UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA
IN

INFORMATICA

Progettazione di MicroAppEngine iOS:

Isolamento delle Componenti ed Estendibilità

RELATORI:

Prof. Genoveffa Tortora

Dott. Michele Risi

CANDIDATO:

Abagnale Giuseppe

Matr. 0512101956

ANNO ACCADEMICO 2017/2018

“Chiunque abbia perso la nozione del tempo mentre usava
un computer conosce la propensione a sognare, il bisogno di realizzare i
propri sogni e la tendenza a saltare i pasti.”

Tim Berners-Lee

Sommario

Introduzione

Nel lontano 1973 l'inventore Martin Cooper, da una strada di New York, fece la sua prima telefonata pubblica attraverso un telefono cellulare portatile. Il primo prototipo di telefono cellulare, il Dyna-Tac, pesava 1,5 kg e aveva una batteria che durava 30 minuti, ma che impiegava 10 ore a ricaricarsi. Quella prima telefonata, indirizzata a Joel S. Engel, capo della ricerca ai Bell Labs, rappresentò il momento fondamentale del passaggio tecnologico con cui si arrivò a raggiungere una persona invece di un luogo. Fu il prodotto della sua concezione di una comunicazione telefonica senza fili di tipo personale e portatile, distinta dalla telefonia per le automobili. In seguito Cooper raccontò che l'idea ispiratrice del telefono cellulare gli venne dalla visione del telefilm Star Trek in cui il Capitano Kirk usava un dispositivo analogo.

Dalla sua comparsa, il telefono cellulare ha usato diversi sistemi di funzionamento principali denominati "generazioni", basati su differenti tecnologie e standard di comunicazione, dai sistemi analogici degli anni 70'/90' (160/450/900 MHz) a quelli digitali basati su standard GSM, GPRS, UMTS/EDGE, 3G e 4G.

Il passaggio da analogico a digitale ha permesso di implementare oltre alla chiamata vocale, l'uso dei messaggi di testo SMS, registrare, visualizzare ed inviare foto e filmati, streaming audio e video, navigare in internet e spedire e-mail.

Nel corso degli anni, come la tecnologia avanza, la portabilità e il design dei telefoni cellulari vengono messi a disposizione di tutti. I software dei cellulari di ultima generazione sono veri e propri Sistemi Operativi in grado di gestire i più svariati applicativi come la posta elettronica, giochi, programmi complessi come fotoritocco, per il controllo del computer, per la protezione crittografica della conversazione etc.

I più recenti ed evoluti cellulari vengono chiamati Smartphone, in italiano "cellulare intelligente", è un dispositivo che abbina funzionalità proprie di

un telefono cellulare a quelle di gestore di dati personali per migliorare la produttività personale ed aziendale; costruito su una piattaforma di mobile computing, è caratterizzato da una elevata capacità computazionale e connettività rispetto ai suoi predecessori.

Uno degli aspetti peculiari per gli smartphone è la possibilità di installarvi ulteriori applicazioni, aggiungendo così nuove funzionalità. Tali dispositivi si differenziano inoltre per l'adozione di API (application programming interfaces) avanzate, permettendo alle applicazioni di terze parti una migliore integrazione con il sistema operativo del dispositivo.

Negli ultimi anni si è notato un rapido avanzamento nel campo dell'innovazione dei sistemi operativi e delle piattaforme applicative orientate ai dispositivi mobile, dato il costante e rapido incremento della capacità computazionale dei processori, reso possibile grazie alle fiorenti tecnologie disponibili. Tutto questo ha permesso di definire un livello ancora più alto di "intelligenza" nei dispositivi, dove la geo-localizzazione, basata su differenti tecnologie come GPS, Wi-Fi, GSM, si presenta ora come una funzionalità indispensabile; i sistemi operativi associati a questa fascia di prodotti offrono tutti un market place o uno store che permettono di scaricare applicazioni di ogni tipo e per qualsiasi utilizzo, cambiando radicalmente la percezione che l'utente finale ha di tali dispositivi: da semplici cellulari si giunge a considerarli veri e propri computer in miniatura.

Un passo immediatamente successivo sarà caratterizzato dalla presenza di un livello di consapevolezza riferita al contesto applicativo, maturata all'interno dei dispositivi per poter non solo tener traccia dei dati personali dell'utente, ma soprattutto tracciare e riconoscere il comportamento individuale, riuscendo quindi ad anticipare le intenzioni decisionali dell'utilizzatore. Queste ultime potenzialità descritte vanno a costituire la parte veramente "smart" dei dispositivi; in questo senso questi ultimi assumono anche le sembianze di sensori a tutti gli effetti.

In un ambiente in continua evoluzione, dove le potenzialità sembrano essere infinite, sono emerse le prime idee relative alle enormi possibilità offerte da piattaforme software consolidate, applicate a dispositivi con tecnologia pari ai ben noti computer desktop, iniziando ad investire nella realizzazione di sistemi operativi mobile come supporto ai moderni smartphones; tra i più comuni SO si ricordano Google's Android, Apple's iOS, Microsoft's Windows Phone.

Il successo di tali investimenti è reso evidente dal numero sempre maggiore di smartphone connessi ad Internet rispetto ai computer. Lentamente ci si sta avvicinando ad una situazione in cui l'adozione di tali dispositivi mobili "intelligenti" divenga la norma per i normali internauti, dato che molte delle azioni di routine che ora sono caratteristiche dei computer desktop e laptop diverranno molto presto possibili per tutti gli smartphone, e le nuove applicazioni sviluppate per questi ultimi incontrano perfettamente le esigenze degli utenti che non usano computer.

In quest'ottica si pone MicroApp, sviluppata presso l'Università di Salerno, un'applicazione per il sistema operativo Android che permette di comporre, generare ed eseguire applicazioni a partire da funzionalità semplici ed atomiche.

MicroApp e il suo modello di esecuzione chiamato MicroAppEngine, sono state progettate per gli utenti che si concentrano solamente sul comporre una microApp desiderata, ignorando il modo in cui queste azioni vengano eseguite.

L'utente compone l'applicazione tramite un editor visuale, che permette l'inserimento e la composizione di varie componenti, dove ogni componente rappresenta una funzionalità disponibile sul dispositivo. Nella seconda fase l'utente seleziona la microApp[1] composta, e la esegue sfruttando tutte le funzionalità per la quale era stata implementata.

La scelta di sviluppare MicroApp per iOS è dovuta al fatto che al giorno d'oggi molti utenti scelgono di acquistare un device iOS per un motivo

[1] Con i termini "MicroApp" e "microApp" si farà riferimento, rispettivamente, all'intero progetto software oggetto della tesi e a una tipica applicazione personalizzata creata dall'utente.

molto semplice: l'usabilità. Infatti iOS dispone di un'interfaccia user-friendly molto semplice e intuitiva che permette all'utente di avere un sistema efficiente, sicuro, veloce e confortevole. Inoltre iOS è un sistema chiuso e a differenza di Android, questo non permette di apportare modifiche al sistema da parte degli utenti, proprio perché si vuole mantenere un sistema omogeneo e sicuro su tutti i dispositivi.

Pertanto, l'obiettivo della tesi riguarda la progettazione e implementazione del motore MicroAppEngine per l'esecuzione delle microApp per dispositivi iOS, seguendo la linea politica di Apple, dando maggiore attenzione alle necessità degli utenti finali, senza però dimenticare di evidenziare le potenzialità di estensione del progetto. In particolare, MicroAppEngine iOS si rinnova rispetto alla versione Android: esso offre adesso un maggiore isolamento delle componenti rispetto al sistema, un meccanismo di estendibilità che rende molto semplice l'aggiunta di nuove componenti ed una gestione e un flusso dei dati più efficiente. Questo è l'aspetto principale su cui si basa il lavoro svolto.

Di seguito è mostrata l'organizzazione del lavoro di tesi. Nel primo capitolo è fornita un'introduzione sulle tecnologie utilizzate, verrà fatta una breve introduzione al sistema operativo iOS, l'ambiente di sviluppo e il linguaggio utilizzato per lo sviluppo di MicroApp iOS. Il secondo capitolo tratterà dello studio sulla fattibilità, nello specifico andremo a parlare della sicurezza e della complessità del sistema iOS e della fase di deploy di un'applicazione non nativa. Il terzo capitolo descrive le fasi di progettazione e implementazione del MicroAppEngine, mentre nel quarto capitolo sono forniti gli aspetti di isolamento ed estendibilità delle componenti mostrando i dettagli di quelle implementate. Il quinto capitolo mostra un esempio di microApp, di come dovrebbe essere implementata e testata su un emulatore o su un dispositivo reale. Infine, le conclusioni e i lavori futuri concluderanno la tesi.

Capitolo 1 - Tecnologie utilizzate

In questo capitolo descriveremo le tecnologie utilizzate per lo sviluppo di MicroAppEngine, parleremo delle funzionalità del sistema iOS, dell'ambiente di sviluppo e il linguaggio utilizzato.

1.1 Il Sistema iOS

Il sistema iOS rappresenta la versione mobile del sistema desktop Mac OS X appartenente alla ben nota azienda internazionale produttrice di hardware-software; è utilizzato in tutti i dispositivi mobili della medesima marca di fascia alta (iPhone, iPod touch, iPad) e mette a disposizione le tecnologie necessarie per implementare applicazioni web e/o native, oltre che dare la possibilità agli utenti di usufruire di un'innumerevole quantità di applicazioni gratuite o a pagamento pronte per essere installate.
Il successo di questo sistema è stato determinato dalla facilità d'uso e semplicità, due dei maggiori punti di forza, che da sempre hanno reso possibile un utilizzo immediato, rapido, anche grazie ad un'ottima fluidità del sistema; inoltre i modi in cui si è stata ideata l'interfaccia grafica e si sono gestite le interazioni con i componenti hardware per supportare il touchscreen, hanno aumentato il grado di intuitività nei gesti eseguiti in un utilizzo giornaliero. E' opportuno ricordare che tali caratteristiche generali non consentono un'alta personalizzazione e manipolazione del dispositivo, limitando la libertà dell'utente finale. Però, così facendo, tale piattaforma mobile risulta essere molto sicura e robusta, rispecchiando una scelta progettuale, non ché una linea di percorrenza dell'azienda produttrice ben precisa.
Come già accennato in precedenza è possibile progettare e sviluppare applicazioni native mediante interfacce, tools e risorse, dove queste ultime sono molto vincolanti e stringenti; l'aspetto più interessante è la possibilità di giungere ad un livello di performance e ottimizzazione formidabile

tramite l'interfacciamento con strutture e tecnologie di basso livello, perseguendo metodi più rischiosi e difficoltosi, oppure affidarsi a metodi "preconfezionati" che risultano sicuramente più rapidi, ma meno personalizzabili.
Per quanto riguarda gli aspetti architetturali si può notare una grande affinità presente fra l'ambiente desktop e quello mobile, mettendo in luce un buon grado di analisi e organizzazione, ma soprattutto percependo la volontà dell'azienda produttrice di procedere, nel corso degli anni, verso un'unificazione dei due ambienti, al fine di pervenire ad una eterogenea gamma di dispositivi, dal punto di vista dell'equipaggiamento hardware ed obiettivi specifici, nei quali è in esecuzione il medesimo sistema software sovrastante.

1.1.1 Struttura Architetturale

L'architettura di iOS è simile a quella basica in Max OS X e come essa, è strutturata in una serie di quattro livelli di astrazione, o layers, ognuno dei quali implementa funzionalità ben specifiche, per rendere semplice la scrittura di applicazioni che funzionano in modo coerente su dispositivi con differenti capacità hardware.
Nel livello più alto, il sistema operativo agisce come intermediario fra l'hardware sottostante e l'interfaccia grafica, mentre le varie applicazioni installate comunicano con i livelli sottostanti attraverso un ben determinato set di interfacce, aumentando considerevolmente la protezione delle applicazioni da eventuali modifiche hardware. Il livello più basso del sistema ospita i servizi e le tecnologie fondamentali da cui tutte le applicazioni dipendono.
La maggior parte delle interfacce di sistema sono rese disponibili in pacchetti speciali chiamati framework (1). Un framework consiste in un direttorio contenente una libreria dinamica di funzioni e risorse (header file, immagini, etc.) a supporto di essa. In aggiunta ai frameworks, Apple

rende disponibili alcune tecnologie nella forma di librerie dinamiche in formato standard; molte di esse sono appartenenti al livello più basso del sistema operativo e derivano da tecnologie Open Source, come naturale conseguenza del fatto che iOS, similmente al corrispondente sistema desktop di casa Apple, sia basato sulla piattaforma Unix.

1.1.2 Livelli Architetturali

Procedendo in un'analisi più accurata circa la composizione e la coesione delle stratificazione che vanno a formare l'architettura di iOS, si possono individuare quattro differenti livelli di astrazione, come mostrato in figura.

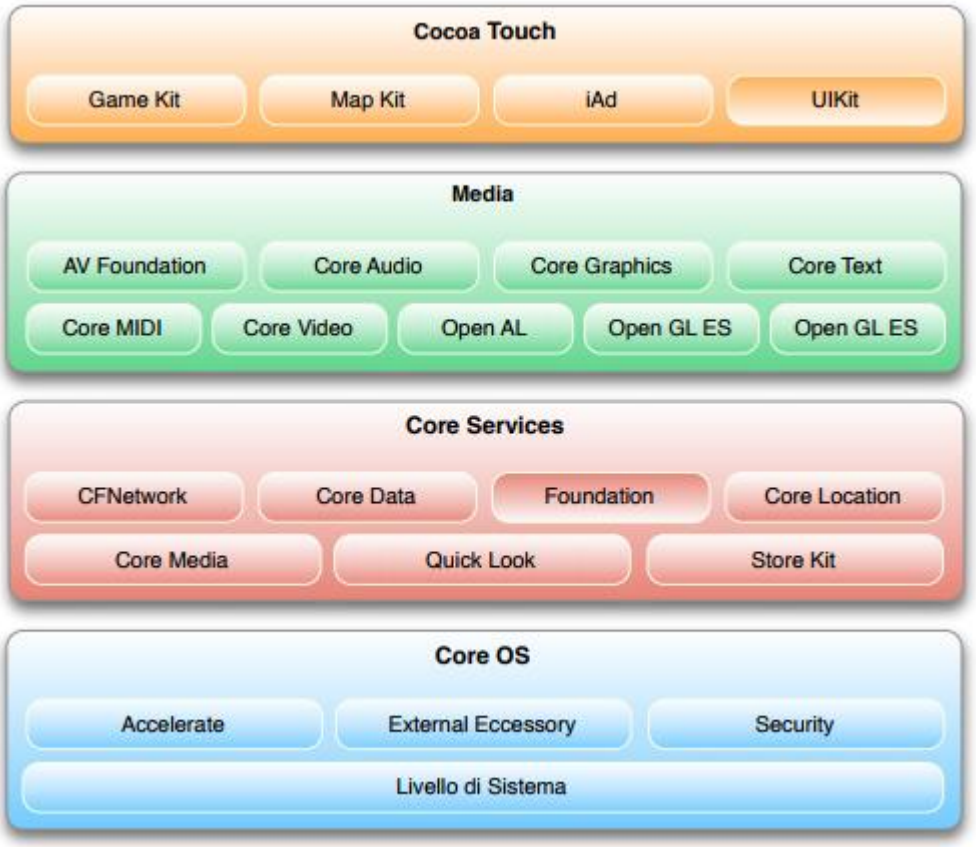

Figura 1.1 Livelli Architetturali di iOS

- **Core Os:** è lo strato che permette di lavorare a diretto contatto con l'hardware sottostante ed è considerato il vero cuore del sistema operativo; infatti in esso sono presenti gli elementi considerati fondamentali, utilizzati poi dalle tecnologie di livelli sovrastanti. Uno dei compiti principali di questo strato è quello di gestione della potenza, ovvero di gestire in maniera più efficace possibile l'energia messa a disposizione della batteria del dispositivo, senza sprechi.

- **Core Services:** come suggerisce il nome assegnato, questo livello contiene i servizi di sistema fondamentali, utilizzati da tutte le applicazioni, spesso considerate utility. Le tecnologie chiave presenti si possono riassumere nelle aree di interesse che comprendono la programmazione concorrente, il commercio elettronico, la gestione e memorizzazione dei dati all'interno di database e presentazione/manipolazione delle informazioni ricevute o trasmesse.

- **Media:** rappresenta lo strato che contiene tutte le funzionalità e le librerie per la gestione di video e audio. Mediante le tecnologie presenti, si è orientati verso la creazione della migliore esperienza multimediale raggiungibile su un dispositivo mobile. In questo strato, infatti, sono ubicate le librerie OpenAl per la gestione e manipolazione di flussi audio e le famose librerie OpenGL ES per produrre grafica 2D con animazioni anche molto complesse. Queste ultime sono le sorelle minori delle librerie OpenGL che permettono anche di produrre grafica 3D di altissimo livello. Le OpenGL ES sono state strutturate per essere utilizzate in sistemi embedded come i dispositivi telefonici ed il suffisso ES rappresenta l'acronimo Embedded System.

- **Cocoa Touch:** rappresenta lo strato più vicino all'applicazione utente e i frameworks di questo livello supportano direttamente le applicazioni basate su iOS. Esso si occupa della gestione del touch e multi-touch, interpretando i differenti gesti (gestures) compiuti dall'utente finale mediante i gesture recognizer, oggetti collegati alle view (a loro volta definite come schermate visibili sul video), utilizzati per rilevare i tipi più comuni di gestures, come lo zoom-in o la rotazione di elementi; non appena collegati si può stabilire quale comportamento associare ad essa. Oltre alla gestione del touch, molti dei frameworks del livello Cocoa Touch contengono specifiche classi genericamente denominate View Controller, per poter visualizzare interfacce standard di sistema. Tali componenti rappresentano particolari tipi di controller molto utilizzati per presentare e gestire un insieme di view; giocano un ruolo importante nella progettazione ed implementazione di applicazioni iOS perché forniscono un'infrastruttura per gestire i contenuti correlati alle view e coordinare la comparsa/scomparsa di queste ultime. Un'ulteriore caratteristica di questo livello è data dal supporto a due differenti modalità, in cui utilizzare le notifiche, le quali danno la possibilità di avvisare l'utente di nuove informazioni mediante un segnale sonoro o visivo. Infine, molto importante per gli argomenti correlati alle prestazioni di sistema, è il pieno supporto dato al Multitasking anche ad alto livello architetturale, per poter coordinare ed impostare azioni tipiche del livello applicativo, come la ricezione di notifiche, il passaggio da uno strato attivo ad uno passivo.

1.2 L'ambiente di sviluppo

Xcode è un ambiente di sviluppo integrato IDE (Integrated development environment) sviluppato da Apple Inc. per agevolare lo sviluppo di applicazioni per Mac OS X e iOS. Supporta la compilazione incrementale

ed è in grado di compilare il codice mentre viene scritto, in modo da ridurre il tempo di compilazione.

1.2.1 Struttura Xcode

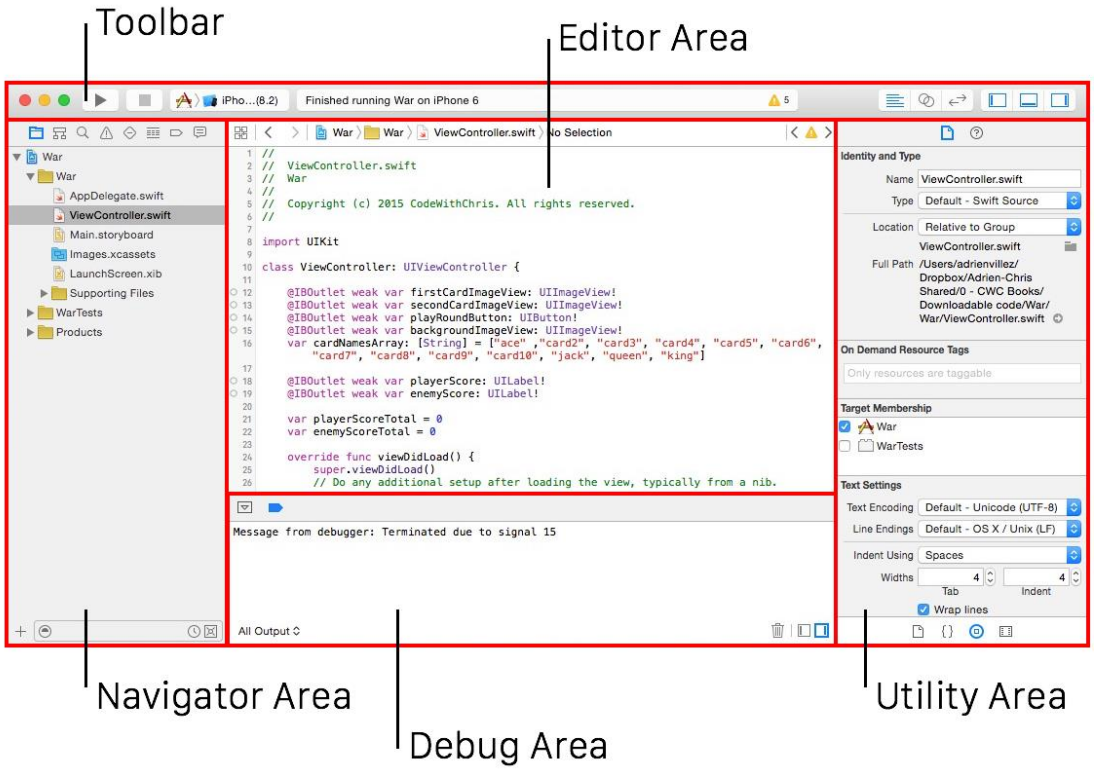

Figura 1.2 Interfaccia di Xcode

L'interfaccia dell'ambiente di sviluppo di Xcode è così formata:

- **Barra degli strumenti:** posizionata nella parte superiore della finestra, include i controlli di base per la costruzione di un progetto e per l'elaborazione delle diverse sezioni dell'area di lavoro. A sinistra sono riportati i pulsanti Run e Stop, che avviano e interrompono la simulazione dell'App.

  A fianco a questi si trova il selettore "Set the Active Scheme", per decidere con quale tipologia di device deve avviarsi il simulatore. Infine, il pulsante Organaizer apre la finestra omonima, che riguarda

elementi relativi a più progetti, per esempio la gestione dei device, la documentazione e i repository di controllo delle sorgenti. Si può nascondere la barra degli strumenti utilizzando il menu, ma si suggerisce di averla sempre a disposizione perché è particolarmente utile.

- **Area Navigator:** il pannello di sinistra, che può essere visualizzato o nascosto dal pulsante Show Navigator, permette di esaminare il contenuto del progetto. In questa colonna si può selezionare, tramite una barra orizzontale, la sezione alla quale si vuole accedere. Le sezioni sono:
  - Project, che mostra i file sorgente e i file risorse del progetto ed è pertanto quello più importante e maggiormente impiegato.
  - Symbol Navigator, dove vengono elencati tutte le tipologie di metodi utilizzate nel progetto.
  - Find Navigator, permette di fare una ricerca di file in tutto il progetto.
  - Issue Navigator, sezione molto importante per la fase di testing. Qui vengono riportati tutti gli errori in fase di compilazione o simulazione.
  - Test Navigator, per avviare il test dell'App.
  - Debug Navigator, in questa sezione di testing viene mostrato il peso che ha l'App nel simulatore. Vengono prese in considerazione: l'utilizzo della CPU, la RAM utilizzata, il peso dell'App sull'Hardisk e sul Web.

- **Area Editor:** La parte principale dell'area di lavoro è costituita dall'Editor, l'unica vista che non può essere nascosta. Il suo contenuto dipende dal file che si deve modificare. Si supponga per esempio di selezionare un file sorgente: in questo caso si visualizza un tipico editor di codice sorgente. La selezione di un file GUI

comporta invece che l'area diventi un editor GUI visuale. Il pulsante Editor della barra degli strumenti consente di selezionare tre modalità di visualizzazione del pannello di editing:

Standard, che corrisponde all'editor predefinito per il tipo di file selezionato;

Assistant, che mostra i file collegati fianco a fianco;

Version, che utilizza il controllo sorgente per mostrare le versioni corrente e storiche del file, una a fianco dell'altra. Questa terza modalità consente di individuare il "responsabile" di ogni riga di codice, oppure riportare una serie di commenti inseriti nel codice.

L'area Editor contiene inoltre una barra di accesso rapido in stile Breadcrumb[2] che mostra la gerarchia dell'oggetto che si sta elaborando; per un file sorgente, la barra mostra per esempio il percorso "progetto, gruppo, file, metodo". Ogni elemento della barra corrisponde a un menu a comparsa che permette di navigare altri punti di interesse collegati o elaborati di recente.

- **Area Utility:** la selezione a destra dell'area di lavoro del progetto contiene utility che forniscono viste dettagliate e modificano determinate soluzioni dell'area Editor. La barra degli strumenti nella parte superiore dipende dal file da modificare e riporta strumenti differenti nel pannello File Inspector. L'area mostra sempre informazioni di base relative al file selezionato, cui si aggiunge un aiuto rapido legato alla selezione corrente. Per i file GUI sono disponibili inspector che elaborano singole entità di classi degli oggetti UI, i rispettivi attributi da configurare, la loro dimensione, il loro layout e i collegamenti al codice sorgente. Nella parte inferiore

[2] Le Breadcrumb sono una tecnica di navigazione usata nelle interfacce utente. Il loro scopo è quello di fornire agli utenti un modo di tener traccia della loro posizione in documenti o programmi.

dell'area Utility si trova il pannello Library, che consente di accedere con facilità (clic e trascina) a Snippet di codice, a oggetti dell'interfaccia utente e altro ancora.

- **Area Debug:** In fondo alla finestra, sotto l'area Editor e tra le aree Navigator e Utility, è presente un pannello che visualizza le informazioni del Debug in fase di esecuzione di un'App. La sua piccola barra degli strumenti include un menu a tendina per passare da una vista di debug che ispeziona la memoria quando si incontra un Breakpoint, a una vista che mostra l'output prodotto dall'applicazione mobile oppure a una vista che presenta entrambe le opzioni.

Infine, il grosso pulsante Run consente l'avvio dell'applicazione. Bisogna verificare che l'opzione "Scheme" sia il dispositivo prescelto per la simulazione e poi fare clic su Run. L'area di stato mostra una barra di progresso che si riempie durante la build dei file e il bundle dell'applicazione; al termine dell'operazione si avvia il Simulatore iOS e l'applicazione viene eseguita.

1.3 Linguaggio Swift

Swift dall'inglese "rondine" e "rapido", è un linguaggio di programmazione object-oriented per sistemi OS X e iOS che pian piano andrà a sostituire Objective-C.
Swift (2) (3) (4) è un linguaggio di programmazione conciso, tipizzato, efficiente e facile da imparare. Esso incorpora le migliori caratteristiche e funzionalità appartenenti ai moderni linguaggi di programmazione, come C#, JavaScript, Python, Rust e GO.

Rispetto ad Objective-C, il codice risulta più conciso e leggibile infatti sono scomparse le parentesi quadre, e i punto e virgola a fine riga sono opzionali (vedere Fig. 1.3).

E' stato introdotto il concetto di type inference[3] grazie al quale non è necessario specificare esplicitamente il tipo di variabili e costanti se questo può essere determinato dal contesto. Non è più necessario avere un header file ed è in implementation file separato per ogni classe, ma la definizione di queste avviene in un unico file.

```
1  func sayHelloWorld() -> String {
2      return "hello, world"
3  }
4  println(sayHelloWorld())
5  // prints "hello, world"
```

Figura 1.3 Esempio di linguaggio Swift

Swift eredita tante moderne features da altri linguaggi, come gli Optional, le collezioni, gli array che assomigliano alle List di C#, e una sintassi molto concisa per la creazione dei dizionari. Altre novità sono le Closure, che prendono il posto dei blocchi in Objective-C, e i Generic, classi che consentono di definire comportamenti e azioni senza decidere a priori su che tipo di dati devono operare.

Infine Apple afferma che Swift risulta essere due volte più veloce rispetto ad Objective-C e che c'è ancora margine per il miglioramento. Il linguaggio è ancora abbastanza giovane e Apple ad ogni release aggiungerà nuove caratteristiche.

[3] Type inference si riferisce alla deduzione automatica di un tipo di dato o di una espressione in un linguaggio di programmazione

Capitolo 2 - Studio di Fattibilità

Apple ha progettato la piattaforma iOS dando massima priorità alla sicurezza (vedere Fig. 2.1).
Ogni singolo dispositivo iOS unisce software, hardware e servizi progettati per lavorare insieme al fine di garantire la massima sicurezza e un'esperienza utente semplice e intuitiva. iOS non protegge solo il dispositivo e i dati in esso contenuti quando il sistema è a riposo, bensì protegge l'intero ecosistema, incluse tutte le operazioni che l'utente esegue localmente, su rete e con i principali servizi Internet.

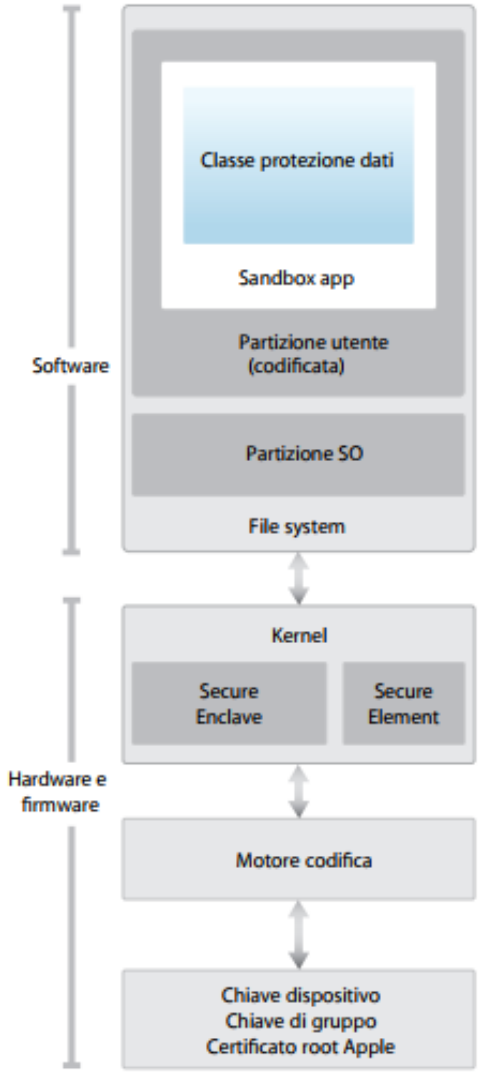

Figura 2.1 Diagramma dell'architettura di sicurezza di iOS

2.1 Sicurezza del sistema iOS

La sicurezza del sistema è stata progettata in modo integrato affinché sia il software sia l'hardware siano sicuri in tutti i componenti principali di ogni singolo dispositivo iOS. Ne fanno parte la procedura di avvio, gli aggiornamenti software e Secure Enclave (5). Questa architettura occupa una posizione cardine nella sicurezza di iOS e non compromette la facilità d'uso del dispositivo.

2.1.1 Procedura di avvio sicuro

Ogni passo del processo di avvio contiene componenti che sono stati firmati digitalmente da Apple attraverso opportuna codifica per garantire l'integrità e a cui viene consentito procedere solo dopo aver verificato la catena di affidabilità. Tale processo include i bootloader, il kernel, le estensioni del kernel e il firmware baseband. Questa catena di avvio sicuro aiuta a garantire che i livelli più bassi del software non vengano alterati.
Quando un dispositivo iOS è acceso, il processore per le applicazioni esegue immediatamente il codice dalla memoria di sola lettura conosciuta come ROM di avvio.
Il codice della ROM contiene la chiave pubblica dell'autorità di certificazione root di Apple, utilizzata per verificare che il bootloader iBoot sia stato firmato da Apple prima di consentirne il caricamento. Dopo aver concluso le proprie attività, iBoot controlla ed esegue il kernel iOS.

2.1.2 Autorizzazione software di sistema

Apple rilascia periodicamente aggiornamenti software volti a risolvere sul nascere eventuali problematiche di sicurezza e a fornire nuove funzionalità; gli aggiornamenti sono resi disponibili nello stesso momento per tutti i dispositivi. Gli utenti ricevono notifiche sugli aggiornamenti sul dispositivo e attraverso iTunes; gli aggiornamenti vengono forniti in modalità wireless

e ne viene consigliata l'installazione agli utenti sottolineando l'importanza dei miglioramenti apportati alla sicurezza.
Per evitare che i dispositivi possano venire riportati a versioni precedenti prive degli ultimi aggiornamenti di sicurezza, iOS utilizza un processo chiamato "Autorizzazione software di sistema". Se fosse possibile riportare il software a una versione precedente non più aggiornata, un malintenzionato che si fosse illegalmente impossessato del dispositivo potrebbe facilmente trarre vantaggio da un problema di vulnerabilità del sistema risolto in una versione più recente.
Durante un aggiornamento iOS, si collega al server Apple di autorizzazione per l'installazione e invia a esso un elenco di misurazioni di codifica per ogni singola parte del pacchetto di installazione che deve essere installato, un valore anti-replay casuale e l'ID unico del dispositivo (ECID). Il server di autorizzazione verifica l'elenco di misurazioni che è stato presentato e lo paragona alle versioni in cui è stata permessa l'installazione; se trova una corrispondenza, aggiunge l'ECID alla misurazione e firma il risultato. Autorizzando e firmando solo le misurazioni conosciute, il server assicura che l'aggiornamento avvenga esattamente secondo i parametri dettati da Apple.
Questi passi servono a garantire che l'autorizzazione sia legata a un dispositivo specifico e che una versione precedente di iOS non possa venire copiata da un dispositivo all'altro.

2.1.3 Secure Enclave

Secure Enclave è un coprocessore che fornisce tutte le operazioni codificate per la gestione della chiave di protezione dei dati e preserva l'integrità della protezione dei dati anche qualora sia stato compromesso il kernel. La comunicazione tra Secure Enclave e il processore delle applicazioni è isolata in una casella di posta guidata da interrupt (interrupt-driven) e in buffer di dati di memoria condivisa.

Quando il dispositivo si avvia, viene creata una chiave effimera, legata all'UID e utilizzata per codificare la porzione di Secure Enclave nello spazio di memoria del dispositivo.
Inoltre, i dati salvati nel file system da Secure Enclave sono codificati con una chiave legata all'UID e con un contatore anti-replay.
Secure Enclave si occupa dell'elaborazione delle impronte digitali ottenute dal sensore Touch ID, di determinare se esiste una corrispondenza con le impronte registrate e di consentire l'accesso o gli acquisti per conto dell'utente.

2.2 Codifica e protezione dati

La procedura di avvio sicuro, la firma del codice e la sicurezza del processo di runtime contribuiscono a garantire che solo il codice e le app affidabili possano venire eseguiti su un dispositivo. iOS dispone di funzionalità aggiuntive di codifica e protezione dei dati volte a salvaguardare i dati dell'utente, anche quando sono state compromesse altre parti dell'infrastruttura di sicurezza (ad esempio su un dispositivo con modifiche non autorizzate).

2.2.1 Protezione dati dei file

Apple usa una tecnologia chiamata "Protezione dati" per proteggere ulteriormente i dati archiviati nella memoria flash sul dispositivo. La protezione dati consente al dispositivo di rispondere a eventi comuni quali le chiamate in entrata, ma abilita anche un alto livello di codifica per i dati utente. Le app chiave del sistema utilizzano di default la protezione dati e le app di terze parti installate su iOS 7 o versione successiva ricevono questa protezione automaticamente.
La protezione dati è implementata creando e gestendo una gerarchia di chiavi e costruisce sull'hardware tecnologie di codifica integrate in ogni singolo dispositivo iOS. La protezione dati è controllata per ogni singolo

file e assegna a ognuno di essi una classe; l'accessibilità dei file dipende dallo stato, sbloccato o meno, delle chiavi della classe a cui appartengono.
Ogni volta che viene creato un file sulla partizione dati, la protezione dati crea una nuova chiave a 256 bit (la chiave "per file") e la consegna al motore AES hardware, che utilizza la chiave per codificare il file mentre viene scritto nella memoria flash.
La chiave per file è cifrata con una delle diverse chiavi di classe, a seconda delle circostanze in base alle quali il file deve essere accessibile. La chiave per file cifrata è memorizzata nei metadati del file.
Quando viene aperto un file, i suoi metadati sono decodificati con la chiave del file system, rivelando la chiave per file cifrata e una nota sulla classe che la protegge. La cifratura della chiave per file viene tolta con la chiave di classe, quindi fornita al motore AES hardware che a sua volta decodifica il file mentre viene letto dalla memoria flash. La gestione delle chiavi dei file cifrate avviene interamente in Secure Enclave; la chiave per i file non viene mai esposta direttamente al processore per le applicazioni. All'avvio, Secure Enclave negozia una chiave effimera con il motore AES.
Quando Secure Enclave rimuove la cifratura dalle chiavi di un file, queste vengono di nuovo cifrate con la chiave effimera e inviate di nuovo al processore per le applicazioni.
I metadati di tutti i file nel file system sono codificati con una chiave casuale, creata quando iOS viene installato per la prima volta o quando il dispositivo viene inizializzato dall'utente. La chiave del file system è archiviata in Effaceable Storage. Dato che questa chiave è archiviata sul dispositivo, non è utilizzata per garantire la riservatezza dei dati; è piuttosto stata progettata per poter essere cancellata velocemente su richiesta dall'utente oppure da un amministratore. La cancellazione della chiave secondo questa modalità rende tutti i file inaccessibili a causa della codifica.

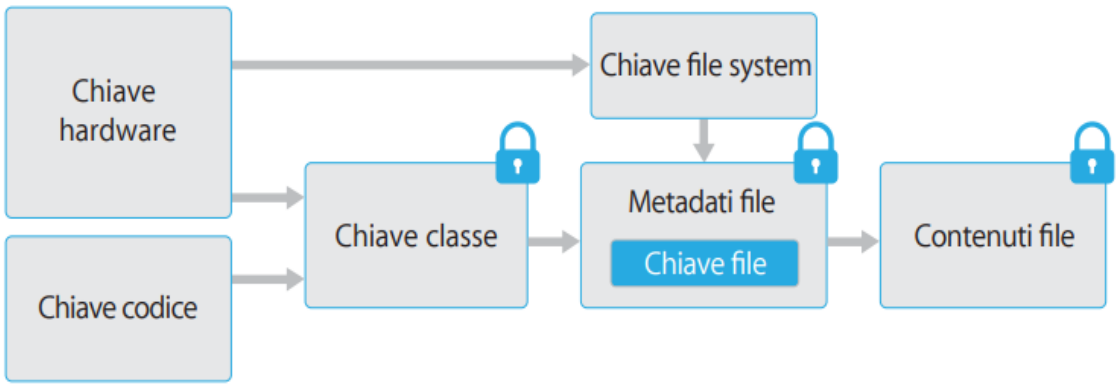

Figura 2.2 Gerarchia delle Chiavi di protezione

Il contenuto di un file è codificato con una chiave per file, che viene cifrata con una chiave di classe e archiviata nei metadati di un file, a sua volta codificato con la chiave del file system. La chiave di classe è protetta con l'UID dell'hardware e per alcune classi, con il codice dell'utente. Questa gerarchia fornisce flessibilità e prestazioni ottimale. Ad esempio, per modificare la classe di un file occorre solo cifrare nuovamente la sua chiave per file; la modifica del codice cifra di nuovo la chiave di classe.

2.2.2 Classi di protezione dati

Quando viene creato un file nuovo su un dispositivo iOS, a esso viene assegnata una classe dall'app che l'ha creato. Ogni classe utilizza diverse politiche che stabiliscono quando i dati sono accessibili.

Protezione completa
(NSFileProtectionComplete): la chiave di classe è protetta da una chiave derivata dal codice utente e dall'UID del dispositivo. Poco dopo il blocco del dispositivo da parte dell'utente, la chiave di classe decodificata viene scartata, così da rendere tutti i dati in tale classe inaccessibili finché l'utente non inserisce di nuovo il codice o sblocca il dispositivo tramite Touch ID.

Protetto se non è aperto
(NSFileProtectionCompleteUnlessOpen): in alcuni casi i file devono essere scritti mentre il dispositivo è bloccato, come ad esempio nel caso di un allegato e-mail che viene scaricato in background. Questo comportamento è ottenuto grazie all'uso della crittografia a curva ellittica.

Protetto fino alla prima Autenticazione Utente
(NSFileProtectionCompleteUntilFirstUserAuthentication): questa classe si comporta nello stesso modo di "Protezione completa", l'unica differenza è che la chiave di classe decodificata non viene rimossa dalla memoria quando il dispositivo è bloccato.

Nessuna protezione
(NSFileProtectionNone): questa chiave di classe è protetta solo con l'UID ed è conservata in Effaceable Storage. Dato che tutte le chiavi necessarie per decodificare i file in questa classe sono archiviate sul dispositivo, la codifica può solo trarre vantaggio dalla cancellazione rapida a distanza.

Classe A	Protezione completa	(NSFileProtectionComplete)
Classe B	Protetto se non è aperto	(NSFileProtectionCompleteUnlessOpen)
Classe C	Protetto fino alla prima Autenticazione Utente	(NSFileProtectionCompleteUntilFirstUserAuthentication)
Classe D	Nessuna protezione	(NSFileProtectionNone)

Figura 2.3 Chiave di classe per la protezione dei dati

2.2.3 Protezione dati del portachiavi

Molte app devono gestire password e altri dati non di grandi dimensioni ma sensibili, quali ad esempio chiavi e token di login. iOS fornisce un portachiavi sicuro in cui conservare questi elementi.
Il portachiavi è implementato come un database SQLite archiviato nel file system.

Esiste solo un database; il daemon security determina quali sono gli elementi del portachiavi a cui possono avere accesso i processi o le app. Le API di accesso al portachiavi eseguono chiamate al daemon, che a sua volta esegue la richiesta alle autorizzazioni "keychain-access-groups" "application-identifie" e "application-group" per l'app. I gruppi di accesso, piuttosto che limitare l'accesso a un singolo processo, consentono la condivisione tra app degli elementi del portachiavi.
Gli elementi del portachiavi possono essere condivisi solo tra app dello stesso sviluppatore. Perché sia possibile, viene richiesto alle app di terze parti di utilizzare gruppi di accesso con un prefisso assegnato a esse attraverso il programma per sviluppatori di iOS tramite i gruppi di applicazioni. La richiesta di prefisso e l'unicità di appartenenza a un gruppo di applicazioni sono applicate attraverso la firma del codice, i profili di fornitura e il programma per sviluppatori.
I dati del portachiavi sono protetti utilizzando una struttura di classe simile a quella usata nella protezione dati dei file. Queste classi hanno comportamenti equivalenti alle classi di protezione dati dei file, ma usano chiavi distinte e fanno parte di API che hanno un nome diverso.

Disponibilità	Protezione dati dei file	Protezione dati del portachiavi
Quando sbloccato	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
Quando bloccato	NSFileProtectionCompleteUnlessOpen	N/A
Dopo primo sblocco	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Sempre	NSFileProtectionNone	kSecAttrAccessibleAlways
Codice abilitato	N/A	kSecAttrAccessible WhenPasscodeSetThisDeviceOnly

Figura 2.4 Chiavi di classe per la protezione dati del portachiavi

2.3 Sicurezza delle app

Le app sono senza dubbio degli elementi cruciali all'interno di un'architettura mobile moderna e sicura. Se da un lato le app apportano agli utenti incredibili benefici dal punto di vista della produttività, dall'altro

rappresentano un rischio potenziale per la sicurezza del sistema, la stabilità e i dati dell'utente se non sono gestite correttamente. iOS fornisce vari livelli di protezione per garantire che le app siano firmate e verificate, e che siano "sandboxed" (ossia non possono accedere ai dati archiviati da altre applicazioni) per proteggere i dati dell'utente.

2.3.1 Firma del codice delle app

Il kernel di iOS, una volta avviato, controlla i processi utente e le app che possono venire eseguite. Per assicurarsi che tutte le app provengano da una fonte conosciuta e approvata, e che non siano state danneggiate, iOS richiede che tutto il codice eseguibile venga firmato utilizzando un certificato emesso da Apple. Le app fornite con il dispositivo, come Mail e Safari, sono firmate da Apple. La firma del codice obbligatoria estende il concetto di catena di fiducia e lo porta dal sistema operativo alle app, e impedisce alle app di terze parti di caricare risorse codice non firmate o di usare codice auto modificante.

Per poter sviluppare e installare app sui dispositivi iOS, gli sviluppatori devono registrarsi presso Apple e prendere parte al programma per sviluppatori di Apple. Prima di emettere il certificato, Apple verifica l'identità reale di ogni sviluppatore, sia esso un individuo o un'azienda. Questo certificato abilita gli sviluppatori a firmare le app e a inviarle a App Store per la distribuzione.

iOS permette agli sviluppatori di incorporare nelle proprie app dei framework che possono essere utilizzati dall'app stessa o da estensioni incorporate all'interno dell'app. Per proteggere il sistema e altre app ed evitare che carichino codice di terze parti all'interno del loro spazio di indirizzi, il sistema eseguirà una convalida della firma del codice di tutte le librerie dinamiche a cui un processo si collega all'avvio. Questa verifica si compie attraverso l'identificatore di team (Team ID), che viene estratto da un certificato emesso da Apple. Un identificatore di team è una stringa alfanumerica di lunghezza pari a 10 caratteri, ad esempio 1A2B3C4D5F.

A differenza di altre piattaforme mobili, iOS non consente agli utenti di installare app non firmate potenzialmente nocive scaricate da siti web, e nemmeno di eseguire codice non attendibile. Durante l'esecuzione, la firma del codice di tutte le pagine di memoria eseguibili viene verificata durante il caricamento della pagina, per garantire che un'app non sia stata modificata dall'ultima volta che è stata installata o aggiornata.

2.3.2 Sicurezza del processo di runtime

Una volta verificato che l'app proviene da una fonte approvata, iOS mette in pratica le misure di sicurezza progettate evitare che altre app o il resto del sistema vengano compromessi.
Tutte le app di terze parti sono "sandboxed", ovvero non possono accedere ai file archiviati da altre applicazioni o apportare delle modifiche al dispositivo. Questo meccanismo fa sì che le app non possano raccogliere o modificare le informazioni archiviate da altre app. Ogni app dispone di una cartella Inizio unica per i propri file, che viene assegnata in maniera casuale al momento dell'installazione dell'app. Se un'app di terze parti deve accedere a informazioni diverse dalle proprie, lo può fare unicamente usando i servizi forniti specificamente da iOS.

2.3.3 Estensioni

iOS consente alle app di fornire funzionalità ad altre app attraverso le estensioni. Le estensioni sono binari eseguibili firmati con uno scopo specifico, inseriti in un pacchetto all'interno di un'app. Il sistema le rileva automaticamente al momento dell'installazione e le rende disponibili per le altre app che utilizzano un sistema corrispondente. L'area di sistema che supporta le estensioni è chiamata punto di estensione. Ogni punto di estensione fornisce delle API e applica delle politiche per quell'area specifica. Il sistema determina le estensioni disponibili basandosi su regole

di corrispondenza specifiche per ciascun punto di estensione e avvia automaticamente i processi delle estensione quando necessario e ne gestisce la durata. Per limitare la disponibilità delle estensioni ad applicazioni di sistema specifiche, possono venire utilizzate le autorizzazioni. Ad esempio, un widget per la schermata Oggi compare solo in Centro Notifiche e un'estensione di condivisione è disponibile solo dal pannello Condivisione.
Le estensioni vengono eseguite nel proprio spazio di indirizzo. La comunicazione tra l'estensione e l'app che l'ha attivata utilizza comunicazioni inter-process (IPC) mediate dal framework di sistema. Non hanno accesso ai rispettivi file o spazi di memoria. Le estensioni sono progettate per essere isolate l'una dall'altra, oltre che dalle app che le contengono e da quelle che le utilizzano.

2.3.4 Protezione dati nelle app

Il kit SDK (Software Development Kit) di iOS offre una suite completa di API grazie alla quale gli sviluppatori di terze parti e in-house possono adottare con estrema facilità la protezione dati e che garantisce il massimo livello di protezione nelle app. La protezione dati è disponibile per API di file e di database, inclusi NSFileManager, CoreData, NSData e SQLite.
Tutti i dati provenienti dalle varie app sono archiviati e codificati con chiavi protette tramite il codice dell'utente sul dispositivo. Ad esempio i Contatti, Note, Messaggi e Foto implementano la classe "Protetto fino alla prima autenticazione Utente".
Le app installate dall'utente che non optano per una classe specifica di protezione dati ricevono automaticamente la classe "Protetto fino alla prima Autenticazione Utente".

2.4 iOS-Deployment

Il significato più comune del termine “deployment” in informatica è la consegna o rilascio al cliente, con relativa installazione di una applicazione o un sistema software. Lo si può di fatto considerare come una fase del ciclo di vita del software, la quale conclude lo sviluppo e il relativo testing e da inizio alla manutenzione.

2.4.1 Procedura iOS-Deploy

Come già accennato in precedenza, MicroApp iOS è un’applicazione che consente di creare delle app complesse che soddisfino le varie necessità degli utenti.
Il problema che abbiamo riscontrato durante lo sviluppo è stata la procedura di Deploy, perché il sistema iOS non permette l’installazione diretta di app sul dispositivo, senza passare prima per l’App Store o per l’ambiente di sviluppo Xcode.
Per ovviare a questo dilemma molti utilizzano la procedura del jailbreak che rimuove le restrizioni software imposte da Apple nei dispositivi iOS, permettendo di installare software e pacchetti di terze parti non firmati e autorizzati da Apple.
Anche se questa procedura consiste solo nello sblocco del File System e dell’installazione di uno store alternativo, rimane comunque una procedura illegale.
L’alternativa è stata quella di installare le app tramite linea di comando, fornendoci di un macOS, evitando poi di utilizzare direttamente Xcode.

- Il primo passo da compiere è quello di installare:
  - **HomeBrew,** un gestore di pacchetti mancanti per macOS;

- **Node.js,** una piattaforma event-driven per il motore JavaScript V8, che contiene delle librerie open source che possono essere utilizzate da tutti gli sviluppatori;
- **ios-deploy**, un programma che tramite l'utilizzo delle librerie di node.js, riesce ad installare e fare il debug di un'applicazione.

- Il secondo passo da compiere è quello di esportare il file .IPA dal nostro progetto MicroApp.xcodeproj ed estrarre il file .APP che verrà poi installato sul dispositivo iOS.
  Questa procedura viene effettuata tramite il comando xcodebuild, che consente di creare e configurare progetti e schemi di Xcode.

1) *xcodebuild clean -project MicroApp.xcodeproj -configuration ReleaseAdhoc*
-alltargets

Questo comando ci consente di pulire il progetto da eventuali file intermedi che potrebbero causare qualche problema inaspettato.

2) xcodebuild archive -project MicroApp.xcodeproj -scheme "MicroApp" -archivePath MicroApp.xcarchive

Il seguente comando distribuisce un archivio di nome MicroApp.xcarchive che contiene tutti i file riguardanti la nostra MicroApp, da questo archivio ricaveremo il file MicroApp.ipa.

3) xcodebuild -exportArchive -archivePath MicroApp.xcarchive -exportPath
MicroApp.ipa -exportOptionsPlist exportPlist.plist

Questo comando estrae da MicroApp.xcarchive il file MicroApp.ipa, durante l'estrazione è richiesto anche di esportare il file

exportOptionsPlist.plist (6) che contiene alcune informazioni di configurazione relative alla build come:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>teamID</key>
    <string>MYTEAMID123</string>
    <key>method</key>
    <string>development</string>
    <key>uploadSymbols</key>
    <true/>
</dict>
</plist>
```

Le informazioni riguardo il metodo di distribuzione, che può esssere impostato come: app Store, enterprise, ad hoc e development, l'identificatore del team di sviluppo (teamID), l'opzione per includere simboli nel file IPA creato, e un opzione booleana per includere il Bitcode.

Se il comando viene eseguito senza errori, esaminando il contenuto del file IPA troveremo le seguenti cartelle: Payload, SwiftSupport, Symbols e WatchkitSupport, dove all'interno della cartella Paylod sarà presente il file MicroApp.app, che verrà installato sul dispositivo iOS tramite il comando:

4) ios-deploy --debug --bundle MicroApp.app

Ottenendo così la nostra applicazione.

Capitolo 3 - Progettazione di MicroApp iOS

Il progetto MicroApp iOS si basa sull'idea di dare la possibilità all'utente di creare e modellare a proprio piacimento delle applicazioni ad hoc senza avere conoscenze informatiche; per fare ciò l'applicazione dovrebbe essere modellata visualmente attraverso componenti grafiche associate ad un particolare comportamento applicativo e ad una specifica interfaccia utente.
L'applicazione MicroApp è già esistente da diversi anni sui dispostivi Android, col passare degli anni l'applicazione è stata estesa apportando diverse migliorie e funzionalità come i servizi web e domotica, interfacce grafiche personalizzate, e l'installazione delle applicazioni personalizzate direttamente sul dispositivo (7). Queste sono solo alcune caratteristiche con la quale possiamo dimostrare le potenzialità e le possibilità di crescita di un'applicazione di questo tipo.
Il nostro obiettivo è quello di realizzare una nuova versione di MicroAppEngine per dispositivi iOS, cercando di rinnovare e migliorare tutte le caratteristiche già presenti nella versione Android.

3.1 Analisi dei requisiti

La prima fase è stata individuare i requisiti che dovranno essere soddisfatti dall'applicativo che si va a sviluppare.
Di seguito sono descritti i requisiti funzionali individuati in fase di analisi.

- **Editor visuale**
  Fornire un'interfaccia grafica che consente all'utente di selezionare ed inserire delle icone che collegandole tra di loro, rispettando delle regole di composizione, permettano la creazione dell'applicazione desiderata.

- **Il Motore di esecuzione**

  Implementare un Engine che in base alla modellazione dell'applicazione da parte dell'editor visuale, riesca a costruire la microApp garantendone tutte le specifiche espresse in fase di editing.

- **Scelta del file di deploy**

  Una volta che l'utente avrà composto la sua microApp, verrà creato un file di deploy, scritto in xml, che descriverà tutte le caratteristiche dell'app che si vuole andare a creare. L'Engine ha bisogno di conoscere tutte le specifiche dettate in fase di editing. Quindi l'editor provvederà ad analizzare le componenti, la loro disposizione e il flusso di dati impostato e li trascriverà in un file xml destinato all'Engine.

  Come formato abbiamo scelto di utilizzare l'xml per via della versatilità del linguaggio che lo rende molto facile da analizzare, e la sua semplicità di utilizzo.

- **Gestione flusso dei dati**

  Implementare un meccanismo di gestione del flusso dei dati in modo tale che ogni componente abbia sempre a disposizione tutti i dati di cui ha bisogno prima di essere eseguita (8). Per fare questo, l'Engine raccoglie gli output generati dall'ultima componente eseguita, dopodiché si occupa di inviarli alle componenti destinatarie, assicurandosi che siano le sole ad aver ricevuto quei dati.

- **Tipi di dato e molteplicità**

  I dati che andremo a gestire sono tutte le informazioni che possono essere elaborate all'interno di una microApp. Allo stato attuale l'applicazione dovrà poter gestire i seguenti tipi di dato:

  - **Stringa**: descrive una sequenza di caratteri;

- **Immagine**: descrive un'immagine acquisita da una fotocamera o da una galleria;
- **Contatto**: descrive le informazioni riguardante una persona;
- **Posizione**: descrive le coordinate geografiche di un luogo (latitudine, longitudine);
- **Object**: descrive un oggetto generico non facente parte dei precedenti tipi.

Tali dati devono essere gestiti tutti in maniera analoga.

I dati prodotti da una componente sono di tipo multiplo, ciascun dato viene rappresentato come un array, il dato multiplo può contenere un numero arbitrario di istanze dei tipi definiti, ma potrebbe contenere anche un solo elemento.

- **Funzionamento delle componenti**

  Le componenti sono rappresentate da una classe che contiene le caratteristiche principali e funzionalità. Per rendere più efficiente la loro gestione e semplificare l'aggiunta di nuove componenti, sono state individuate due entità distinte:

  - Una prima entità che si occupa di memorizzare i dati e tener traccia delle componenti da cui si ricevono gli input e quelle a cui si inviano gli output;
  - La seconda entità implementa la logica applicativa di una componente, come inviare un messaggio o salvare un contatto, così come l'elaborazione dei dati che varia in base allo scopo della componente.

  Quindi ogni componente sarà composta da entrambe queste entità. Questa suddivisione ci ha permesso di isolare la componente da diverse responsabilità.

- **Input & Output**
  Ogni componente dovrà occuparsi di memorizzare ed elaborare dati, per fare questo è necessario usare un metodo che aiuti a strutturare in modo adeguato la ricezione dei dati in input e l'invio dei dati in output.
  Visto che avremo input diversi, rappresentati da dati distinti, che possono essere anche dello stesso tipo, è stato stabilito che questi vengano suddivisi per nome, in questo modo si è evitato qualunque tipo di conflitto dovuto ai tipi di dati ricevuti.
  Per gli output, invece, è stato stabilito di inviare un unico pacchetto contenente tutti i dati, suddivisi per tipo, da spedire alle componenti. Quando la componente riceverà in input questo pacchetto sarà lei a specificare quale tipo di dato intende ricevere.

- **Estendibilità dell'applicazione**
  L'applicazione deve essere facilmente estendibile, poiché l'aggiunta di nuove componenti e nuovi tipi di dato deve essere quanto più naturale possibile per lo sviluppatore.

- **Navigabilità dell'applicazione**
  In qualsiasi punto dell'applicazione deve essere data la possibilità all'utente di tornare indietro alle componenti eseguite in precedenza e reimpostare i dati prodotti.

3.2 Sviluppo dell'ambiente di esecuzione

Il lavoro dell'Engine consiste nel costruire la microApp definitiva, raccogliendo le informazioni dal file di deploy per poi mandarla in esecuzione o installarla sul dispositivo iOS.
Le informazioni di cui ha bisogno l'Engine sono:

- L'elenco di tutte le componenti che compongono la microApp, comprendendo tutte le loro caratteristiche principali;
- Gli input e gli output di ciascuna componente.

Queste informazioni permettono all'Engine di determinare l'ordine sequenziale di esecuzione delle componenti e di coordinare l'invio e la ricezione dei dati tra le varie componenti.

3.2.1 Struttura di MicroAppEngine

Da una prima modellazione sono state individuate le seguenti entità:

- **MicroAppViewController:** si occupa di coordinare il flusso di esecuzione delle varie componenti, eseguendo di volta in volta la componente successiva o quella precedente, e raccogliendo gli output della componente terminata per poi inviarli alle rispettive componenti destinatarie.

- **Deploy Parser:** si occupa di analizzare il file di deploy, di ricavare l'insieme delle componenti e il flusso di dati che formano la microApp, deve decidere in che ordine le componenti devono essere eseguite, secondo un ordinamento topologico, e una volta ordinate, inviarle a MicroAppViewController.

- **Component:** questa classe descrive una componente generica, che sarà utilizzata dal sistema come contenitore di informazioni, tra cui:
  - Informazioni riguardo il proprio ID;
  - Gli input ricevuti dalle componenti precedenti;
  - Informazioni riguardo le componenti mittenti e destinatarie;
  - Un controller corrispondente al tipo della componente.

- **Component Controller:** un protocollo che descrive la logica applicativa di una componente, definendo:

- Le informazioni generiche come il tipo della componente;
- I dati che la componente vuole ricevere in input e quelli che vuole inviare in output.

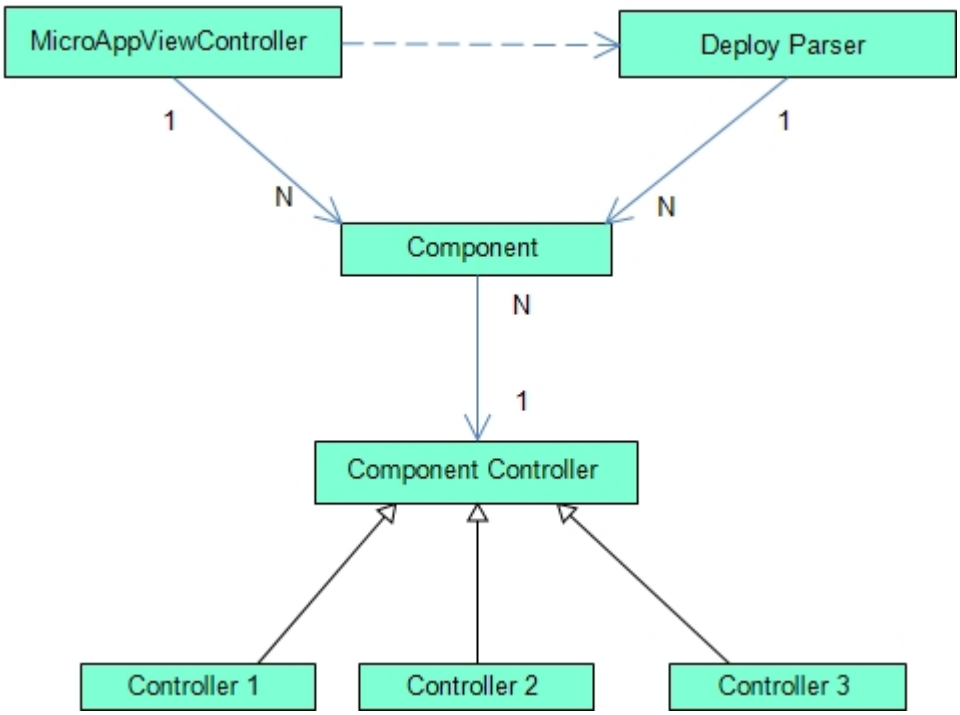

Figura 3.1 Diagramma delle entità principali di MicroAppEngine

- **Controllers:** sono le componenti all'interno dell'applicazione che permettono di fornire una data funzionalità, ricevendo eventualmente dati in input e producendo, eventualmente, dati in output.

Nella Figura 3.2 viene mostrato il Sequence Diagram realizzato secondo lo standard UML che mostra l'interazione che avviene tra utente e il sistema e le varie entità.

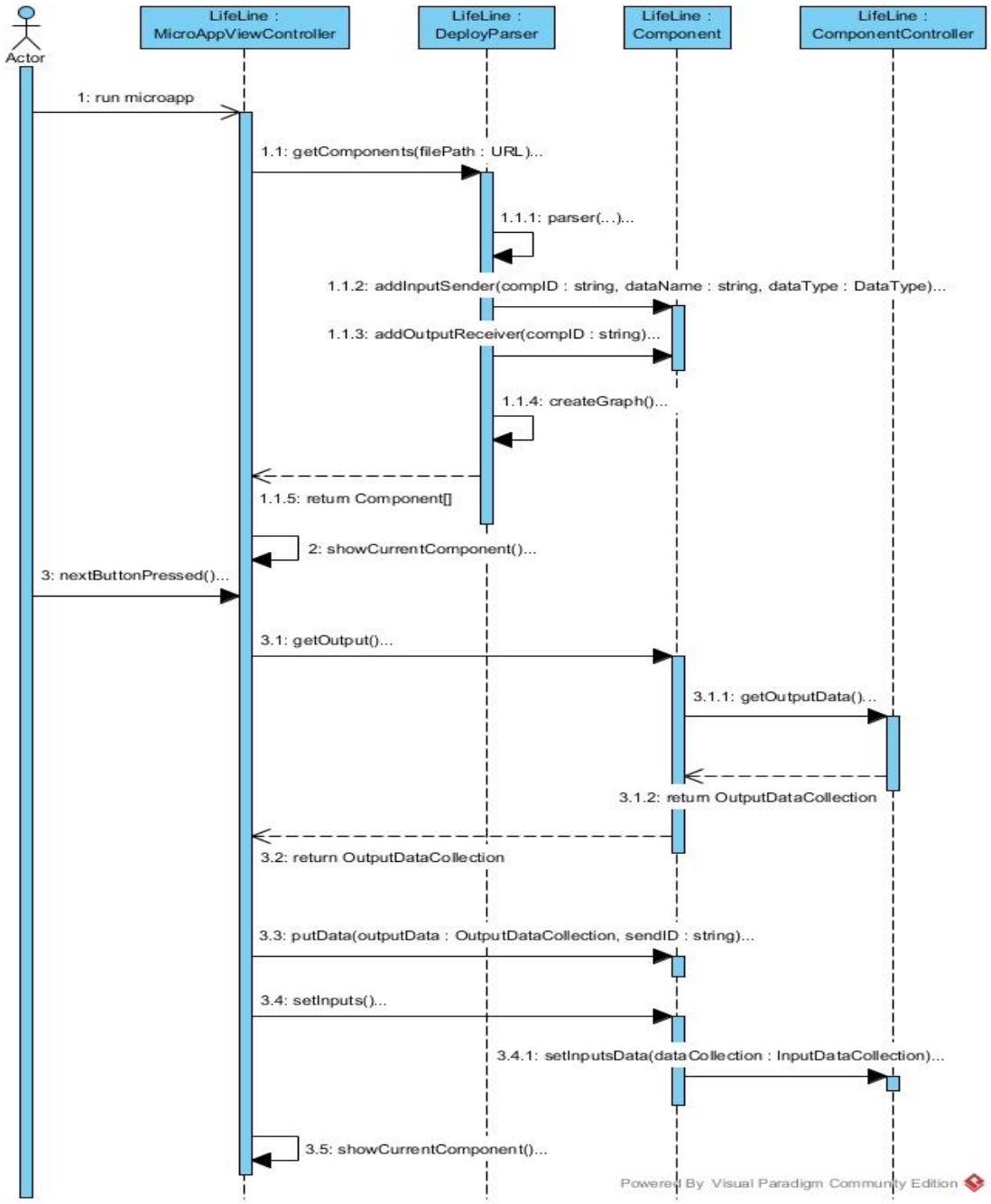

Figura 3.2 Sequence Diagram Riguardante l'esecuzione della microApp

3.3 Sviluppo dell'Editor visuale

In questa sezione descriviamo come l'editor dovrebbe essere sviluppato, come già accennato in precedenza, l'editor ha lo scopo di permettere all'utente di comporre graficamente una microApp avendo a disposizione varie componenti.

Due aspetti importanti sono:

- Offrire all'utente una visuale semplice e intuitiva;
- La necessità di avere regole e vincoli affinché venga garantita la coerenza e la robustezza della microApp.

3.3.1 Composizione grafica

L'editor è composto da una griglia contenente diverse celle vuote, queste celle possono essere occupate da una qualsiasi componente. Ogni componente è rappresentata da un quadrato colorato, contenente queste caratteristiche:

- Un'icona e un nome che illustrano il suo compito;
- Sul lato superiore del quadrato, saranno posti dei pin colorati che identificano i dati di input;
- Sul lato inferiore del quadrato, sarà presente un unico pin che identifica un pacchetto di dati di output.
- Oltre alle icone principali saranno presenti delle icone che identificano altre caratteristiche, come lo stato della componente o la presenza di dati statici.

Figura 3.3 Esempi delle componenti con relativi pin di input e di output

La composizione della microApp avverrà nel seguente modo:

- L'utente potrà scegliere tra le componenti disponibili quella di partenza che verrà posta al centro della griglia;

- Dopo aver posizionato la componente di partenza, l'utente potrà sceglie di posizionare altre componenti, collegandole tra di loro oppure inserendole in un punto isolato della griglia;
- Al termine della composizione, a seconda di come verranno poste le componenti, verrà determinato il flusso di esecuzione della microApp.

Il nuovo editor rispetto a quello Android ha l'obiettivo di favorire e rendere più agevole l'inserimento di nuove componenti, infatti, l'utente potrà estendere la microApp in maniera circolare, collocando le componenti intorno a quella di partenza, avendo così un maggiore controllo sul flusso di esecuzione desiderato.

3.3.2 Funzionalità dei pin

Come è già stato accennato l'editor ha bisogno di essere sottoposto a dei vincoli di composizione, riguardanti gli input e gli output delle componenti. Le componenti possono connettersi tra di loro attraverso dei pin, delle icone a forma di cerchio poste ai lati delle componenti che vanno ad indicare il tipo di dato che si vuole ricevere o inviare. In precedenza è stato detto che gli input e gli output sono strutturati diversamente, quindi bisogna mostrare questa differenza anche graficamente. Ogni componente potrà avere:

- Diversi pin di input, a seconda del dato che la componente avrà bisogno di ricevere;
- Un solo pin di output, che rappresenta il pacchetto di dati che la componente vorrà inviare.

Ci possono essere casi in cui la componente non preveda di ricevere dati o di inviarne, quindi non presenterà nessun pin nella sua rappresentazione grafica.

Per connettere due componenti tra di loro, all'interno dell'editor, è necessario che i pin di output trovino una corrispondenza con i pin di input e viceversa. Questo serve a verificare che una componente riceva esattamente l'oggetto di cui ha bisogno, nel caso in cui non si verifichi, le due componenti non possono connettersi tra di loro. Viceversa, se una componente deve inviare un oggetto, non è detto che questo venga inviato ad un'altra componente, e in questo caso il dato di output viene semplicemente perso, perché non viene utilizzato.

Riassumendo, quando una componente riceve in input un pacchetto di output da un'altra componente, questa individua il tipo di dato di cui ha bisogno e scarta i dati non necessari. Quindi l'editor dovrà accettarsi che quando si vuole far connettere due componenti, il pacchetto di output dovrà contenere il tipo di dato che la componente destinataria vorrà ricevere in input.
Elenchiamo ora i possibili casi di connessione tra due componenti:

- Ogni componente può ricevere diversi input da più componenti;
- Ogni componente può ricevere diversi tipi di input da una stessa componente o da più componenti;
- Ogni componente può ricevere più input dello stesso tipo da una stessa componente;
- Ogni componente può inviare il suo output a diverse componenti;
- Ogni componente può inviare il suo output più volte a una stessa componente;
- Ogni componente può decidere di non inviare il suo output.

In alcuni casi sarà richiesto all'utente di inserire esplicitamente i dati mancanti ad una o più componenti, nel caso in cui l'editor abbia trascurato degli input non assegnati. In questo modo si garantisce che nessuna componente rimanga sprovvista di input e viene garantita la correttezza della microApp che si andrà a creare.

3.4 Fasi di sviluppo

La progettazione e l'implementazione di MicroAppEngine si sono svolte seguendo le seguenti fasi di sviluppo:

- Realizzazione del procedimento di deploy di un'applicazione iOS;
- Individuazione dello schema principale delle entità;
- Sviluppo del processo di parsing del file di deploy;
- Sviluppo del grafo topologico per la linearizzazione delle componenti;
- Realizzazione del meccanismo di navigabilità ed esecuzione delle componenti;
- Definizione della struttura delle componenti con l'isolamento dei controller;
- Aggiornamento delle funzionalità per la gestione di dati tra l'Engine e le componenti;
- Implementazione dei controller principali.

Capitolo 4 - Isolamento delle Componenti

In questo capitolo descriveremo il meccanismo di isolamento delle componenti e il modello di estendibilità dell'applicazione, con le classi e le risorse messe a disposizione di futuri developer per l'aggiunta di nuove componenti. Infine parleremo delle varie componenti implementate.

4.1 Isolamento della Componente

Prima di poter parlare del concetto di isolamento è necessario accennare come era la prima applicazione di MicroApp per versione Android. Il motore di questa versione era basato su due classi fondamentali: MAComponent e MAActivity, la prima si occupava di descrivere una componente generica, la seconda definiva la logica di base delle varie Activity (schermate) presenti all'interno dell'applicazione (9) (10). Tramite queste due classi, l'Engine era in grado di realizzare tutte le componenti e fornire tutti i meccanismi previsti per il corretto funzionamento.
Analizzando a fondo l'applicazione abbiamo notato che la gestione dei dati veniva completamente gestita dall'Engine, infatti al suo interno c'era un repository, strutturato su due livelli, che utilizzava due mappe chiave valore annidate: un primo livello catalogava le classi delle componenti, un secondo livello in cui vi era una classificazione per tipo di dato.
Quando una componente doveva memorizzare i dati prodotti (dati in output) chiedeva all'Engine di farlo; allo stesso modo interrogavano l'Engine per verificare la presenza di dati ad essa destinati (dati in input).
Questo meccanismo ha portato l'Engine a sovraccaricarsi, così nella nuova progettazione abbiamo deciso di alleggerirlo facendo memorizzare i dati direttamente alle componenti, in modo tale che l'Engine debba solo preoccuparsi di far viaggiare i dati da una componente all'altra.

Nella versione iOS, per rendere più agevole la gestione dei dati e la gestione delle componenti, semplificando così anche l'aggiunta di nuove componenti, sono state sviluppate due entità:

- La classe Component che interagisce con l'Engine, si occupa di memorizzare nel suo DataCollection i dati provenienti dalle altre componenti, di spedire i suoi dati in output all'Engine e fornire informazioni riguardanti le componenti mittenti e destinatarie.

- Il protocollo ComponentController che definisce la logica applicativa di una specifica componente, ed elabora i dati delle varie componenti.

La classe Component ha un oggetto controller di tipo ComponentController, contenente tutti i dati generici riguardanti la componente specifica che si vuole andare ad implementare, quando l'Engine deve inviare gli input alla componente successiva, questi dati vengono raccolti dalla classe Component e a loro volta vengono memorizzati nel suo DataCollection usando il metodo putData(). Questi dati vengono passati dall'Engine alla classe Component e quando questa sta per essere eseguita, i dati vengono passati al controller. Invece, quando la componente deve spedire dei dati, gli output che vengono generati dalla componente, vengono passati alla classe Component tramite il controller, dopodiché la Component si occuperà di passare i dati all'Engine.

Vediamo un po' nel dettaglio come vengono gestiti i dati e le componenti:

- Ogni volta che una componente termina la sua esecuzione, MicroAppViewController invoca la funzione nextButtonPressed() richiamando a sua volta il metodo getOutput() della classe Component, quest'ultima gli passa i suoi output, e una volta ottenuti,

l'Engine li analizza e controlla a chi deve mandare i dati appena ricevuti e li spedisce alle componenti destinatarie. Il metodo getOutput() della classe Component richiama a sua volta, il metodo getOutputsData() del ComponentController, perché i dati effettivi vengono elaborati nel controller, infatti è lui che ha il compito di definire le funzioni per i vari compiti specifici delle componenti come fare una fotografia, inviare una mail o fare una chiamata etc.
I dati in output sono divisi per tipo di dato, così le componenti destinatarie possono identificare quali sono i dati di cui hanno bisogno.

- Quando l'Engine lancia una componente, deve mandarle gli input necessari per la sua esecuzione. Per fare questo, viene invocato il metodo setInputs() della classe Component che invia il suo DataColletion, contenente tutti i dati necessari, al controller, attraverso la chiamata al metodo setInputsData(). Infine il controller prende i dati e li salva nel suo DataCollection. I dati in input sono identificati con un nome, perché così la componente nel caso in cui ci siano più input dello stesso tipo riesce a distinguerli.

- Il Depoly Parser quando riceve il file .xml generato dall'editor, tramite il metodo parser() lo esamina, e quando trova il tag "component" costruisce una nuova componente con tutti i suoi attributi e i riferimenti alle componenti mittenti e destinatarie. Per conoscere le informazioni inerenti a come collegare le componenti tra di loro, il Parser interagisce con la Component che tramite i metodi addInputSender() e addOutputReceiver() ottiene gli ID e i nomi dei dati delle componenti mittenti e gli ID e i tipi di dato delle componenti destinatarie.
  Quando vengono create tutte le componenti necessarie per comporre la microApp, queste vengono ordinate secondo un ordinamento

topologico, creando un grafo diretto aciclico così definito: i nodi di un DAG si definiscono ordinati topologicamente se questi sono disposti in maniera tale che per ogni arco (A,B) che collega il nodo A al nodo B, nell'ordinamento il nodo A precede il nodo B. Infine il grafo viene inviato a MicroAppViewController che provvederà ad eseguire la microApp.

Come si può notare, separando la logica applicativa da quella di gestione dei dati, si ottiene l'isolamento della Component Controller nei confronti della Component.
Nella Component abbiamo racchiuso tutti i metodi che i Controller hanno in comune, in questo modo non è stato più necessario scriverli manualmente in ogni Controller perché questi metodi sono generici per tutti i Controller. Inoltre racchiudendo il Controller nella Component, abbiamo fatto in modo che l'Engine e il Parser interagiscano solo con la classe Component, senza preoccuparsi di cosa fanno nello specifico i Controller.
Di conseguenza l'aggiunta di una nuova componente consisterà semplicemente nell'implementare una nuova classe Controller con le funzionalità specifiche della stessa, tralasciando tutti i meccanismi di gestione dei dati che vengono forniti dal sistema. Inoltre, avendo una suddivisione del genere, che separa le funzionalità dell'intera struttura, otteniamo una riduzione delle dipendenze tra le componenti del software e quindi una buona leggibilità del codice e una maggiore efficienza e facilità di manutenzione.

4.2 Struttura architetturale

Per quanto riguarda il meccanismo di isolamento della componente, è stata utilizzata come architettura, lo strategy pattern (vedere Fig. 4.1), uno dei pattern fondamentali, definiti dalla gang of four. Il suo obiettivo è quello di

isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare utile in quelle situazioni dove sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione.

Questo pattern prevede che gli algoritmi siano intercambiabili tra loro, in base ad una specifica condizione, in modalità trasparente al client che ne fa uso (11). In altre parole, data una famiglia di algoritmi che implementa una certa funzionalità, essi dovranno esportare sempre la medesima interfaccia, così il client dell'algoritmo non dovrà fare nessuna assunzione su quale sia la strategia istanziata in un particolare istante.

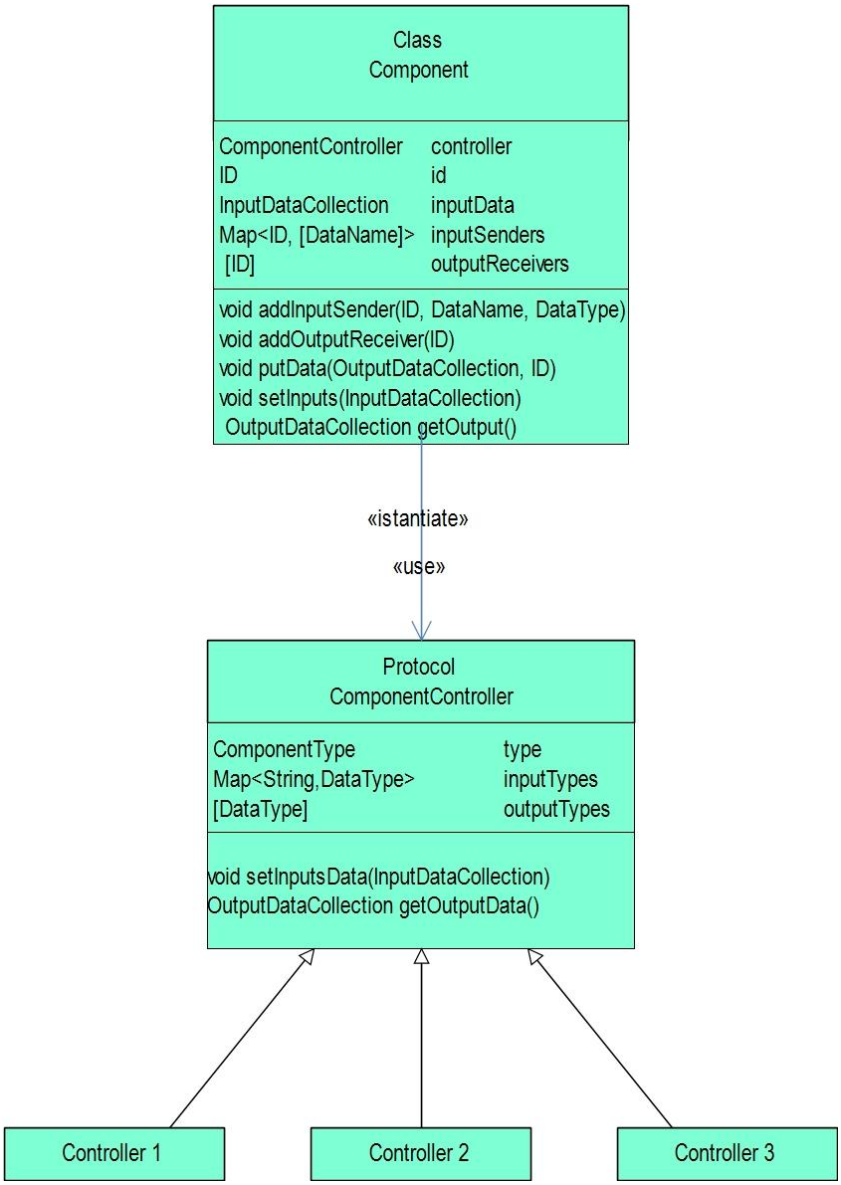

Figura 4.1 Strategy Pattern - Struttura Isolamento della Componente

4.2.1 Partecipanti e Implementazione

Questo pattern è composto dai seguenti partecipanti:

- Strategy: dichiara una interfaccia che verrà invocata dal Context in base all'algoritmo prescelto.

- ConcreteStrategy: sono le classi che implementano i vari algoritmi.

- Context: detiene le informazioni generiche dell'algoritmo da utilizzare e ha il compito di invocare l'algoritmo.

Definiamo la nostra classe di contesto in questo caso la classe Component, che rappresenta il legame con l'Engine e il Component Controller.

```
class Component: Equatable {

    let controller: ComponentController
    let id: String

    var inputData: InputDataCollection
    var inputSenders: [String: [String]]
    var outputReceivers: [String]

    init(id: String, type: ComponentType) {
        self.id = id
        let storyboard = UIStoryboard(name: "MicroApplication", bundle: nil)
        self.controller = storyboard.instantiateViewController(withIdentifier:
                          (type.rawValue)) as! ComponentController
        inputSenders = [String: [String]]()
        outputReceivers = [String]()
        inputData = InputDataCollection()
    }
```

Figura 4.2 Costruttore della classe Component

Al costruttore della Component vengono passati dal Deploy Parser, l'identificatore e il tipo della componente che si vuole andare a creare, al suo interno sono presenti tutte le variabili che la compongono come:

- Un oggetto storyboard che in base al tipo della componente si ricava il controller speficico.
- Un oggetto controller di tipo ComponentController contiene tutte le funzioni specifiche che deve implementare;
- La mappa inputSenders contenente gli id delle componenti mittenti e i nomi dei dati ricevuti in input da quelle componenti;
- L'array outputReceivers contenente gli id delle componenti destinatarie;
- La mappa inputData che viene utilizzata per memorizzare tutti i dati in entrata.

```
    func setInputs() {
        for (dataName, _) in controller.inputTypes {
            if !inputData.getNames().contains(dataName) {
                print("input mancanti")
                // return
            }
        }
        controller.setInputsData(inputData)
    }

    func getOutput() -> OutputDataCollection? {
        let dataCollection = controller.getOutputData()
        if dataCollection != nil {
            for dataType in controller.outputTypes {
                if !dataCollection!.getTypes().contains(dataType) {
                    print("output mancanti")
                    return nil
                }
            }
        }
        return dataCollection
    }

    static func ==(comp1: Component, comp2: Component) -> Bool {
        return comp1.id == comp2.id
    }

}
```

Figura 4.3 Metodi principali della classe Component

I metodi setInputs() e getOutput() implementati da Component, vengono invocati dalla classe MicroAppViewController e interagiscono con il protocollo Component Controller.

Ogni volta che una componente termina, l'Engine prende gli output di quella componente e li distribuisce a tutti i destinatari e quando arriva il momento di eseguire una nuova componente viene invocato il metodo setInputs() che tramite l'inputTypes del controller verifica se tra tutti i dati passati dalle componenti mittenti ad inputData ci sono quelli da mandare alla componente, in seguito si occuperà di prendere i dati e mandarli al controller, invocando a sua volta il metodo setInputsData(inputData) del protocollo Component Controller.

Invece il metodo getOutput(), al termine di ogni componente, richiama il getOutputData() del Component Controller e verifica tramite outputTypes se ci sono tutti i dati in output che la Component deve inviare, li memorizza temporaneamente nel dataCollection, e in seguito li invia a MicroAppViewController che li elabora e li spedisce alle componenti destinatarie.

```
import Foundation
import UIKit

protocol ComponentController {

    var type: ComponentType { get set }
    var inputTypes: [String: DataType] { get }
    var outputTypes: [DataType] { get }

    func setInputsData(_ dataCollection: InputDataCollection) -> Void
    func getOutputData() -> OutputDataCollection?

}
```

Figura 4.4 Proprietà del protocollo Component Controller

Nel protocollo Component Controller che rappresenta l'interfaccia che collega i vari Controller alla classe Component, sono dichiarate tutte le variabili e i metodi che i controller andranno ad implementare:

- type è una proprietà del protocollo che può assumere uno dei valori dell'enumeratore ComponentType, quest'ultimo, contiene tutti i tipi delle componenti;

- inputTypes è una mappa che contiene i nomi e i tipi di dati in input;
- outputTypes è un array che contiene i tipi di dati in output;
- setInputsData() è la funzione che viene implementata dai controller per prendere i dati in ingresso dall'InputDataCollection se il controller ne necessita. L'InputDatacollection è la tabella contenente tutti i dati in input, questa viene passata al controller tramite inputData della classe Component.
- getOutputData() crea temporaneamente un'OutputDataCollection dove vengono aggiunti i dati appena elaborati dal controller e vengono inviati alla Component.

Di seguito andremo a descrivere tutti i controller che implementeranno le proprietà del Component Controller, ognuno di esso li eseguirà in maniera diversa, a seconda della funzione che dovrà impiegare.

4.2.2 Componenti Implementate

In questa sezione sono descritti tutti i Controller implementati durante la realizzazione del progetto. Un Controller implementa la logica della componente in MicroApp iOS, al suo interno, infatti, sono presenti tutte le funzioni che ogni singola componente deve compiere come ad esempio fare una foto, inviare una mail, salvare un oggetto etc.

Tutti i Controller che andremo a descrivere implementano il protocollo ComponentController, di conseguenza, implementeranno tutte le proprietà e metodi necessari a soddisfare un compito specifico.

TakePhotoViewController

Questo controller permette di catturare una nuova immagine mediante la fotocamera del dispositivo. Quando viene eseguito mostra un'interfaccia di base molto simile all'applicazione fotocamera presente all'interno di iOS. Questo controller ha due funzioni specifiche: TakePhoto() e

ImagePickerController(); La prima permette di accedere alla fotocamera del dispositivo e scattare una foto. L'altra, invece, se la fotocamera del dispositivo non può essere utilizzata, si accede alla galleria immagini, si seleziona un'immagine e questa viene memorizzata e inviata in output alla prossima componente. Per far sì che questi metodi funzionino a dovere è stato utilizzato il controllore UIImagePickerController che contiene un'interfaccia preconfezionata e la logica necessaria per recuperare e selezionare immagini da tre tipi di sorgente: la fotocamera, gli album e la cartella delle immagini.
Il controller non presenta parametri in input, mentre in output restituisce un oggetto di tipo IMAGE.

PreviewImageViewController

E' molto semplice la logica che c'è dietro questo controller. Il metodo setInputsData() ricava dall'InputDataCollection, l'immagine che gli è stata inviata dalla componente precedente, questa immagine poi viene mostrata sullo schermo tramite la funzione showImage(). In fase di output viene semplicemente replicato il dato letto in input.

SelectContactViewController

Questa componente permette di selezionare uno o più contatti presenti nella rubrica mediante l'uso del framework ContactUI che fornisce ai suoi controllori le operazioni di visualizzazione, di modifica, di selezione e la creazione di nuovi contatti.
SelectContactViewController utilizza la funzione selectContact() per selezionare i contatti dalla rubrica, mentre contactPicker() tramite il controllore CNContactPickerViewController recupera tutti i contatti della rubrica memorizzati sul dispositivo. Infine, in fase di output restituisce un oggetto di tipo CONTACT.

CallContactViewController

Questa componente non prevede un'interfaccia, il suo unico scopo è quello di chiamare il contatto selezionato in precedenza. Quindi all'interno del metodo setInput() viene recuperato il contatto precedentemente salvato da un'altra componente, viene estratto il numero di questo contatto e mediante la funzione CallContact() viene eseguita la chiamata. Se precedentemente sono stati selezionati più contatti, alla fine di ogni chiamata verrà eseguita quella successiva. Infine non vi sono particolari operazioni da eseguire in fase di output.

SendMessageViewController

Invia un messaggio a uno o più contatti letti in input attraverso setInputsData(). Viene utilizzato il framework MessageUI che permette di creare un'interfaccia utente per la composizione di messaggi di posta elettronica e di testo. Le funzioni principali sono:

- messageComposeViewController() che si occupa di creare l'interfaccia utilizzando il controllore MFMessageComposeViewController che fornisce il necessario per l'invio di messaggi SMS o MMS.
- sendMessage() si occupa di prendere i numeri dei vari contatti passati dalla componente precedente e una volta scritto il messaggio, inviarlo a tutti i destinatari.

Non sono previsti dati in output da inviare ad altre componenti.

SendMailViewController

Questa componente è simile a quella descritta precedentemente, nel metodo setInputsData() oltre ai contatti vengono recuperati gli oggetti di tipo AnyObject che possono essere delle foto, file di testo, file musicali etc.,

e gli oggetti locations che servono per conoscere la località da cui si sta spedendo il messaggio.
Anche SendMailVieController utilizza il framework MessageUI, infatti nel metodo messageComposeViewController(), viene utilizzato MFMailComposeViewController che permette di creare un'interfaccia standard per la gestione, la modifica e l'invio di un messaggio di posta elettronica.
Il metodo SendMail() prende la lista di contatti e ne ricava gli indirizzi email, in seguito, vengono aggiunti gli allegati e la posizione del mittente. Infine quando la mail è pronta, viene spedita ai vari destinatari. Dopo l'invio non sono previste scritture di dati per l'output.

LocationViewController

Questa logica implementa il meccanismo mediante il quale la nostra applicazione riesce a interfacciarsi col GPS. Viene utilizzato il framework CoreLocation che ci permette di ottenere la posizione geografica e l'orientamento del dispositivo. I metodi utilizzati per il funzionamento del controller sono:

- getLocation() che tramite l'uso di CLLocationManager avvia e arresta la consegna degli eventi correlati alla posizione e tramite CLLocation ottiene le informazioni di latitudine, longitudine e altitudine, infine li memorizza e li invia al metodo showLocation().
- showLocation() mostra a video le coordinate di latitudine, longitudine e altitudine.
  L'operazione di output è di salvare l'ultima Location aggiornata in modo che sarà disponibile alle componenti che né devono fare uso.

MapViewController

Non fa altro che recuperare i dati della Location e visualizzarne la posizione geografica. Utilizza il framework MapKit che permette di visualizzare una mappa o le immagini satellitari direttamente dall'interfaccia dell'applicazione.

Non sono previsti dati in output da inviare ad altre componenti.

SaveViewController

Implementa il salvataggio di un qualsiasi dato all'interno dell'applicazione, recupera i dati di output delle componenti precedenti attraverso il metodo setInputData() e li passa al metodo saveData() che analizza i dati, controlla di che tipo sono, e a seconda del tipo li salva all'interno del dispositivo. Ad esempio, se un oggetto è di tipo IMAGE, questo viene rinominato in formato .JPG e salvato nella memoria del dispositivo.

Il controller non prevede dati in output da passare alle altre componenti.

4.3 Come aggiungere nuove componenti

Come descritto in precedenza, è possibile aggiungere nuove componenti. Questi sono i passi da eseguire quando si vuole aggiungere una nuova componente.

1. Creare una nuova classe Controller
2. Aggiungere all'enumeratore ComponentType il nuovo tipo della componente che si vuole andare a creare.
3. Aggiungere, eventualmente, all'enumeratore DataType il nuovo tipo di dato che deve elaborare il Controller.
4. Prendere i vari metodi e proprietà dal protocollo Component Controller ed implementarli nel nuovo Controller:

- setInputsData(), metodo utilizzato per preparare eventuali dati in input che possono servire al Controller;
- getOutputData(), metodo utilizzato per eventuali dati in output che il Controller ha elaborato per passarle poi alle componenti destinatarie;
- La variabile type usata per aggiungere il nuovo tipo della componente;
- La mappa inputTypes che contiene i nomi e i tipi di dati in ingresso;
- L'array outputTypes che contiene i tipi di dati in uscita.

5. Implementare i metodi che caratterizzano il Controller
6. Creare una nuova schermata ViewController nello storyboard dell'applicazione, per creare una connessione tra l'interfaccia utente e il codice implementato dal Controller.
7. Collegare la View alla classe Controller e inserire nello storyboard ID il tipo della componente, perché così il metodo storyboard.instantiateviewcontroller della classe Component trova la View corrispondente al controller.
8. Infine, inserire nel metodo viewDidLoad() del ViewController, i metodi del Controller.

4.3.1 Esempio di aggiunta di una nuova componente

In questo paragrafo sarà illustrata l'aggiunta di una componente generica che prende una stringa in input, viene visualizzata sullo schermo, ed infine inviata alla componente successiva.

Aggiungere una nuova componente è molto semplice, si crea una nuova classe Controller, in questo caso la chiameremo GenericViewController,

che andrà ad implementare i metodi del protocollo ComponentController e verranno aggiunte le funzionalità caratterizzanti della classe.

```
import UIKit

class GenericViewController: UIViewController, ComponentController {

    override func viewDidLoad() {
        super.viewDidLoad()

        dataLabel.text = string as String
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()

    }

    func setInputsData(_ dataCollection: InputDataCollection) {
        string = dataCollection.getData(named: "string")?[0] as! NSString
        print("La stringa ricevuta è", string)
    }

    func getOutputData() -> OutputDataCollection? {
        var output = OutputDataCollection()
        output.addData(string, ofType: DataType.STRING)
        return output
    }
```

Figura 4.5 Creazione della classe GenericViewController

In questo caso, setInputsData() prende come parametro una stringa e viene passata al metodo viewDidLoad(). Questo metodo provvederà a mandare la stringa alla ViewController della componente. Invece getOutputData() al termine dell'esecuzione della componente provvederà ad inviare la stringa visualizzata alla prossima componente.

Una cosa molto importante da fare è quella di aggiungere un nuovo valore all'enumeratore ComponentType, perché contiene una lista di tutte le componenti che MicroApp deve riconoscere. Inoltre, se necessario, va aggiunto un nuovo valore anche all'enumeratore DataType che contiene la lista di tutti i tipi di dati che viaggeranno tra le varie componenti.

```
import Foundation

enum ComponentType: String {

    case TAKEPHOTO
    case PREVIEWIMAGE
    case CALLCONTACT
    case CALLINTERCEPTOR
    case SELECTCONTACT
    case PREVIEWCONTACT
    case SENDMESSAGE
    case LOCATION
    case MAP
    case SAVE
    case SENDMAIL

    case GENERIC
}
```

```
import Foundation

enum DataType: String {

    case CONTACT
    case STRING
    case LOCATION
    case IMAGE
    case URI
    case OBJECT

}
```

Figura 4.6 Enumeratori di ComponentType e DataType

Una volta che è stato deciso il compito che deve eseguire il nostro Controller, bisogna implementare le variabili del Component Controller. In questo caso, la variabile type assumerà il valore GENERIC, aggiunto precedentemente all'enumeratore. In inputTypes vengono aggiunti come chiavi i nomi e come valore i tipi dei dati in input che ci interessa ricevere, in questo caso delle stringhe. Invece, in outputTypes vengono aggiunti i tipi di dati che si vogliono inviare, nel nostro caso consideriamo semplicemente delle stringhe. Infine vengono aggiunte le altre variabili che possono servire alla classe.

```
var type = ComponentType.GENERIC
var inputTypes = ["string": DataType.STRING]
var outputTypes = [DataType.STRING]
var string = NSString()
@IBOutlet var dataLabel: UILabel!
```

Figura 4.7 Variabili utilizzate dal Controller

Dopo aver creato la classe Controller, bisogna creare una ViewController (Fig. 4.8) che rappresenterà l'interfaccia del GenericViewController.
Nella View vanno inseriti tutti gli elementi per la visualizzazione ed interazione con i dati posseduti dal Controller come: buttons, label, textview, textfield, etc.

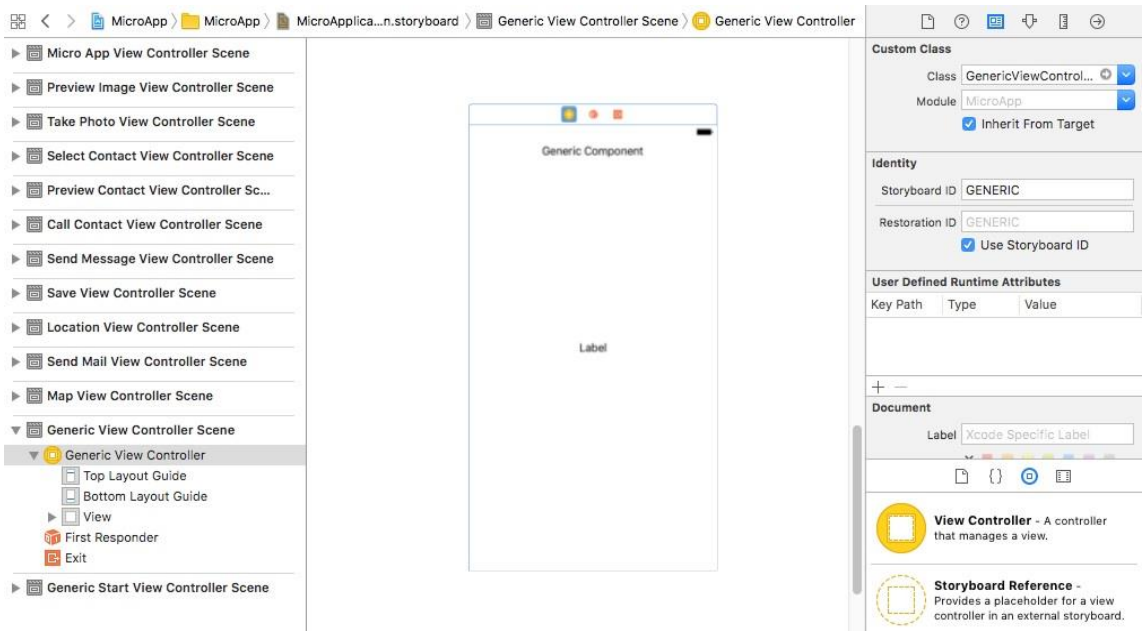

Figura 4.8 Creazione della View

Dopo aver composto la view, bisogna collegarla al Controller, e assegnare il tipo della componente allo Storyboard ID.

Una volta completati tutti i passaggi, basterà semplicemente salvare e compilare il progetto, installare l'applicazione sul dispositivo o sull'emulatore di Xcode e provare a creare una nuova microApp con la nuova componente creata (Fig. 4.9).

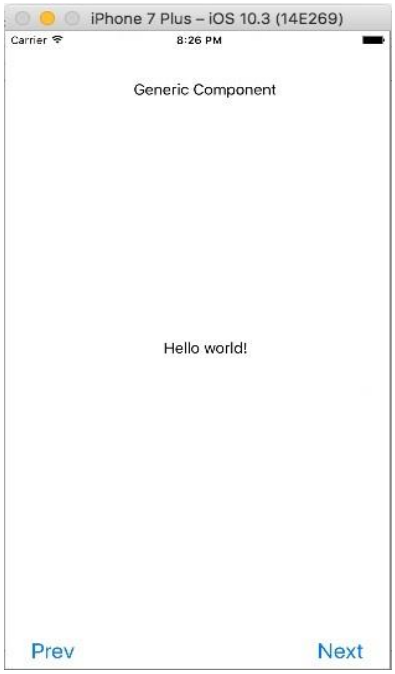

Figura 4.9 Esecuzione della GenericViewController

Capitolo 5 - Esempi di utilizzo

In questo capitolo è riportato un caso di utilizzo della nostra applicazione. Attualmente l'Editor non ancora è stato sviluppato, quindi verranno mostrati dei mockups creati tramite il programma Balsamiq Mockups 3, che mostreranno indicativamente l'andamento della microApp.

5.1 La microApp "TakeSend&Notify"

Di seguito è illustrato un esempio di microApp che coinvolge buona parte delle componenti create. All'avvio dell'applicazione è presentata una schermata iniziale da cui selezionare l'operazione da svolgere.

Figura 5.1 schermata di MicroApp iOS

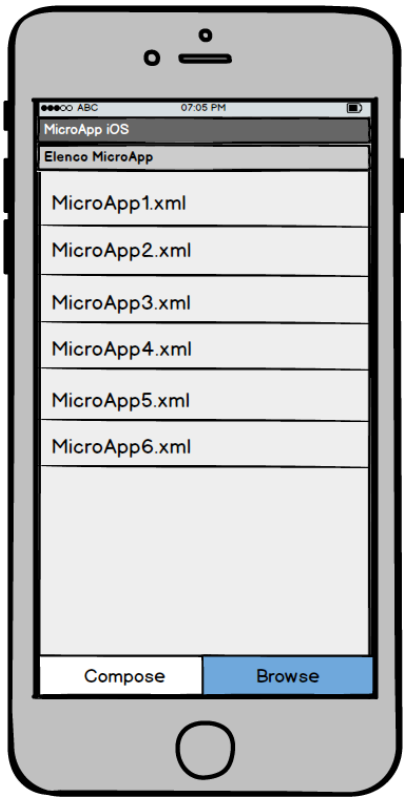

Figura 5.2 schermata di Browse

Mediante il pulsante "Compose", illustrato in Fig. 5.1, l'utente accede all'area di Editor dove potrà comporre la propria microApp, mentre con il

pulsante “Browse” (vedere Fig. 5.2) l’utente accede alla lista delle microApp che in precedenza aveva creato. Una volta cliccato esce una schermata in cui saranno presenti la lista di tutti i file XML presenti in cartella, selezionando uno dei file presenti sarà possibile visualizzare ed eseguire le operazioni su di esso.

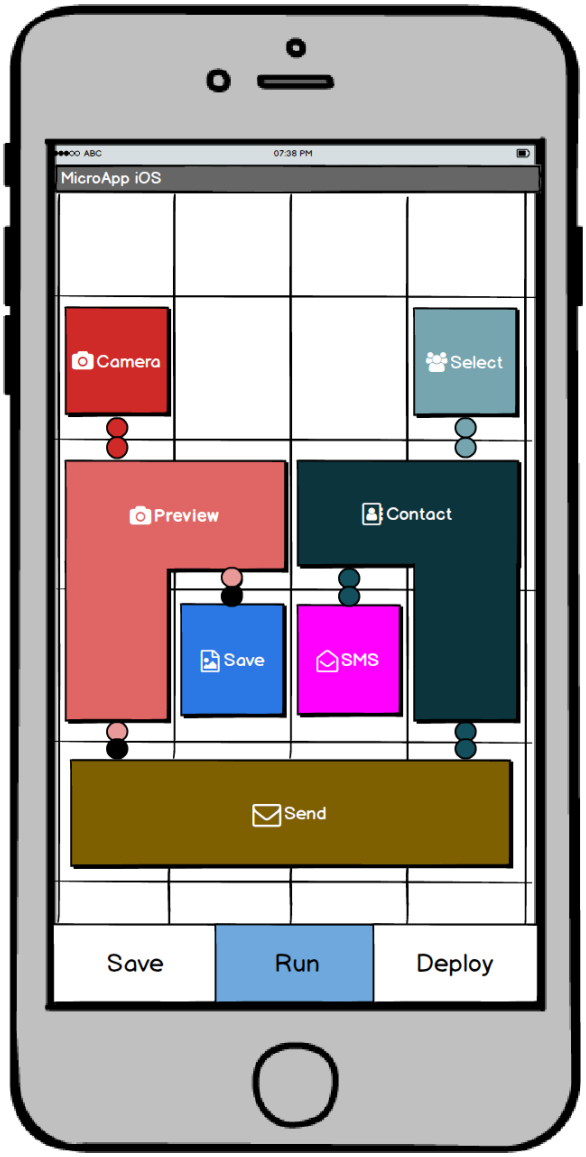

Figura 5.3 Editing di TakeSend&Notify

La microApp di esempio illustrata in Fig. 5.3 prevede che venga acquisita una fotografia, dopodiché viene visualizzata una preview dell’immagine precedentemente scattata, e successivamente l’immagine sarà salvata e

inviata acquisendo gli indirizzi email dei diversi contatti. Infine insieme all'email viene spedito un messaggio per i contatti che sono sprovvisti di un indirizzo email.
L'esecuzione della microApp avviene alla pressione del tasto "Run", con il tasto "Save" si avvia il procedimento per il salvataggio del file XML, mentre con Deploy viene installata la microApp sul dispositivo.

A fine composizione l'Editor provvede a generare il file di deploy (Fig. 5.4) della microApp da passare all'Engine che analizzerà il file e provvederà alla creazione della struttura.

```
<xml version="1.0" encoding="UTF-8"?>

        <component id="1" type="TAKEPHOTO">
            <output id="2">
        < /component>

        <component id="2" type="PREVIEWIMAGE">
            <input id="1"  dataname="image">
            <output id="3">
            <output id="6">
        < /component>

        <component id="3" type="SAVE">
            <input id="2" dataname="image">
        < /component>

        <component id="4" type="SELECTCONTACT">
            <output id="5">
        < /component>

        <component id="5" type="PREVIEWCONTACT">
            <input  id="1" dataname="contact">
            <output id="6">
            <output id="7">
        < /component>

        <component id="6" type="SENDMAIL">
            <input id="2" dataname="image">
            <input id="5" dataname="contact">
        < /component>

        <component id="7" type="SENDMESSAGE">
            <input id="5" dataname="contact">
        </component>

        < /xml>
```

Figura 5.4 File di deploy della microApp di esempio

Il grafo della microApp di esempio che si ottiene nel metodo è quello di seguito illustrato in Fig. 5.5. Gli elementi importanti da prendere in considerazione sono Contact e Send. La prima perché invia la stessa lista di contatti a due componenti distinte, la seconda perché necessita di più input per poter partire.

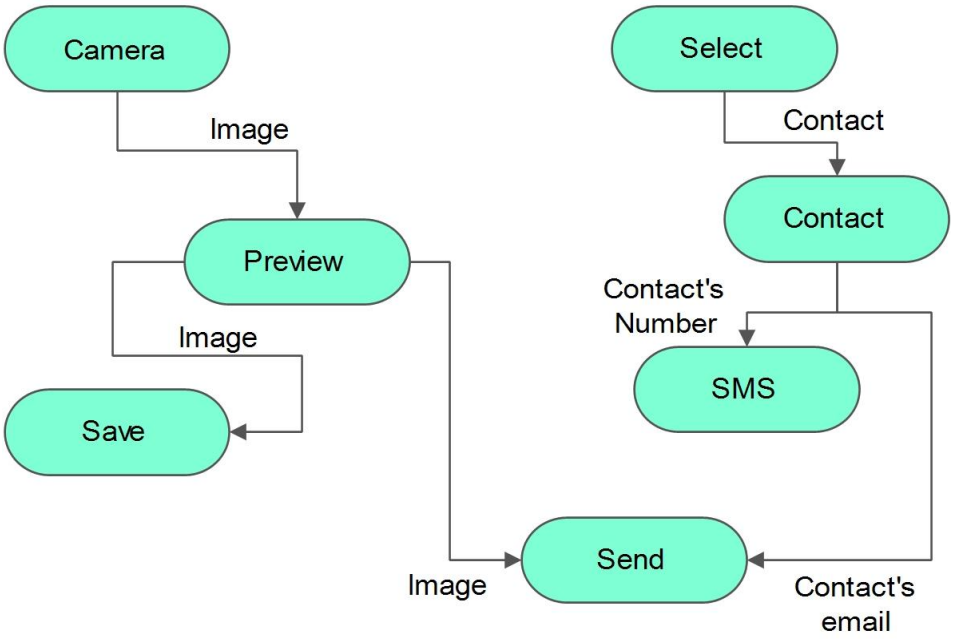

Figura 5.5 Grafo della microApp

Il risultato ottenuto è visibile nella seguente Fig. 5.6.

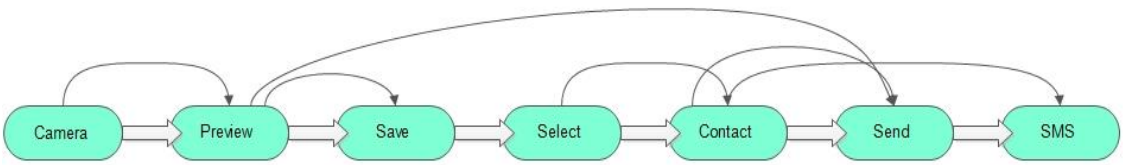

Figura 5.6 Componenti ordinate secondo il flusso di esecuzione

L'esecuzione come previsto partirà dal primo elemento definito che troviamo nel file di descrizione, la componente Camera, per poi proseguire con le altre componenti collegate, tenendo in considerazione la necessità di input da parte

della componente Mail, che otterrà gli elementi da Preview e Contact eseguiti in precedenza.

Eseguita l'inizializzazione, ovviamente trasparente all'utente, viene avviata la prima componente (Fig. 5.7).

Figura 5.7 Esecuzione della Camera

Figura 5.8 Esecuzione della Preview

Dopo aver scattato la foto, l'applicazione passa alla componente successiva, Preview (Fig. 5.8), che non è altro che un'anteprima dell'immagine appena scattata.
Quando l'utente preme il tasto Next, si passa alla componente successiva, nel caso in cui voglia tornare indietro, in qualsiasi momento può premere

il tasto Prev che permetterà all'utente di tornare alla componente precedente.

La componente Save della microApp prevede il salvataggio dei dati in input. In questo caso è salvata solamente l'immagine perché è l'unico dato disponibile (Fig. 5.9).
Premendo Next, la prossima componente è quella di selezionare i contatti a cui vogliamo inviare l'email o l'SMS (Fig. 5.10).

Figura 5.9 Esecuzione di Save

Figura 5.10 Esecuzione della Select

La componente Contact prevede semplicemente la visualizzazione dei vari contatti selezionati, con tutte le informazioni dettagliate (vedere Fig. 5.11).

Figura 5.11 Esecuzione di Contact

Infine le ultime componenti ad essere eseguite saranno quelle per l'invio della mail e dell'SMS (vedere Fig. 512 e Fig. 5.13). In queste componenti verranno caricate le informazioni dei contatti, rispettivamente l'indirizzo email e il numero telefonico del contatto. Inoltre alla mail viene allegata l'immagine che è stata scattata dalla prima componente.

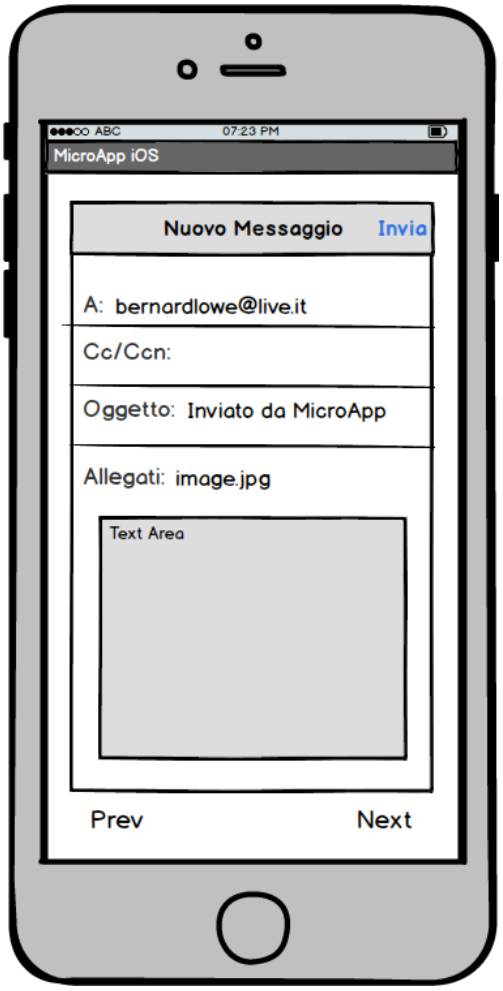

Figura 5.12 Esecuzione della Mail

Figura 5.13 Esecuzione di SMS

Una volta compilate tutte le aree di testo si può procedere con l'invio, concludendo così i passi dell'applicazione. Quest'App può rivelarsi utile in casi in cui si desidera inviare una mail simile ai post dei moderni social network come Facebook. Infatti oltre all'invio di messaggi a contenuto testuale si possono allegare anche dei file. Inoltre è possibile inviare un ulteriore messaggio di notifica a chi si invia l'email oppure agli utenti che ne sono sprovvisti. Infine, questo è uno dei tanti esempi completi che sono stati pensati durante la fase di sviluppo di MicroAppEngine.

Conclusioni

In questa tesi abbiamo descritto lo sviluppo di MicroAppEngine, il modulo di un'applicazione innovativa a supporto dell'utente, chiamata MicroApp iOS, che mediante un editor visuale, permette agli utenti di creare una propria app personalizzata, in maniera veloce e flessibile, senza la necessità di scrivere nemmeno una riga di codice, ma solo attraverso dei piccoli gesti eseguiti con il dispositivo iOS.

La parte riguardante la gestione dell'ambiente applicativo, in particolare la gestione del flusso di esecuzione e lo scambio di dati tra le componenti sono trattati in maniera completa nella tesi (8) del mio collega che ha collaborato al progetto.

In questa tesi ci siamo occupati della parte riguardante l'isolamento delle componenti e dell'estendibilità dell'applicazione, cioè il meccanismo che permette di isolare il comportamento interno della componente al motore di esecuzione in maniera tale che quest'ultimo non debba preoccuparsi di com'è fatta la componente, e di aggiungere in maniera semplice nuove componenti per la composizione delle microApp.

I risultati ottenuti dal lavoro svolto sono soddisfacenti, nonostante MicroApp iOS sia ancora in fase prototipale. Infatti, il suo motore di esecuzione MicroAppEngine ha risposto positivamente sia in termini di efficienza che di usabilità. Abbiamo avuto modo di testare l'applicazione sull'emulatore messo a disposizione da Xcode e sia su dispositivo iOS. L'esempio utilizzato per il test, anche se non molto complesso, ha fornito le informazioni necessarie per accertarci che il flusso di dati tra una componente e l'altra avvenga in modo corretto e che le componenti vengano ordinate nel modo giusto.

La progettazione e lo sviluppo del lavoro sono avvenuti sotto la guida della Prof.ssa Tortora e del Dott. Risi, che ci hanno guidato nel migliore dei modi al raggiungimento dell'obiettivo. Il lavoro svolto ci ha portato ad avere un'applicazione più funzionale rispetto alla precedente versione Android,

migliorando il modello di esecuzione, la gestione delle componenti, la gestione del flusso dei dati e semplificando l'aggiunta di nuove componenti.
In futuro, si prevede che l'applicazione possa crescere, fornendo componenti più avanzate di quelle attualmente implementate, oltre allo sviluppo di un editor visuale che permetta di estendere le componenti in qualsiasi direzione; si potrebbe poi permettere un'esecuzione circolare delle componenti in modo tale che una volta che queste abbiano terminato il loro compito, la microApp possa ricominciare nuovamente il suo ciclo di esecuzione; fornire delle componenti condition che permettano la realizzazione dei cicli; infine si potrebbe permettere agli End-User di personalizzare anche la User Interface dell'applicazione.
Ovviamente queste sono solo alcune idee che potrebbero essere realizzate, i colleghi che vorranno continuare lo sviluppo di questo progetto avranno modo di dare sfogo a tutta la loro creatività.
In conclusione posso dire che è stata una bellissima esperienza perché durante la fase di sviluppo ho avuto modo di avere un primo impatto verso il mondo del lavoro, di conoscere un nuovo linguaggio di programmazione e un nuovo ambiente di sviluppo, che mi hanno permesso di amplificare le mie conoscenze in ambito lavorativo e informatico.

Bibliografia

1. [Online] Apple Inc., iOS Developers Library, 2012, https://developer.apple.com/library/content/navigation/

2. **HILLEGASS, CHRISTIAN KEUR & AARON**. *iOS Programming: The Big Nerd Ranch Guide*. 2016

3. *Advanced iOS Programming: The Big Nerd Ranch Guide*. 2016

4. Swift Programming: The Big Nerd Ranch Guide

5. Libro Bianco. [Online] https://images.apple.com/it/business/docs/iOS_Security_Guide.pdf

6. matrixproject.net. [Online] http://www.matrixprojects.net/p/xcodebuild-export-options-plist/

7. *Visual Mobile Computing for Mobile End-Users*. **Maurizio Tucci, Rita Francese. Michele Risi, Genoveffa Tortora**. 2015

8. **Gagliardi Roberto**. *Progettazione e sviluppo di MicroApp Engine su iOS: modello di esecuzione e gestione dei dati*. 2017

9. **Valenza Daniele**. *Progettazione e implementazione di MicroAppEngine: Componenti e modello di esecuzione*. 2011

10. **Immobile Luigi**. *Activity e modello di estendibilità in MicroAppEngine*. 2011

11. Giuseppe Dell'Abate's Blog. [Online] https://dellabate.wordpress.com/2012/07/27/gof-patterns-strategy/

Ringraziamenti

Vorrei fare un ringraziamento a tutte le persone che mi sono state vicine in questi anni, soprattutto in questo ultimo periodo, in particolar modo la mia famiglia che mi ha permesso di continuare gli studi e che non ha mai smesso di credere in me. Mi auguro che tutti i loro sacrifici siano in questo modo, almeno in parte, ripagati.

Ringrazio calorosamente la mia ragazza Maria Camilla che con amore, pazienza e fiducia mi ha sostenuto per tutto questo periodo, con la speranza che continui a farlo a lungo.

Ringrazio altresì la Prof.ssa Tortora e il Dott. Risi per la disponibilità dimostrata durante la stesura di questo lavoro di tesi.

Un grazie ai miei amici che ci siamo sempre sostenuti a vicenda, nella buona e nella cattiva sorte, sia durante le fatiche, lo sconforto, le giocate fatte insieme e le discussioni in chat, che hanno caratterizzato il nostro percorso nei momenti di gioia e soddisfazione al raggiungimento del traguardo.

Infine ringrazio tutte quelle persone che hanno voluto conoscermi, quelle che mi hanno fatto sorridere e quelle che mi sono state accanto col passare degli anni.

Ringrazio anche tutte quelle persone che invece hanno fatto l'esatto opposto, perché mi hanno aperto gli occhi e mi hanno fatto capire che non tutte le persone riescono a vivere senza dover indossare una maschera, questa anche se ben fatta, si arriva sempre, con un po' di attenzione, a distinguerla dal volto.

Concludo dicendo che questo è solo un piccolo passo verso il futuro che mi aspetta, prometto di impegnarmi sempre di più affinché diventi un uomo capace di poter ottenere tutto dalla vita. Plus Ultra!

Giuseppe Abagnale