

5.1.4 ObjectStore

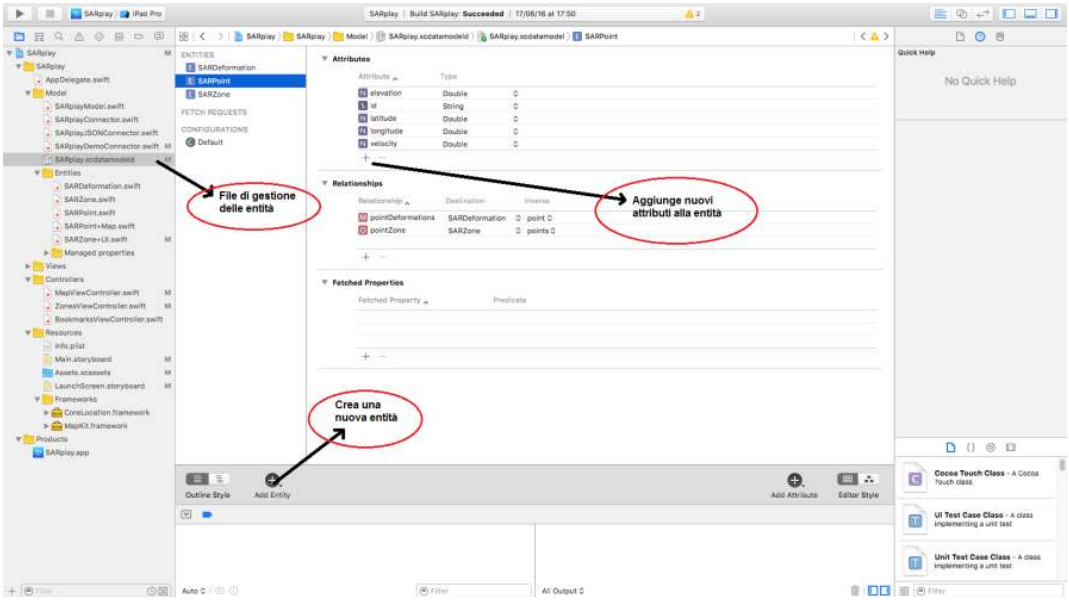
Rappresenta lo strato più basso della pila di Core Data, nel quale i dati vengono memorizzati. Vengono supportate le codifiche XML, binaria e SQLite. Lo sviluppatore non interagirà mai direttamente con questo oggetto.
Nella maggior parte dei casi si ha solo un ObjectStore.
Esistono due tipi di "Negozi": 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.

5.2 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.

5.2.1 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 aventi caratteristiche comuni. Ogni entità possiede i propri attributi, ossia gli elementi che permettono di definirla e di distinguere due istanze della stessa classe.**
Nell'immagine sotto rappresentata troviamo gli elementi che costituiscono un entità e le modalità per crearla.



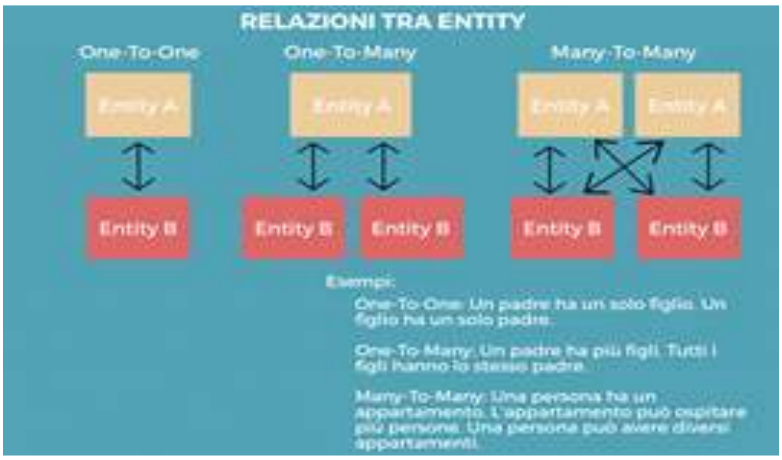
Le entità che verranno definite in Core Data, per essere tramutate in istanze di una determinata classe, necessitano di essere allocate tramite la creazione di una classe che implementa un costruttore per i suoi attributi. Quella che si andrà a realizzare non sarà che una sottoclasse della NSObjectClass.

```
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
}
```

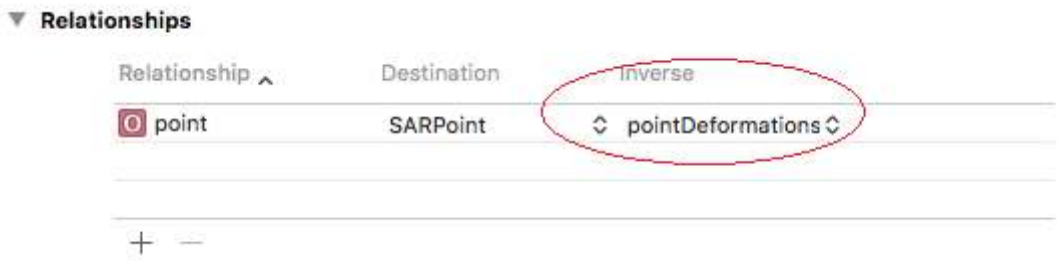
5.2.2 Relazioni

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 apsetto importante ossia la possibilità che essa sia inversa, e quindi bidirezionale, oppure semplicemente unidirezionale. L'inversibilità 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.



5.3 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.

5.3.1 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 altra 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()
            }
        }
    }
```

Una possibile altra 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.

5.3.2 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]
```

CAPITOLO 6
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.

6.1 Vantaggi Realm

Grazie a una serie di vantaggi e le ragioni abbiamo scelto di utilizzare Realm per lavorare con database nel Mobile App:

- 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.

6.2 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.

6.3 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!
```

6.4 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.

6.4.1 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
    }

}
```

6.4.2 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>()
```

6.5 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.

6.5.1 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:

1. 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()
```

2. 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)
```

Il salvataggio avviene mediante questo semplice codice:

```
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 che automaticamente, grazie alle associazioni, tutto verrà memorizzato in memoria.

6.5.2 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.

```
letpoints=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 outptus le zone preferite, ossia che valore di bookmarked_=true.

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

    let realm = Realm.getInstance()

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

RIFERIMENTI BIBLIOGRAFICI

Core Data:

- https://github.com/iascchen/SwiftCoreDataSimpleDemo
- https://www.xcoding.it/introduzione-al-core-data-in-swift/
- https://developer.apple.com/library/ios/documentation/DataManagement/Devpedia CoreData/persistentStoreCoordinator.html

Realm:

- https://www.raywenderlich.com/81615/introduction-to-realm
- https://realm.io/news/jesse-squires-core-data-swift/