UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica applicata

Corso di Laurea in Ingegneria Informatica

Elaborato di laurea

"Mobile SAND": progetto e sviluppo di un'app iOS per il coding hands-free.

Relatore:

Ch. Prof. Luca Greco

Candidato:

Alfonso Cavallo

Matricola:

0612703985

Anno Accademico 2019/2020

Dedicato a mia nonna **Tina**,
per la pazienza e l'affetto che mi ha sempre dimostrato,
con la consapevolezza che ne sarebbe stata orgogliosa.

SOMMARIO

INTRODUZIONE

L'obiettivo di questo capitolo è presentare il dominio del problema ed i processi decisionali che hanno condotto all'ideazione della soluzione.

L'efficienza e la comodità delle soluzioni informatiche hanno portato, negli ultimi anni, le moderne tecnologie nelle case di centinaia di milioni di persone.
Con l'espansione del mercato e l'allargamento della nicchia dei consumatori, si è dedicata una forte attenzione anche al tema dell'accessibilità di tali prodotti. Come riportato nel *World report on disability*[1] circa una persona su sette soffre di una disabilità di qualche tipo. Secondo i dati raccolti nell' European health interview survey[2] (EIHS), il 10.1% della popolazione europea soffre di gravi impedimenti sensoriali o fisici. Volendo, ancor più nello specifico, indagare sulla sfera delle disabilità motorie, le statistiche del Center for Desease Control and Prevention[3] (CDCP) riportano un conto, negli Stati Uniti, di circa 15 milioni di persone affette da severe limitazioni fisiche.
Sebbene questo tema sia fortemente sentito da molte aziende come Apple, Google ed Amazon, solo per citare alcune tra le più note, vi sono ancora settori tecnici in cui tali tecnologie agevolative sono ancora in via di sviluppo. Un esempio immediato è il campo della programmazione, dove le necessarie interazioni, manuali con la tastiera e visive con lo schermo, si mostrano ostacoli proibitivi per certi tipi di disabilità. Fortunatamente, nel corso degli anni, grazie anche all'apprezzamento della clientela, l'approccio *hands free*, libero dall'utilizzo delle mani, si è visto protagonista di un notevole sviluppo di cui gli assistenti vocali sono l'emblema. Questi dispositivi, nati come strumento di ricerca per il WEB, hanno rapidamente conquistato il mercato e prevedono vendite entro l'anno 2023 di un numero superiore perfino alla popolazione mondiale. Questo comune interessamento ha consentito a queste tecnologie innovative di svilupparsi in tempi molto brevi, migliorando in termini di affidabilità ed efficienza, allo scopo di espandere il loro impiego in tanti altri settori come l'e-commerce di cui si stima arriveranno a monopolizzare circa il 50% delle interazioni. Proprio grazie a tale sviluppo, tale approccio è stato impiegato in progetti come Voice Code[4] e Talon Voice[5], supporti alla programmazione tramite voce. I linguaggi di programmazione più attenzionati da queste applicazioni sono generalmente quelli che, per loro natura, maggiormente si avvicinano al linguaggio naturale, come Python e Dragon.
Attesi, dunque, i benefici e le potenzialità della programmazione tramite voce, che hanno motivato lo sviluppo di queste soluzioni, sarebbe possibile trasferire questa funzionalità, liberi dai vincoli di una tastiera fisica, in una dimensione tascabile e, se possibile, ancor più confortevole ed immediata, mediante l'implementazione di un ambiente di sviluppo pensato per smartphone. Data anche la comodità di questo tipo di approccio, si vuole sviluppare un applicativo capace di espandere il proprio mercato anche oltre il target per cui esso nasce, ed a cui esso, in primis, è rivolto. Nasce così l'idea di **Mobile SAND** (Speech Assistant for Native Development), un'ambiente nativo per lo sviluppo in Swift tramite voce che richiami nel suo acronimo anche il "sandbox", ossia il concetto di un *playground* (terreno di gioco), aperto ai programmatori novizi.

[1] https://www.who.int/publications/i/item/world-report-on-disability (WHO, 2011)
[2] https://ec.europa.eu/eurostat/ (eurostat, 2019)
[3] https://www.cdc.gov/nchs/fastats/disability.htm (CDC, 2018)
[4] https://voicecode.io/
[5] https://talonvoice.com/docs/

Si riporta una breve presentazione dei capitoli del presente lavoro di tesi e del loro contenuto:

- **Capitolo 1, Tecnologie impiegate**: una panoramica dei mezzi tecnologici e delle risorse hardware e software impiegate nello sviluppo della soluzione.
- **Capitolo 2, Progettazione**: una descrizione delle scelte di progettazione e delle motivazioni alla loro base.
- **Capitolo 3, Implementazione**: un'analisi tecnica dello sviluppo dell'applicativo.
- **Capitolo 4, Conclusioni e sviluppi futuri**: un riassunto del lavoro svolto e dei suoi sviluppi futuri.

1 TECNOLOGIE IMPIEGATE

L'obiettivo di questo capitolo è la presentazione di una panoramica delle tecnologie e delle risorse utilizzate per la realizzazione del progetto Mobile Sand.

1.1 TECNOLOGIE APPLE

1.1.1 Risorse Hardware

Le tecnologie impiegate per la realizzazione del presente lavoro di tesi sono, in buona parte, risorse di marca Apple. La partecipazione all'Apple Foundation Program, un programma accademico finalizzato all'acquisizione di competenze relative allo sviluppo di applicazioni iOS, ha infatti consentito l'accesso alle risorse tecnologiche necessarie all'implementazione di questa soluzione. Il kit messo a disposizione include un MacBook Pro 16 ed un iPhone XR.

1.1.2 Risorse Software

Al fine di concepire una soluzione nativa, si è optato per la realizzazione di un applicativo orientato all'utilizzo del moderno linguaggio Swift, rilasciato da Apple Inc. nel corso del 2014. L'ambiente di sviluppo integrato scelto per lo sviluppo del progetto finale è Xcode, una soluzione offerta direttamente dalla Apple e ottimizzata per lo sviluppo di applicativi pensati per iOS.

I prototipi generati sono stati manualmente testati nel corso dello sviluppo sul simulatore integrato nell'ambiente e su diverse versioni di iOS installate su iPhone.

1.2 ALTRI SERVIZI E SOFTWARE

Il servizio di *versioning and continuous integration* utilizzato per tener traccia delle modifiche del progetto in corso di sviluppo è gitLab. Un sistema di *Version Control* è uno strumento utile alla gestione di versioni multiple di un insieme di informazioni.

Per la realizzazione del logo e delle grafiche dell'applicazione si è impiegato l'editor Sketch integrato con il plugin sketch-iconfont sottoposto a licenza MIT.

1.3 RISORSE

Le risorse utilizzate includono icone, caratteri di testo ed altre componenti grafiche fornite dai pacchetti standard di Apple integrati in Xcode.

Il logo dell'applicazione è costruito attorno all'icona "dune" gratuita offerta con licenza di libero utilizzo dall'account Freepik sul sito flaticon.com.

1.4 FRAMEWORKS

La realizzazione del presente applicativo richiede l'utilizzo di diversi *frameworks*[6], che supportino l'implementazione di determinate funzionalità dell'app finale. Un *framework* è una *structured directory* che contiene delle librerie e le risorse ad esse associate.

1.4.1 Foundation

Il *framework* Foundation definisce tutti i servizi ed i tipi di dati essenziali per l'implementazione dell'app. Esso fornisce un insieme di funzionalità che compongono il livello base dell'applicativo. Alcuni esempi sono: *text processing*, calcoli su tipi di dati complessi, gestione e ordinamento di collezioni.

1.4.2 UIKit

Il *framework* UIKit fornisce tutte le risorse necessarie alla realizzazione e l'amministrazione dell'interfaccia dell'applicativo. Esso definisce tutte le infrastrutture quali, finestre e *views* che consentono di gestire le interazioni tra l'utente, il sistema e l'app. Fornisce, inoltre, le seguenti funzionalità:

- Supporto alle animazioni;
- Supporto alla gestione di documenti;
- Supporto alla gestione di immagini e della stampa a schermo;
- Informazioni sul dispositivo;
- Gestione e rappresentazione del testo;
- Supporto alla funzione di ricerca;
- Supporto all'accessibilità;
- Supporto alle estensioni dell'applicativo;
- Gestione delle risorse.

Particolare importanza viene data ai *View Controller*, le classi responsabili della gestione delle interazioni con la schermata.

1.4.3 AVFoundation

Il *framework* AVFoundation consente di lavorare con *assets* audiovisivi. In particolare, fornisce gli strumenti necessari a configurare i sistemi ed a processare le informazioni audiovisive. Questo *framework* combina sei aree tecnologiche che insieme soddisfano tutte le necessità legate alla manipolazione dei media precedentemente citati sulle piattaforme Apple. Vengono di seguito riportate le funzionalità principali:

- Registrazione;
- Processing;
- Sintetizzazione;
- Controllo;
- Importazione;
- Esportazione.

[6] La documentazione relativa a tutti i frameworks della Apple è consultabile all'indirizzo: https://developer.apple.com/documentation/technologies (Apple Inc., 2020)

Le funzionalità di maggiore utilizzo per il presente applicativo sono quelle di registrazione dell'audio. Le due classi di maggiore utilizzo all'interno di Mobile SAND sono le seguenti:

- **AVAudioSession**: comunica al sistema operativo il modo nel quale l'applicazione intende utilizzare il contenuto audio, mantenendo l'astrazione dal lavoro dell'hardware. È possibile configurare il comportamento di questo oggetto modificandone la *category*. Un esempio è `playback`, la categoria di maggiore utilizzo, che stabilisce la centralità della funzionalità di *audio playback* nel contesto dell'applicazione;
- **AVAudioEngine**: è un insieme di oggetti rappresentanti nodi di registrazione che vengono impiegato per generare, processare ed eseguire il *record* e la riproduzione di segnali audio.

Altre funzionalità impiegate nell'applicativo concernono la funzione di sintetizzazione vocale per la riproduzione di stringhe sotto forma di voce. Le classi adoperate sono le seguenti:

- **AVSpeechUtterance**: definisce il testo da riprodurre ed i parametri relativi;
- **AVSpeechSynthesisVoice**: rappresenta la voce a cui è affidata la sintesi;
- **AVSpeechSynthesizer**: è responsabile dell'effettiva riproduzione del discorso sintetizzato a partire dal testo.

1.4.4 Speech

Il *framework* Speech consente di processare "discorsi" dell'utente, memorizzati in *files* audio preregistrati oppure processati in tempo reale e di ricevere trascrizioni accompagnate da un *confidence level* che stabilisce l'affidabilità del risultato. Esso fornisce un insieme di funzionalità per il riconoscimento di parole all'interno di audio registrati o *live*, similmente alla funzionalità di dettatura accessibile dalla tastiera virtuale. Il riconoscitore responsabile di queste operazioni è disponibile in diverse lingue ma può essere configurato su una sola di esse per volta. Sebbene tale funzionalità sia configurata anche localmente per un certo numero di lingue tramite un riconoscitore *On-device*, occorre sempre assumere che l'operazione di trascrizione richieda la connessione ad Internet.

Le classi di maggiore utilizzo per il *processing* della voce sono le seguenti:

- **SFSpeechRecognizer**: è l'oggetto principale per l'utilizzo delle funzionalità di riconoscimento sulla lingua configurata. È responsabile della richiesta e della verifica delle autorizzazioni fornite al servizio di *recognition* e dell'esecuzione di *tasks* di interpretazione. Fornisce tutta una serie di proprietà per l'amministrazione di tali parametri ed è sottoposto a delle limitazioni sull'utilizzo dei servizi, come, ad esempio, la lunghezza massima di un minuto degli audio da convertire;
- **SFSpeechAudioBufferRecognitionRequest**: specializza la classe astratta `SFSpeechRecognitionRequest` e consente di eseguire l'interpretazione di un contenuto audio. La fonte di tale contenuto, che è in questo caso il microfono dell'iPhone, viene passata ad esso come argomento per la funzione `append`, che la configura come sorgente delle informazioni da processare.
- **SFSpeechRecognitionTask**: rappresenta l'effettivo *task* di riconoscimento e consente di monitorare lo stato di avanzamento del processo di elaborazione del contenuto audio processato dal riconoscitore.

Si riportano di seguito le classi impiegate nell'elaborazione del risultato del processo di trascrizione:

- **`SFSpeechRecognitionResult`**: contiene un insieme di possibili interpretazioni, parziali o complete, del contenuto audio sottoposto a trascrizione, accompagnate da un fattore di affidabilità. Tale oggetto non viene direttamente creato ma costruito dal *framework* e passato come parametro agli opportuni *handlers* o *delegates* che gestiscono la notifica di risposta dal server responsabile del *task* di trascrizione;
- **`SFTranscription`**: fornisce una rappresentazione testuale del parlato, registrato dal microfono, ed elaborato dal server. Da esso è possibile ricavare un insieme di *utterances*, ossia delle parole o delle espressioni che assumono un unico significato per il riconoscitore. La proprietà `formattedString` consente di ricavare una stringa testuale partendo da questo risultato. Anch'esso non viene direttamente costruito ma ricavato a partire da un'istanza di `SFSpeechRecognitionResult` ottenuta in risposta ad una richiesta di riconoscimento.

1.5 COMPILATORE ONLINE

Al fine di eseguire frammenti di codice, Mobile SAND stabilisce una connessione *http* con un compilatore online. Si espone di seguito una breve definizione di compilatore:

> Un compilatore è un programma informatico che traduce una serie di istruzioni scritte in un determinato linguaggio di programmazione (codice sorgente) in istruzioni di un altro linguaggio (codice oggetto).[7]

Il servizio di compilazione, ossia di traduzione di codice Swift in codice eseguibile, utilizzato dal presente applicativo è accessibile tramite l'API[8] di paiza.io. Per l'impiego di tale compilatore è necessario l'invio di messaggi di richiesta *http* all'*host* paiza.io con l'URL relativo al servizio richiesto.
I tipi di richiesta e gli URL associati sono di seguito elencati:

- **POST /runners/create**: Consente di creare una sessione di compilazione specificando diversi parametri come, ad esempio, il codice da compilare ed il linguaggio di programmazione in questione. Il messaggio di risposta conterrà l'ID della sessione;
- **GET /runners/get_status**: Consente di ricevere un messaggio di risposta con le informazioni sullo stato della compilazione, inviando come parametro, nell'URL, l'ID della sessione;
- **GET /runners/get_details**: Consente di ricevere un messaggio di risposta con le informazioni relative alla compilazione ed all'esecuzione del codice, eseguite da un calcolatore remoto, inviando come parametro, nell'URL, l'ID della sessione.

[7] Compilatori. Principi, tecniche e strumenti (Aho, Sethi, & Ullman, 2006)
[8] Le API Internet sono un insieme di regole che il modulo software mittente deve seguire in modo che i dati siano recapitati al programma di destinazione. (Kurose & Ross, 2013)

2 PROGETTAZIONE

L'obiettivo di questo capitolo è la presentazione del processo di realizzazione del software Mobile SAND, partendo dalla scelta del metodo, arrivando alla descrizione della struttura dell'applicativo.

2.1 SCELTA DEL METODO DI PROGETTAZIONE

L'obiettivo finale del presente lavoro di tesi è la realizzazione di un'applicazione che fornisca un ambiente di sviluppo integrato per la programmazione tramite voce. In tal senso le librerie di Apple forniscono notevoli agevolazioni tramite l'offerta di numerosi servizi per l'interfacciamento con microfono e l'interpretazione del parlato. Il *framework* chiave per l'implementazione delle funzionalità dell'assistente vocale è *Speech*, che fornisce l'accesso ad un sistema remoto per la traduzione del parlato in testo scritto. Si vuole intraprendere, inoltre, uno sviluppo flessibile all'evoluzione di queste tecnologie, il cui stato dell'arte è tuttora in costante aggiornamento.
Non è da escludere, poi, la possibilità di definire, grazie allo specifico target di consumatori delineato nella prima fase di analisi, la caratterizzazione di un preciso modello utente che supporti il progettista nell'immaginazione di scenari di utilizzo reali e pratici.
Prefissandosi, infine, l'obbiettivo di concentrare gli sforzi di lavoro sulla produzione di funzionalità complete piuttosto che documentazione tecnica. L'approccio che si è trovato più conforme a questi scopi è sicuramente la metodologia di progettazione agile.

2.2 METODO AGILE

2.2.1 Introduzione al metodo agile

I metodi di sviluppo agili nacquero verso le fine degli anni '90 sulla base di idee condivise da diverse persone e maturate quasi allo stesso tempo. Questo tipo di progettazione nacque in risposta al gravoso impatto dei tempi di progettazione sullo sviluppo dei progetti. Capitava, infatti, soprattutto in contesti aziendali di piccole o medie dimensioni, che si passasse più tempo a pianificare il lavoro che a svolgerlo, inficiando negativamente la produttività. In riposta a questa problematica, la metodologia agile scardina diversi principi base della progettazione accentrando l'interesse maggiore sugli individui, le interazioni e la flessibilità del prodotto.

> Stiamo scoprendo modi migliori di creare software, sviluppandolo e aiutando gli altri a fare lo stesso. Grazie a questa attività siamo arrivati a considerare importanti:
>
> **Gli individui e le interazioni** più che i processi e gli strumenti
> **Il software funzionante** più che la documentazione esaustiva
> **La collaborazione col cliente** più che la negoziazione dei contratti
> **Rispondere al cambiamento** più che seguire un piano
>
> Ovvero, fermo restando il valore delle voci a destra, consideriamo più importanti le voci a sinistra[9]

[9] https://agilemanifesto.org/iso/it/manifesto.html (Manifesto per lo Sviluppo Agile del Software, 2001)

Dato l'enorme successo conseguito da questa metodologia di progettazione, è possibile affermare che quasi tutti i piccoli progetti al giorno d'oggi seguano, per comodità, un approccio agile.

> Virtualmente, tutti i prodotti software e le applicazioni oggi vengono sviluppati utilizzando un approccio agile.[10]

2.2.2 Le funzionalità

Il processo di progettazione agile predica flessibilità, e ne consegue che molti metodi ad esso ispirati seguano uno sviluppo iterativo incrementale, ovvero basato sulla ciclica ripetizione di passaggi di pianificazione, che gradualmente creano i componenti del prodotto finale. Talvolta, tali tecniche tendono addirittura ad estremizzare questa pratica come, ad esempio, la tecnica di *Extreme Programming*. L'applicativo viene, per i motivi precedentemente citati, suddiviso in funzionalità, ognuna delle quali è a sua volta ripartita in task, ed integrata opportunamente nel software solo dopo aver attraversato una fase di implementazione e di testing.

Capita spesso, che questo tipo di lavorazione risulti tanto frenetica che il tempo ad essa dedicato possa essere quantificato in giorni o settimane, laddove avrebbe richiesto tempi ben più lunghi in altre metodologie più classiche basate sulla pianificazione. Questa conseguenza è il risultato di un evidente ispirazione derivata dalla tecnica di produzione snella del mondo dell'hardware. I programmatori che adottano queste tecniche, sacrificano tutto ciò che non risulta rilevante agli occhi del cliente, realizzando produttività ed efficienza al costo di una documentazione più esaustiva considerata, secondo i dettami del manifesto agile, meno importante del prodotto in sé.

2.2.3 Storie utente

Come espresso nelle sezioni precedenti, l'aspetto commerciale assume un ruolo primario nell'approccio agile al processo di produzione del software. La quasi totalità delle fasi di sviluppo, infatti, è costantemente concordata e supervisionata dall'utente o da un suo mediatore. Il punto d'incontro tra il programmatore ed il cliente sono le storie utente. La più classica documentazione viene infatti sostituita da scenari descrittivi delle interazioni tra l'utente e l'applicazione. Vengono prodotte delle "carte della storia" o *story cards*, delle narrazioni che aiutino il programmatore a figurarsi gli aspetti pratici delle funzionalità e che forniscano al cliente una visione chiara del risultato finale, non contaminata dai tecnicismi della fase implementativa.

2.3 ORGANIZZAZIONE DEL LAVORO

In un progetto che privilegia il soddisfacimento dell'utente sopra ogni altro obiettivo, il processo fondamentale è la definizione del *front-end*, ossia dell'interfaccia responsabile delle interazioni tra esso e l'intero sistema. Una volta realizzato un modello grafico, che si arricchirà di tutti i componenti necessari all'utente per l'utilizzo dell'ambiente, è possibile passare alla realizzazione della logica applicativa.

Mobile SAND è un progetto fortemente improntato all'offerta di funzionalità accessibili ad un pubblico quanto più vasto possibile, se non proprio mirato all'elusione di specifici

[10] Sviluppo agile del software, Metodi agili (Sommerville, 2017)

impedimenti fisici che ostacolano la programmazione. Il primo passo è dunque quello di effettuare una sessione di *brainstorming* finalizzata ad ottenere un insieme di funzionalità da integrare gradualmente all'interno dell'applicativo. Questi elementi vengono successivamente analizzati e sistemati nel seguente ordine di priorità:

1. **Dettatura**: in seguito, occorre garantire all'attore la possibilità di colloquiare con l'assistente vocale, convertendo il parlato in testo digitale;
2. **Programmazione tramite voce:** è a questo punto possibile creare un sistema capace di interpretare il linguaggio naturale e convertirne specifiche parole ed espressioni in costrutti e *keywords* del linguaggio di programmazione;
3. **Esecuzione del codice:** l'obiettivo successivo, avendo un codice eseguibile, è quello di renderne possibile la compilazione e, per l'appunto, l'esecuzione;
4. **Controllo dell'applicazione tramite voce:** risolti gli aspetti critici della problematica principale è possibile concentrarsi sulla navigabilità dell'applicazione, implementando un sistema di controllo che permetta di accedere alla funzionalità principale ed a tutte quelle in via di implementazione mediante l'esclusivo utilizzo di comandi vocali;
5. **Accesso alla guida:** a questo punto le basi sono già state realizzate e diventa opportuno guidare l'utente all'utilizzo del software tramite un sistema di info;
6. **Organizzazione dei file:** è infine possibile rendere più comodo l'impiego dell'ambiente tramite una funzionalità di gestione dei files.

Si precisa, che le funzionalità sopracitate rappresentano l'insieme che partecipa a comporre il prodotto finale, alcune delle quali sono state aggiunte, proprio grazie alla flessibilità del metodo agile, in corso d'opera. Altre funzionalità non riportate in questo paragrafo saranno esposte come possibili sviluppi futuri in un apposito capitolo.
Nel corso dello sviluppo, il progetto viene anche ciclicamente sottoposto ad un processo di *refactoring*, al fine di conseguire leggibilità e strutturazione del codice.

2.4 INTERFACCIA GRAFICA

In un'applicazione mobile, l'interfacciamento con l'utente è costruito attorno ad una schermata popolata di componenti quali pulsanti e campi di testo, che forniscono l'accesso ai servizi del software.
Approcciare un problema seguendo un approccio agile implica intraprendere uno sviluppo incrementale dove ogni componente è protagonista di una fase di pianificazione interamente dedicata ad esso ed è considerato autonomo seppur compartecipe al prodotto finale. Tuttavia, sarebbe inopportuno considerare la schermata principale dell'applicazione una funzionalità al pari delle altre, poiché il suo scopo principale non è quello di implementare logica applicativa bensì di mediare le interazioni tra l'utente e le effettive funzionalità del sistema.
Lo scopo di questa fase di progettazione è la strutturazione dell'interfaccia grafica della schermata principale e la realizzazione di un suo modello grafico.
Sebbene il metodo di progettazione agile suggerisca un'integrazione graduale delle diverse funzionalità all'interno del software, è già possibile costruire una schermata di base che sarà popolata successivamente da tutti i componenti grafici necessari ad implementare le interazioni con l'utente.

Per la realizzazione della schermata in Figura 1 sono stati presi in considerazione le linee guida fornite da *Human Interface Guidelines*[11] di Apple.
Il design grafico è stato sviluppato seguendo i principi fondamentali di:

- **Integrità estetica**: l'estetica dell'applicazione deve esaltare le sue funzionalità;
- **Consistenza**: l'app deve uniformarsi a dei precisi standard grafici ben noti associati ad un preciso comportamento;
- **Manipolazione diretta**: la manipolazione dei contenuti dello schermo deve essere diretta per rendere l'esperienza dell'utente immediata ed intuitiva;
- **Feedback**: a specifiche azioni devono corrispondere specifici feedback precettivi;
- **Metafore**: il comportamento degli oggetti virtuali deve ispirarsi ad esperienze tipiche della vita quotidiana;
- **Controllo dell'utente**: L'app può suggerire determinate interazioni ma mai sottrarre il controllo all'utente.

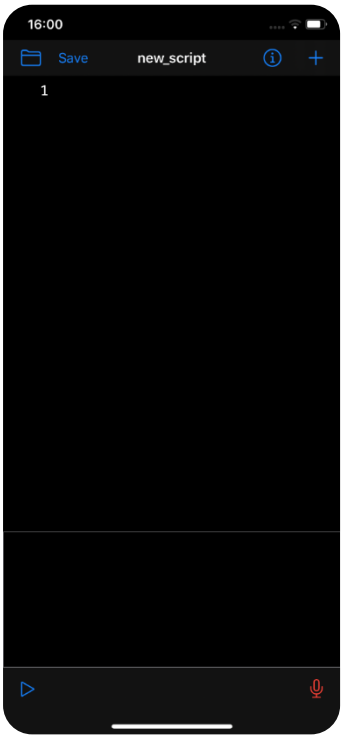

Figura 1 - Schermata principale del prodotto finale.

Tutti i pulsanti vengono rappresentati tramite icone predefinite di sistema che suggeriscono il comportamento ad esse associato.
Viene utilizzato il carattere di testo di default per le etichette, mentre viene scelto per le aree di testo dedicate al codice eseguibile il font "Menlo", impiegato anche da Xcode come standard per la programmazione.
Una certa importanza è attribuita anche alla colorazione dei componenti, che fornisce informazioni di stato e feedback in risposta alle azioni dell'utente. I colori selezionati per il presente applicativo sono le seguenti tinte predefinite di sistema:

- **Blu**: è associato ai pulsanti ed è quindi indicativo dell'esecuzione di una funzionalità;
- **Rosso**: è associato al pulsante del microfono e rappresenta l'inizio di una registrazione. L'impiego di tale colore è un'eccezione atta a distinguere la funzionalità principale da tutte le altre;
- **Grigio**: è utilizzato per indicare una funzionalità inaccessibile;
- **Verde**: indica una registrazione in corso.

A seconda dello stato della registrazione e delle autorizzazioni fornite all'applicativo, il *button* del microfono muterà icona, colore e funzionalità. I pulsanti popolano la *navbar* (barra di navigazione) e la *toolbar* (barra degli strumenti) disposte rispettivamente ai margini verticali superiore e inferiore dello schermo. Il resto della schermata è, infine, colmato da due aree di testo. L'interfaccia si adatta allo schermo grazie ad un sistema di *constraints* (vincoli) ed è configurabile su di essa un tema chiaro o scuro.

[11] https://developer.apple.com/app-store/review/guidelines/ - (Apple Inc., 2020)

2.5 LE FUNZIONALITÀ

L'obiettivo di questa sezione è descrivere tutte le funzionalità dell'applicativo illustrando il loro processo di progettazione. Inoltre, viene presentata per ognuna di esse una *story card* che la descriva in maniera pratica.

2.5.1 Dettatura

La prima funzionalità da progettare è la funzionalità di dettatura. Deve essere reso possibile all'utente comunicare con un assistente che sia in grado di comprendere i suoi comandi e, quindi, di tradurre le parole in testo scritto. Seguendo l'ispirazione tratta dalla tecnica di programmazione estrema, si descrive narrativamente l'interazione tra l'utente e l'applicazione mediante la definizione di una *story card*. Per fare ciò, è prima necessario modellare un profilo utente che rispecchi le caratteristiche del target a cui è diretta la soluzione in progetto. Si immagina, dunque, l'esempio di un utente inabilitato a scrivere tramite tastiera ma che sia capace di premere un pulsante o di utilizzare le funzioni di accessibilità del suo iPhone per raggiungere tale scopo. Al fine di avere una rappresentazione più diretta e informale, si assegna un nome a tale utente, che nel caso del presente lavoro sarà chiamato, a partire da ora, con il nome di "Johnny Debugger".

Dettatura
Johnny Debugger ha una disabilità motoria che non gli consente di muovere liberamente le braccia, ma vorrebbe comunque imparare il linguaggio di programmazione Swift. Johnny apre l'applicazione e clicca sul pulsante con l'icona del microfono. Vedendo che l'icona del pulsante è diventata una forma d'onda verde capisce che il microfono è in ascolto ed inizia a parlare. Mentre Johnny detta le sue parole vengono trascritte sull'area di testo della schermata in tempo reale. Quando Johnny ha finito interrompe la sessione di dettatura cliccando di nuovo sul pulsante.

Figura 2 - Story Card sulla dettatura.

Vengono a questo punto individuati tre task principali:

- Apertura di un canale audio;
- Conversione del parlato in testo;
- Rappresentazione del testo nell'area di testo predisposta.

2.5.2 Programmazione tramite voce

Allo scopo di consentire la programmazione tramite voce l'assistente vocale deve essere in grado di processare ciò che viene dettato in tempo reale ed interpretare le parole come parole chiave del linguaggio di programmazione. Tuttavia, risulta molto più semplice elaborare testo piuttosto che suoni e fortunatamente la funzione di dettatura precedentemente realizzata consente di ottenere una trascrizione digitale del parlato. Partendo da una stringa di testo, l'applicativo può effettuare una conversione in codice eseguibile, purché conservi in uno *storage* tutte le informazioni sulle corrispondenze tra *keywords* e parole del linguaggio naturale. Bisogna, inoltre, soddisfare determinate esigenze dell'utente che possono sopraggiungere in fase di programmazione, come, ad esempio, la necessità di correggere un errore. Appare conveniente passare in rassegna

tutte le eventuali occorrenze che possono coinvolgere il programmatore ed esprimerle concretamente all'interno di una *story card*.
Bisogna precisare che i servizi di conversione *speech-to-text* (da discorso a testo), allo stato attuale, non sono ottimizzati per comprendere correttamente espressioni contenenti termini in lingue diverse. È necessario, dunque, che l'assistente vocale sia configurabile per comprendere la lingua parlata dall'utente, fermo restando che, data la natura del linguaggio Swift e della quasi totalità dei linguaggi di programmazione in uso, l'utilizzo di una lingua differente dall'inglese è fortemente sconsigliato.

Programmazione tramite voce

Johnny apre l'applicazione e clicca sul pulsante con l'icona del microfono per iniziare la sua sessione di programmazione e ciò che egli detta sarà trascritto nell'area di testo.

Supponendo che Johnny sia inglese e che il suo dispositivo sia configurato sulla lingua inglese, se Johnny pronuncia la parola "plus" all'interno del codice sarà riportato il segno "+".

Johnny vuole scrivere una stringa contenente la parola "plus" senza che questa venga convertita. Pronuncia la parola "type" seguita dal testo della stringa, seguito a sua volta dall'espressione "in string case". Il testo viene posto tra virgolette e i caratteri speciali al suo interno non vengono convertiti.

Johnny ha bisogno di definire una variabile con il nome "plus". Johnny pronuncia "variable literally plus". Il convertitore traduce questa espressione in "var plus", convertendo la parola "variable" in "var" e trascrivendo la parola "plus" letteralmente. Se Johnny volesse trascrivere letteralmente più di una parola dovrebbe pronunciare l'espressione "type", seguita dal testo, seguita dall'espressione "literally".

Johnny ha bisogno di scrivere il nome composto di una variabile "my value" nella specifica notazione *camel case*. Johnny pronuncia l'espressione "type my value in camel case" e il convertitore la traduce in "myValue".

Nel corso della sua sessione di programmazione Johnny commette un errore. Pronunciando "undo" annulla l'ultima parola che ha dettata.

Accorgendosi di non aver sbagliato, Johnny pronuncia "redo" per recuperare le modifiche annullate.

Dopo un po' la confusione assale Johnny, il quale non è più convinto del codice che ha scritto ed egli decide di pronunciare "clear" per cancellare tutte le parole della riga su cui sta scrivendo e ricominciare da capo.

Quando ha finito di programmare una riga di codice, Johnny va a capo alla riga successiva pronunciando "new line".

Figura 3 - Story Card sulla programmazione tramite voce.

Dal racconto della *story card* è possibile ricavare un elenco di comandi, pronunciabili in fase di programmazione, che consentono la manipolazione del testo per soddisfare una specifica esigenza. Di seguito è riportato un elenco di tali comandi:

- **Type**: consente di trascrivere il testo in una specifica notazione[12] evitando che questo sia sottoposto ad un processo di conversione delle *keywords*;
- **Literally**: consente di trascrivere una parola senza che questa sia convertita. Può anche esprimere la notazione "letterale" se accompagna il comando *Type*;
- **Undo**: annulla l'ultima parola pronunciata e la cancella dall'area di testo;
- **Redo**: annulla l'ultimo comando *Undo* eseguito. È valido solo se pronunciato dopo il comando *Undo*;
- **Clear**: cancella tutta la riga nell'area di testo annullando tutto ciò che è stato pronunciato fino a quel momento.

Poiché ognuno di questi comandi agisce in maniera differente sul testo, è possibile programmare un task per la realizzazione di ognuno di essi. Si considerano inoltre le seguenti funzionalità:

- autoconfigurazione della lingua dell'assistente vocale;
- conversione delle parole chiave.

Si precisa che il codice dettato tramite il supporto dell'assistente potrebbe essere anche direttamente inserito mediante interazione manuale con l'area di testo ad esso dedicata.

Notazione	Formato
Camel Case	camelCase
Kebab Case	kebab-case
Pascal Case	PascalCase
Sponge Case	SpOnGeCaSe
String Case	"string case"
Title Case	Title Case
Upper Case	UPPER CASE

Tabella - Notazioni.

2.5.3 Esecuzione del codice

La compilazione e l'esecuzione sono aspetti essenziali per il programmatore. In un'ambiente di programmazione, queste due funzionalità consentono di testare il codice e di riscontrarne a schermo il corretto funzionamento.

È reso gratuitamente disponibile, con le dovute limitazioni, il servizio del compilatore online paiza.io. Esso è dotato di una API, ovvero un'interfaccia, e comunica con le applicazioni mediante connessione *http*. Il protocollo applicativo *hypertext transfer protocol* (*http*), consente la comunicazione in rete e si basa sul paradigma richiesta-risposta. Nel caso del presente lavoro di tesi, il server (paiza.io)

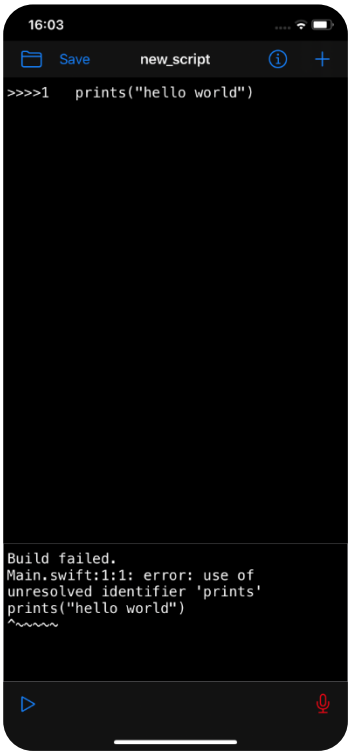

Figura 4 - Esempio di errore di compilazione.

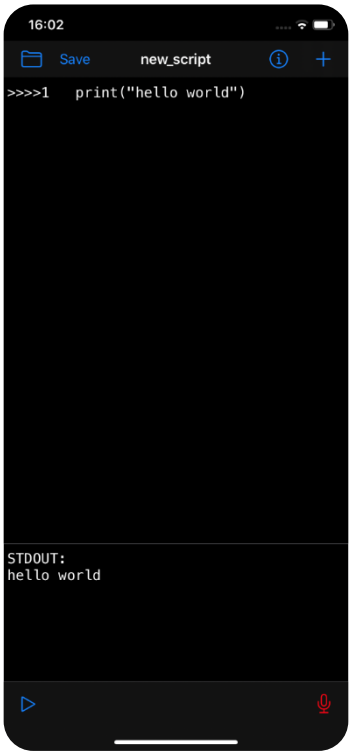

Figura 5 - Esempio di corretta esecuzione.

[12] Le notazioni sono riportate nella Tabella 1

riceve una richiesta dal client (Mobile SAND) e risponde con un messaggio contenente il risultato dell'esecuzione del codice.

Esecuzione del codice
Johnny ha appena terminato la sua sessione di programmazione e vuol testare il codice prodotto, quindi clicca il pulsante *play* di esecuzione. Dopo qualche secondo sul suo schermo, nell'area di testo predisposta, compare un messaggio di errore che riporta sommarie indicazioni sulla causa dell'errore. In una manciata di secondi Johnny individua lo sbaglio commesso e lo corregge la parola funzione "print" che aveva erroneamente pronunciato "prints". Dopo qualche secondo, riceve un messaggio di corretta esecuzione che riporta l'*output* dell'esecuzione del programma: hello world.

Figura 6 - Story Card sull'esecuzione del codice

Al termine di questa fase di progettazione, l'unico task emerso è la realizzazione di un sistema predisposto all'invio ed alla ricezione di messaggi http.

2.5.4 Controllo dell'applicazione tramite voce

L'aspetto fondamentale dell'utilizzo della presente applicazione è l'accessibilità. Allo scopo di rendere l'ambiente più praticabile si vuole estendere il dominio delle esigenze soddisfacibili tramite assistente vocale. A tal proposito si desidera rendere l'applicativo quanto più navigabile possibile mediante l'uso esclusivo dei comandi vocali, implementando un sistema di interpretazione in tempo reale. Il discorso dell'utente deve necessariamente essere processato, ed i comandi in esso rilevati, eseguiti. Si definisce quindi, ricorrendo ancora alla pratica del *brainstorming*, un insieme di comandi ed il corrispettivo comportamento. È ulteriormente richiesto l'immagazzinamento di un insieme di corrispondenze tra comandi vocali ed espressioni del linguaggio naturale.
Attenendosi al principio di "risposta al cambiamento", ci si premura di strutturare, in fase implementativa, la soluzione, in maniera tale da predisporla all'aggiunta di nuovi comandi legati a nuove funzionalità. Al fine di eludere eventuali ambiguità tra comandi vocali ed espressioni del codice, si definiscono separatamente le sessioni di controllo vocale dell'app e di programmazione. Viene, dunque, opportunamente definito un comando che stabilisca il passaggio da una fase all'altra.

Controllo dell'applicazione tramite voce

Johnny inizia a lavorare cliccando il pulsante del microfono sull'applicazione.

Pronunciando il comando "write" comincia a programmare, e detta la riga print("hello world") all'assistente vocale. Al termine dice "stop" ed interrompe la sessione audio.

Dopo aver scritto una decina di righe di codice Johnny vorrebbe ricopiare il comando di stampa che aveva precedentemente dettato. Per farlo sposta il suo cursore virtuale, che tiene traccia dell'ultima riga scritta, alla riga numero 1 pronunciando "line one".

A questo punto Johnny copia nella *clipboard* del dispositivo il testo della riga 1 con il comando "copy", sposta il cursore sulla riga 10 tramite il comando "line ten" e, pronunciando "paste", incolla il contenuto sotto la decima riga, quindi alla riga 11.

Il suo scopo, in realtà, era quello di sostituire il testo nella riga con il contenuto copiato. Per farlo realizza di doversi spostare sulla riga 10 tramite l'apposito comando e pronunciare "delete". La riga 10 a questo punto è cancellata e la riga successiva scalata al suo posto.

A questo punto Johnny impartisce il comando "run" per eseguire l'applicazione ma riceve un messaggio di errore. Non riuscendo ad individuare lo sbaglio chiama sua moglie che usa Mobile SAND già da un po' di tempo ed è una programmatrice più esperta.

La signora Mary, nonostante il suo handicap visivo, che le rende quasi impossibile la lettura, riesce facilmente ad individuare l'errore, muovendosi nell'applicativo tramite il comando "line" e chiedendo all'assistente vocale di leggerle il codice di ogni riga tramite il comando "read".

Accorgendosi che il problema risiedeva in una variabile richiamata prima della sua dichiarazione, Mary sposta il contenuto della riga al posto giusto, tagliando il contenuto della riga in questione con il comando "cut", e posizionandola al posto giusto pronunciando "paste".

Al termine si assicura che l'esecuzione vada a buon fine tramite il comando di esecuzione ed informa il marito sulla causa dell'errore.

Figura 7 - Story Card sul controllo dell'applicazione tramite voce

Dal racconto della *story card*, così come per la funzionalità di programmazione a voce, emergono una serie di comandi. È possibile isolare e suddividere la realizzazione di ognuno dei singoli comandi in uno specifico task.
Di seguito ne è riportato un listato corredato di una descrizione delle loro funzioni:

- **Write**: avvia la fase di programmazione;
- **Stop**: termina la sessione audio;
- **Line**: seguito da un numero intero sposta il cursore alla riga indicizzata a quel numero;
- **Cut**: cancella una riga e ne copia il contenuto nella *clipboard*;
- **Copy**: copia nella *clipboard* il testo sulla riga dov'è posizionato il cursore;
- **Paste**: incolla il contenuto copiato nella *clipboard* tra la riga indicata dal cursore e quella successiva, se presente;
- **Delete**: cancella una riga;
- **Run**: esegue il codice scritto e riporta l'*output* nell'area di testo dello *stream*;
- **Read**: legge la riga indicata dal cursore.

2.5.5 Accesso alla guida

Nessun utente che abbia appena scaricato una nuova applicazione risulta essere già in grado di usarla. Sebbene una buona app offra sempre delle funzionalità intuitive e di immediato utilizzo, esistono alcuni passaggi nei quali è necessario guidare l'utente. A tale scopo occorre fornire un sistema di informazioni che sia immediatamente accessibile. Perché ciò avvenga, queste indicazioni devono essere fornite automaticamente dal sistema quando necessarie, oppure devono poter essere raggiunte intuitivamente tramite un comando. Nel caso del presente ambiente, tale funzionalità è attivabile mediante il *tap* di un pulsante "info", rappresentato dalla caratteristica icona con la lettera "i". Al tocco del pulsante viene illustrato, nelle aree di testo, un insieme di informazioni che guidino all'utilizzo dell'applicativo.

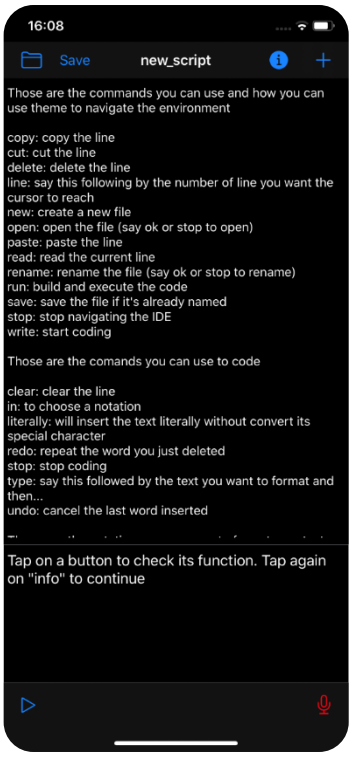

Figura 8 - Informazioni per l'uso.

Accesso alla guida
Johnny apre l'applicazione Mobile SAND per la prima volta, e vorrebbe imparare a programmare tramite l'utilizzo della voce. Nella schermata principale cerca informazioni sul funzionamento dell'app ed individua immediatamente un pulsante con una "i" nella barra superiore. Quando interagisce con il *button* esso diventa completamente blu e scopre che vengono riportate, nella area di testo più grande, un elenco di comandi vocali ed indicazioni su come usarli. Vengono anche descritti i modi per pronunciare i caratteri speciali e gli operatori Swift. Nell'area di testo più piccola legge un messaggio che gli suggerisce di cliccare su uno dei pulsanti per scoprire la sua funzionalità. Cliccando sul pulsante del microfono, Johnny legge che tramite esso è possibile iniziare a programmare tramite voce e, studiando il manuale, inizia a programmare.

Figura 9 - Story Card sull'accesso alla guida

I due task emersi in questa fase sono:

- Caricamento delle informazioni da file;
- Applicazione del contenuto sulle aree di testo.

2.5.6 Organizzazione dei file

Un programmatore che voglia salvare il lavoro svolto ha bisogno di un sistema di documenti che custodisca il contenuto da esso prodotto. In questo caso, i *files* conservati dall'applicativo sono degli *scripts* in formato *swift*. Per consentire all'utente di conservali occorre fornirgli una nuova schermata dedicata alla gestione dei suoi documenti. All'interno di questa interfaccia utente, viene resa possibile la cancellazione e l'apertura dei *files*. I comandi di salvataggio e di cancellazione vengono invece posti nella schermata principale, per consentire un accesso più rapido a tali funzioni. Vengono, inoltre, aggiunti due nuovi comandi vocali che consentano:

- La creazione di un *file*;
- L'apertura di un *file*;
- Il salvataggio di un *file*;
- Modifica del nome di un *file*.

I documenti vengono salvati all'interno del bundle [13] dell'applicazione.

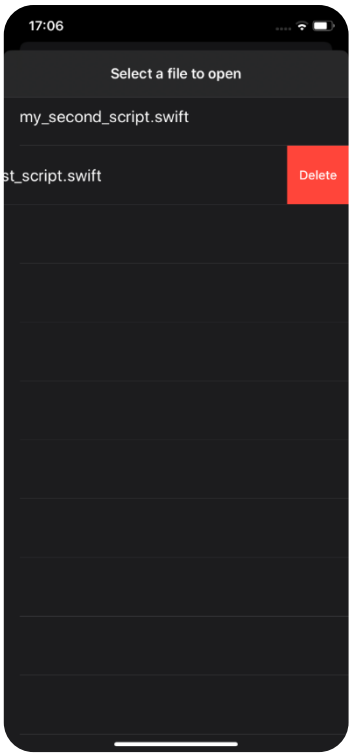

Figura 10 - Schermata dei documenti.

Organizzazione dei file
Dopo una sessione di programmazione, Johnny ha intenzione di passare alla realizzazione del prossimo script. Pronunciando "new" riceve un *popup* che lo avvisa di salvare le modifiche effettuate. Johnny, ricordatosi di non aver salvato il suo lavoro, clicca il tasto di annullamento, e poi tramite l'assistente vocale pronuncia "save". Un messaggio nell'area di testo gli ricorda che per salvare occorre prima che il file abbia un nome. Johnny, ancora, ricorda di aver dimenticato di dare un nome al suo *script* ed assegna, pronunciando "rename my script" il comando. Al termine pronuncia "okay" e lo *script* viene rinominato in "my_script" come riportato sulla barra superiore. Pronunciando "new" e confermando la sua scelta Johnny crea un nuovo script e comincia a lavorare. Dopo un po' Johnny decide di ritornare sul suo lavoro precedente ed impartisce il comando "open my script" per aprire lo script precedentemente salvato.

Figura 11 - Story Card sull'organizzazione dei file

I task che emergono da questa fase di progettazione consistono nella realizzazione dei comandi precedentemente descritti e le seguenti funzioni:

- Elenco dei *file*;
- Cancellazione di un *file* dall'elenco.

[13] Un *bundle* è uno spazio di archiviazione dedicato alle risorse dell'applicazione.

3 IMPLEMENTAZIONE

L'obiettivo di questo capitolo è quello di descrivere gli aspetti implementativi del progetto Mobile SAND, analizzando le classi ed i *packages* relativi all'interfaccia ed alle singole funzionalità.

3.1 ARCHITETTURA DEL SOFTWARE

La struttura finale di Mobile SAND è basata su due cartelle fondamentali:

- **Sources**: contiene l'insieme dei *packages* che realizzano le funzionalità ed il comportamento dell'interfaccia;
- **Assets**: contiene tutte le risorse testuali e grafiche impiegate nell'applicazione.

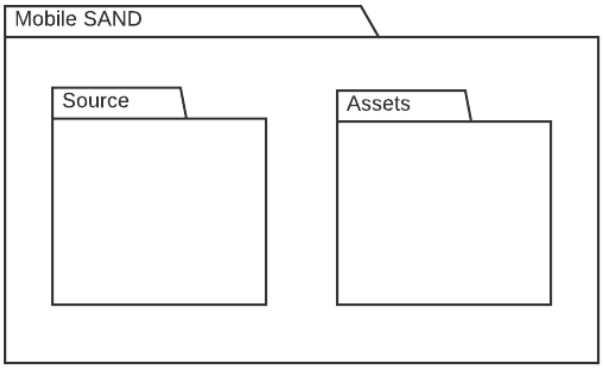

Figura 12 - Organizzazione di Mobile SAND

3.1.1 Sources

L'architettura della cartella *Sources* è organizzata secondo il diagramma rappresentato in Figura 13.

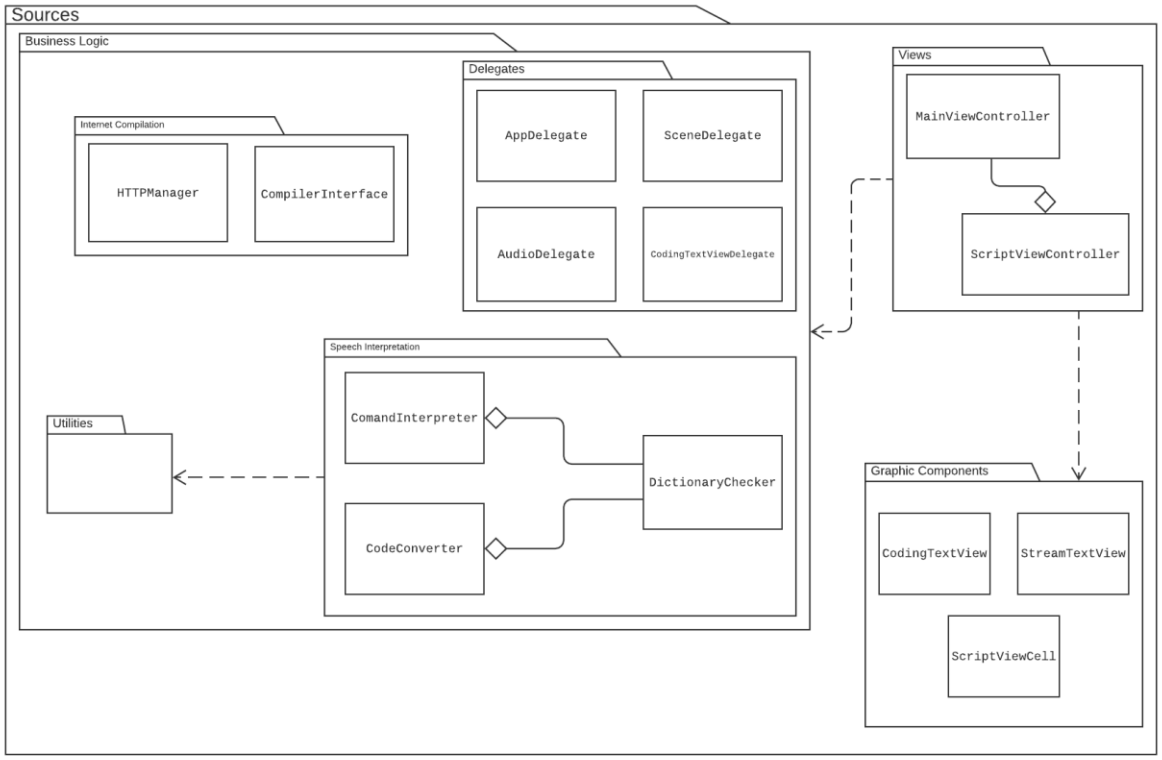

Figura 13 - Diagramma di Sources

Di seguito si riporta un elenco gerarchizzato della struttura della cartella *Sources*, e riassuntivo dei singoli elementi in essa contenuti:

- **Views**: *package* responsabile della gestione delle schermate dell'applicazione. Contiene:
  - **`MainViewController`**: classe responsabile della gestione della schermata principale;
  - **`ScriptViewController`**: classe responsabile della gestione della schermata per la gestione dei *files*.
- **Business Logic**: *package* responsabile dell'implementazione della logica applicativa del sistema. Contiene:
  - **Internet Compilation**: *package* responsabile della comunicazione con l'API del compilatore online. Contiene:
    - **`HTTPManager`**: classe che realizza un client per la comunicazione *http* con il server di paiza.io.
    - **`CompilerInterface`**: classe che realizza un'interfaccia per l'invio di richieste di compilazione.
  - **Utilities**: contiene un insieme di *scripts* e classi di utilità che forniscono servizi generici.
  - **Speech Interpretation**: *package* responsabile di convertire il parlato in codice ed interpretarne i comandi vocali. Contiene:
    - **`DictionaryChecker`**: classe responsabile del caricamento da file di tutti i dizionari e le risorse testuali necessarie al funzionamento dell'app;
    - **`CodeConverter`**: classe responsabile della conversione del parlato in codice e dell'esecuzione di comandi di programmazione;
    - **`CommandInterpreter`**: classe responsabile dell'interpretazione del parlato e dell'esecuzione di comandi di controllo in tempo reale.
  - **Delegates**: *package* responsabile della gestione delle classi che inviano notifiche.
- **Graphic Components**: *package* responsabile della realizzazione dell'interfaccia grafica. Contiene:
  - **`CodingTextView`**: classe responsabile della gestione dell'*editor* di programmazione;
  - **`StreamTextView`**: classe responsabile della gestione dell'area di testo della *stream output*;
  - **`ScriptViewCell`**: classe responsabile della definizione della singola cella della `UITableView` della schermata per l'organizzazione dei *files*.

3.1.2 Assets

L'architettura della cartella Resources è illustrata nel diagramma in Figura 14.

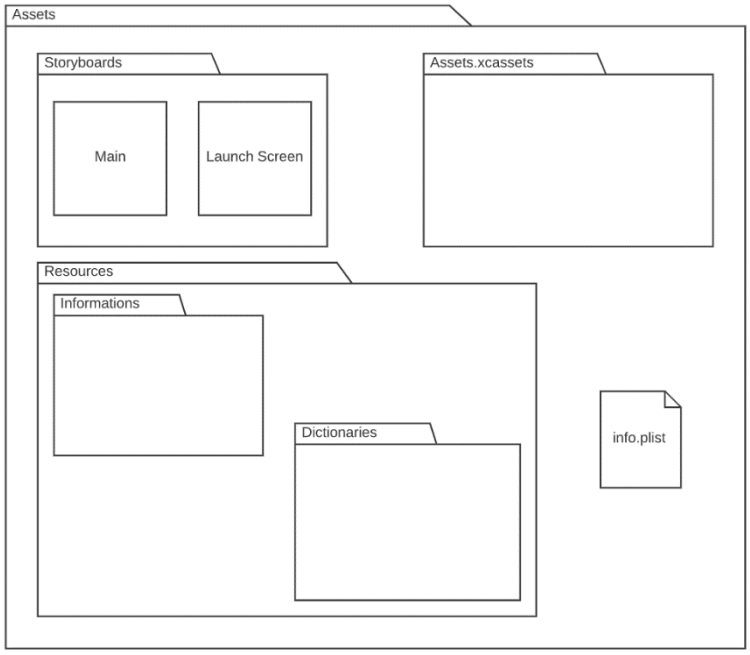

Figura 14 - Diagramma di Resources

La struttura della cartella *Resources* è organizzata nel mondo seguente:

- **Storyboards**: conserva i *files* formato storyboards che definiscono le schermate e le transizioni tra di esse.
  - **`Main`**: responsabile della definizione della schermata principale e della schermata di organizzazione dei *files*;
  - **`LaunchScreen`**: schermata temporanea di caricamento.
- **Assets.xcassets**: è lo storage predefinito di Xcode per conservare le risorse grafiche.
- **Resources**:
  - **Informations**: conserva i *files* testuali con le informazioni in lingua per il manuale ed il sistema di info;
  - **Dictionaries**: conserva i *files* testuali contenenti le informazioni per l'interpretazione del parlato.
- **`info.plist`**: *file* che definisce le impostazioni e le informazioni dell'app.

3.2 INTERFACCIA E STORYBOARDS

La funzionalità di interfacciamento con l'utente richiede la realizzazione di un modello grafico per la schermata principale. La definizione del *mockup* avviene all'interno del file *main.storyboard* contenuto nella cartella *Storyboards* inclusa in *Assets*. Uno *storyboard* è un *file* che definisce un insieme di schermate (*views*) e le entità (*segue*s) che consentono le transizioni tra di esse.
La *view* che definisce il modello rappresentato in Figura 1 è la *Main View*, che si compone dell'insieme di elementi grafici precedentemente descritti nel paragrafo 2.4.

In essa sono anche definiti un insieme di costrizioni (*constraints*) utili a rendere il layout adattabile agli schermi di differenti dispositivi.
Alla *Main View* è associata la classe `MainViewController` che estende `UIViewController`, la classe responsabile della gestione delle interazioni con la singola schermata. In essa sono contenuti:

- i riferimenti alle istanze dei componenti grafici;
- i metodi legati alle azioni dell'interfaccia, come ad esempio la pressione di un pulsante;
- una collezione di variabili e metodi necessari al funzionamento dell'applicazione.

`MainViewController` è una classe inizialmente vuota che si popola, con l'aggiunta di funzionalità, di nuovi metodi e variabili necessarie alla loro gestione. Al caricamento della schermata principale viene eseguito il metodo `viewDidLoad` che inizializza tutte le variabili non opzionali[14] e costruisce il controllore. L'unico blocco di variabili inizialmente definito è quello dei *references* ai componenti grafici.
Nel metodo di caricamento vengono inizializzate le due aree di testo mediante l'istanziazione di due oggetti delle classi `CodingTextView` e `StreamTextView`.
`CodingTextView`, contenuta nel pacchetto *Graphic Components*, implementa, nella `UITextView`[15], le funzioni principali di un *editor* per la programmazione.
In esso sono definite delle *property observers* [16] e delle *computated property* [17] interdipendenti che consentono di conservare il testo scritto dall'utente in diverse forme. Esse fornisce anche un cursore che consente di navigarvi all'interno.
`StreamTextView` è invece un'area predefinita, su cui l'utente non è abilitato ad effettuare operazioni di scrittura.

3.3 IMPLEMENTAZIONE DELLE FUNZIONALITÀ

L'obiettivo di questa sezione è descrivere l'implementazione delle singole funzionalità, analizzando singolarmente le classi che realizzano il loro comportamento.

3.3.1 Dettatura

La realizzazione della funzione di dettatura viene concretizzata dalla classe `MainViewController` tramite due metodi: `startRecording` e `closeAudioSession`. Entrambi questi metodi, i cui scopi sono rispettivamente l'avvio e la chiusura di una sessione di registrazione, sono, a loro volta, chiamati da un metodo `micButtonAction`, eseguito alla pressione del pulsante del microfono. Per la programmazione del loro comportamento sono state impiegate classi predefinite offerte da due *frameworks*:

- `AVFoundation`, che definisce le classi necessarie alla gestione dei nodi e della sessione di registrazione;
- `Speech`, che definisce un insieme di classi utili alla traduzione del parlato in testo.

[14] Una variabile si definisce "*optional*" se può assumere un valore nullo.
[15] `UITextView` è la classe predefinita che implementa il comportamento di un'area di testo.
[16] Una property observer è una variabile che conserva un valore ed esegue codice in caso di assegnazione.
[17] Una computed property è una variabile che non conserva un valore ma lo calcola quando essa viene chiamata, e può eseguire codice in caso di assegnazione.

Si riassume di seguito l'implementazione dei due metodi precedentemente citati e delle operazioni che essi eseguono:

- **`startRecording`**:
    1. Sono inizializzate le istanze delle classi `AVAudioSession` e `AVAudioEngine` per l'avvio della sessione e del nodo di registrazione;
    2. Viene inizializzato un `SFSpeechRecognizer` a cui viene assegnata una richiesta di riconoscimento vocale. Essa si concretizza in un *task* di traduzione del parlato in testo, affidato, se possibile, al server Apple, o altrimenti al processo locale competente. Inoltre, è definita una funzione da eseguire al ricevimento di una notifica di trascrizione completata. Nel caso del presente applicativo, essa consiste nell'interpretazione del risultato di tale trascrizione, come comandi di controllo e di programmazione, ed è contenuta nel *folded code*[18] (identificato dai tre puntini sospensivi) in Figura 15.
- **`closeAudioSession`**: Il nodo di registrazione viene disattivato, la sessione audio viene terminata ed il *task* di riconoscimento vocale viene revocato.

Altro metodo, già citato in precedenza, definito nella classe `MainViewController`, è `micButtonAction`, associato al pulsante del microfono, che è essenzialmente un interruttore per la sessione di registrazione. Esso la avvia se è spenta e la termina se è già attiva.

```
//Assegna il task
speechRecognitionTask = speechRecognizer?.recognitionTask(with: speechRecognitionRequest)
{
    (result, error) in

    var isFinal = false //isFinal è inizialmelmente false
    let transcription = result?.bestTranscription.formattedString //Memorizza la trascrizione

    //Esegue tutte le operazioni di interpretazione del risultato in tempo reale
    if result != nil
    {...}

    //Interrompe la registrazione in caso di errore o di tempo scaduto
    if error != nil || isFinal
    {
        self.closeAudioSession() //Interrompe la sessione audio
    }
}
```

Figura 15 - Codice di assegnazione di un task di riconoscimento vocale

3.3.2 Programmazione tramite voce

La funzionalità di programmazione vocale viene implementata nelle classi `CodeConverter` e `DictionaryChecker`, contenute all'interno del *package Speech Interpretation*. Viene di seguito descritto il loro ruolo nella realizzazione di tale funzionalità.

`DictionaryChecker` è una classe composta prevalentemente da enumerazioni e componenti del tipo personalizzato `Dictionary`[19], e fornisce un metodo pubblico `convertTerms` per la traduzione delle espressioni del parlato. Essa conserva tutte le corrispondenze che consentono di convertire le espressioni in operatori, caratteri speciali e comandi di programmazione. Per rendere possibile tale operazione, `DictionaryChecker` carica, in fase di inizializzazione, tutte le informazioni necessarie dai *files* testuali contenuti nella cartella *Resources*. Per fare ciò, costruisce i *paths* basandosi sulla lingua predefinita di sistema, garantendo la possibilità di programmare in diverse lingue.

`CodeConverter` è la classe responsabile della conversione del parlato in codice e si avvale dei servizi del `DictionaryChecker` per adempiere a tale compito.

[18] È un frammento di codice "ripiegato", ovvero nascosto al programmatore dall'ambiente di programmazione.
[19] Dizionario che ha una stringa come chiave ed un vettore di stringhe come valore.

Il processo di traduzione, realizzato dal metodo pubblico `convert`, si suddivide nella seguente serie ordinata di passi:

1. Viene eseguita la traduzione dei comandi e delle notazioni nella loro stringa identificativa;
2. Vengono eseguiti i comandi e convertite le parole chiave del linguaggio Swift;
3. La stringa viene assemblata e restituita in *output*.

Il cuore di questa trascrizione è la conversione di gruppi di parole in *keywords* di Swift e caratteri speciali. Anche tutti i comandi delineati nella fase di progettazione descritta nel paragrafo 2.5.2, sono comandi di manipolazione del testo, che non devono necessariamente essere eseguiti in tempo reale. Essi possono essere "compilati" in una fase di conversione della stringa, rinnovabile ad ogni nuova notifica di trascrizione del riconoscitore vocale.

In generale, `CodeConverter` richiede il servizio di "conversione dei termini" ad un'istanza di `DictionaryChecker` nel corso dell'intera esecuzione del metodo `convert`. Tuttavia, è necessario assumere che alcuni comandi debbano avere la precedenza su altri, come il comando di *undo* sul comando *type*, che deve annullarlo prima che quest'ultimo venga eseguito. È, dunque, necessario organizzare i passaggi di conversione secondo il seguente ordine di priorità:

1. Redo;
2. Undo;
3. Clear;
4. Type;
5. Literally.

Inoltre, alcune sezioni del testo, come per esempio quelle sottoposte ai comandi *type* e *literally*, vengono contrassegnate da due caratteri speciali di limitazione ed ignorate dalle operazioni di conversione gerarchicamente inferiori. Questo passaggio permette alle stringhe "letterali" di non subire ulteriori conversioni.

Al termine di questi passaggi, viene restituita una *tupla*[20], composta da una stringa di codice Swift eseguibile e da un *flag* che notifica, all'occorrenza, il termine della fase di programmazione.

È presentata in Figura 16 l'implementazione del metodo di conversione precedentemente descritto.

```swift
//Effettua la conversione finale
func convert(_ string: String) -> (convertedString: String, closeAudioSession: Bool)
{
    var words = stringToWords(introduceWhiteBetweenSpecialChar(string)) //Conserva la stringa come array
    //1. Converte i comandi di base nel linguaggio del sistema
    words = dictionaryChecker.convertTerms(words: words, in: dictionaryChecker.statementsDictionary)
    //2. Converte i comandi personalizzati nel linguaggio del sistema
    words = dictionaryChecker.convertTerms(words: words, in: dictionaryChecker.customsDictionary)
    //3. Converte i nomi delle notazioni
    words = dictionaryChecker.convertTerms(words: words, in: dictionaryChecker.notationsDictionary)
    //4. Esegue tutti i redo
    words = executeRedo(words)
    //5. Esegue tutti gli undo
    words = executeUndo(words)
    //6. Esegue tutte le clear
    words = executeClear(words)
    //7. Esegue tutte le type
    words = executeType(words)
    //8. Esegue tutte le literally
    words = specificExecution(on: words, with: executeLiterally)
    //9. Converte tutti i termini nelle corrispondenti keywords
    words = specificConversion(words: words, in: dictionaryChecker.keywordsDictionary)
    //10. Avvisa di un eventuale comando di interruzione alla fine e lo rimuove
    let closeResult = executeClose(words)
    words = closeResult.convertedWords
    let closeAudioSession = closeResult.closeAudioSession
    //11. Rimuove i marcatori dei comando
    words = removeMarkers(words)
    //12. Assembla la stringa
    let output = removeWhiteBetweenSpecialChar(wordsToString(words))
    return (convertedString: output, closeAudioSession: closeAudioSession)
}
```

Figura 16 - Implementazione del metodo `convert` di `CodeConverter`

Questa funzionalità viene integrata nell'applicativo con il completamento del task di riconoscimento citato nel paragrafo 3.3.1 sulla dettatura. Nello specifico, le operazioni di conversione avvengono all'interno del *folded code* illustrato in Figura 15, ricavando il codice dal parametro *result* (risultato della trascrizione del parlato).

[20] Una *tupla* in Swift è un tipo composto che consente il raggruppamento di un insieme di valori.

3.3.3 Esecuzione del codice

Le funzionalità di compilazione ed esecuzione del codice sono implementate nelle classi `CompilerInterface` e `HTTPManager`, contenute nel pacchetto *Internet Compilation*.
`HTTPManager` è la classe fondamentale che stabilisce la connessione *http* con il server paiza.io e gestisce l'invio delle richieste e la ricezione delle risposte. La comunicazione *http* con l'API di paiza per l'utilizzo del servizio è strutturata nel seguente modo:

1. Il client (`HTTPManager`) invia un messaggio di POST all'URL dedicato alla creazione di una sessione di compilazione, specificando il codice da eseguire nell'URL della richiesta;
2. Il server (paiza.io) risponde con un messaggio contenente l'ID assegnato alla sessione di comunicazione;
3. Il client invia un messaggio di GET specificando nell'URL: lo scopo della richiesta, la API *key* e l'ID della sessione corrispondente;
4. Il server risponde con un messaggio contenente tutte le informazioni sulla compilazione e l'esecuzione del codice.

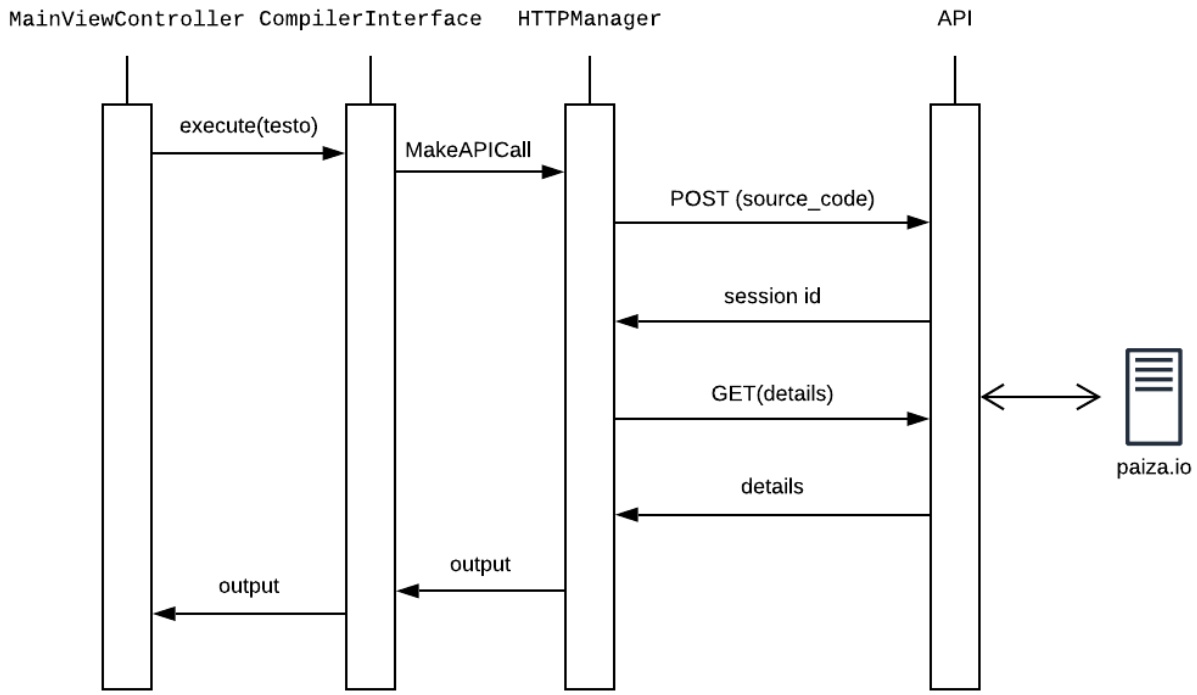

Figura 17 - Protocollo di comunicazione col server paiza.io

Allo scopo di astrarre `MainViewController` dai servizi di rete offerti da `HTTPManager` viene definita una classe di interfaccia, `CompilerInterface`, che fornisce un servizio di esecuzione del codice tramite un unico metodo pubblico: `execute`. La stessa interfaccia istanzia un oggetto di classe `HTTPManager` e si occupa di rappresentare a schermo il risultato dell'operazione di esecuzione del codice. L'esecuzione è avviata alla pressione del pulsante *play*.
Una rappresentazione schematica dell'interno processo di comunicazione descritto in questo paragrafo è illustrata in Figura 17.

3.3.4 Controllo dell'applicazione tramite voce

La classe responsabile dell'interpretazione dei comandi vocali è `CommandInterpreter` del pacchetto *Speech Interpretation*, che si avvale di `DictionaryChecker` per il riconoscimento dei comandi. La realizzazione delle funzionalità di controllo vocale dell'applicazione è un processo affine a quello di programmazione. Tuttavia, mentre il processo di conversione in codice è, a tutti gli effetti, una fase di compilazione, il processo di controllo vocale deve compiere azioni e tener traccia della loro esecuzione in tempo reale. L'interpretazione della trascrizione avviene tramite il metodo `convert`.

La stringa iniziale viene suddivisa in un vettore di stringhe, che viene iterato alla ricerca di comandi vocali da eseguire. Quando `CommandInterpreter` trova un comando, esegue un'azione, generalmente chiamando un metodo di `MainViewController`.

Una problematica riscontrata nella fase di controllo vocale concerne l'esecuzione dei comandi, che, a differenza dell'operazione di conversione in codice, non deve essere ripetuta ad ogni nuova trascrizione ricevuta dal riconoscitore vocale.

Le soluzioni implementabili per far fronte al problema sono le seguenti:

- Eseguire, ad ogni nuova notifica, i comandi identificati nell'*array*, ignorando gli indici delle stringhe già analizzate nelle trascrizioni precedenti;
- Eseguire, ad ogni nuova notifica, i comandi identificati nell'*array*, ignorando un numero di comandi pari a quelli già eseguiti nelle trascrizioni precedenti.

Poiché `SpeechRecognizer` invia un numero variabile di notifiche di trascrizione ad ogni nuova parola pronunciata, spesso anche allo scopo di riformulare una traduzione più corretta, capita frequentemente che una trascrizione possa essere più corta della precedente. Al fine di evitare problemi di inconsistenza con l'indicizzazione dell'*array*, si opta per la seconda soluzione. Quando `CommandInterpreter` riceve un comando di interruzione della fase di controllo o di avvio della fase di programmazione, restituisce, rispettivamente, un segnale di chiusura oppure testo da convertire in codice.

Poiché la maggior parte delle azioni relative ai comandi vocali sono già usufruibili tramite interazione diretta, la loro attuazione consiste nella chiamata di funzioni già definite in `MainViewController` o classi di utilità.

Un modello finale del processo completo di interpretazione vocale è illustrato in Figura 18.

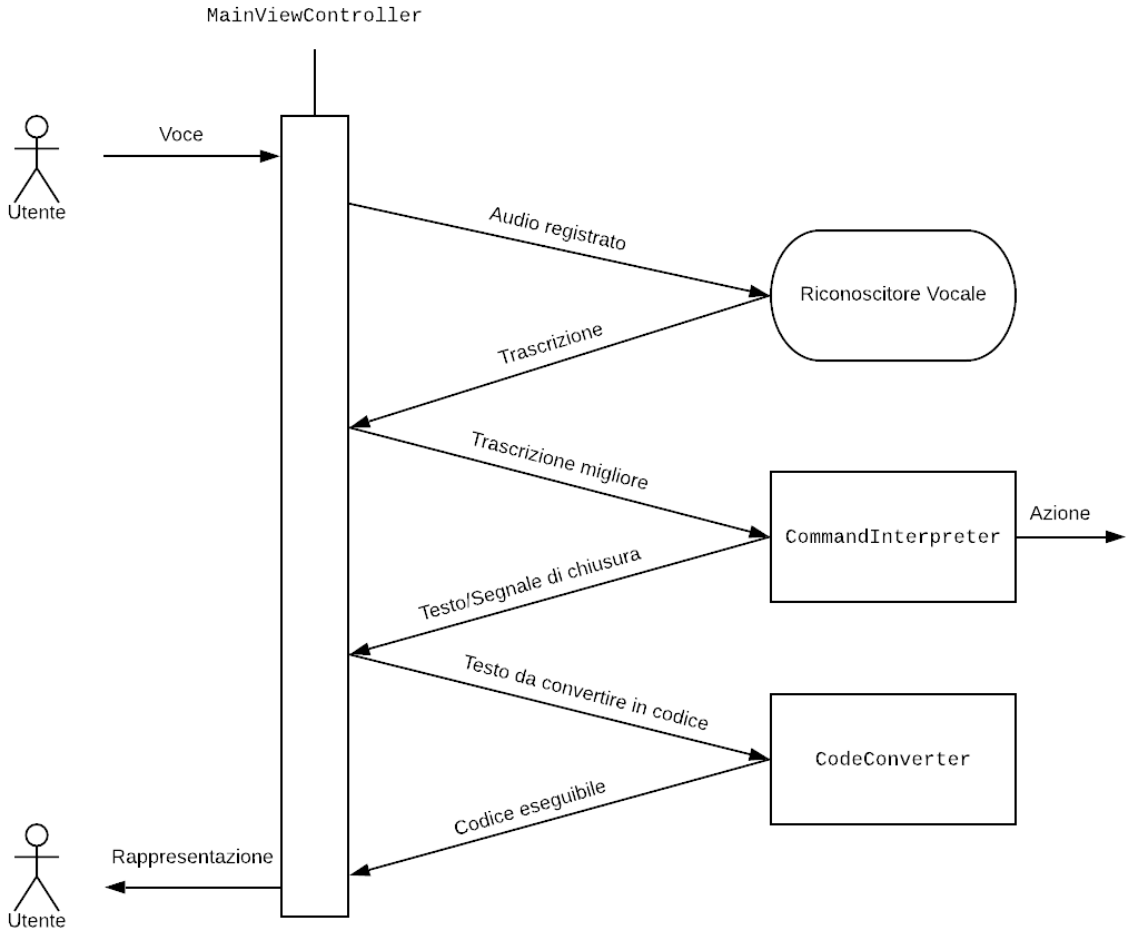

Figura 18 - Processo di interpretazione vocale

3.3.5 Accesso alla guida

La funzionalità di accesso alla guida è implementata nelle classi `MainViewController`, `CodingTextView` e `StreamTextView` dei packages *View*, e *Graphic Components*. L'esposizione delle informazioni consiste nell'attivazione, tramite il *button info*, di una fase di "rappresentazione della guida", determinata da una variabile booleana `isShowingInfo`.

`CodingTextView` conserva informazioni su due stringhe: il codice eseguibile e le info dell'app. Il passaggio da una rappresentazione all'altra avviene mediante un metodo `triggerInfo`, definito all'interno della classe. `MainViewController` richiede tale servizio passando come parametro un testo delle info, costruito caricando le informazioni dalla cartella *Informations* con l'ausilio di `DictionaryChecker`. Viene quindi illustrata una rapida introduzione all'uso dell'applicativo, corredata da un insieme di parole chiave e modi di pronunciarle nella lingua impostata.

`StreamTextView` segue un comportamento analogo ed implementa lo stesso metodo `triggerInfo` per la rappresentazione delle info. A differenza di `CodingTextView`, essa

non mostra un manuale ma illustra, alla pressione di un pulsante della schermata, le sue funzionalità. Anche le informazioni sui *button* vengono caricate tramite `DictionaryChecker`.

3.3.6 Organizzazione dei files

La funzionalità di organizzazione dei *files* salvati dall'utente è implementata in `MainViewController` e `ScriptViewController`. La realizzazione della funzionalità è suddivisa, come riportato nel paragrafo 2.5.6, in diversi task, ognuno dei quali identifica l'implementazione di una sottofunzionalità.
In `MainViewController` vengono definiti i metodi di accesso al file system, tramite l'ausilio di funzioni definite nello *script* di utilità, `FileUtilities`.
Tutte le azioni che richiedono un'interazione più rapida dell'utente vengono implementate tramite pulsanti della schermata principale e sono:

- Creazione di un nuovo *file*;
- Salvataggio di un *file*;
- Modifica del nome di un *file*.

Le seguenti funzionalità a carattere organizzativo sono invece accessibili da una schermata ad esse dedicata:

- Elenco dei *files*;
- Cancellazione di un *file*;
- Apertura di un *file*.

Tale schermata è una `UITableView`[21] e viene gestita da un `ScriptViewController` che chiama i metodi di gestione dei *files* da un *reference* di `MainViewController` ed inizializza un *array* di URLs degli *scripts* contenuti nel *bundle* dell'applicazione.
Questa nuova schermata è accessibile tramite il pulsante *organize* della schermata principale ed è resa possibile da una *segue* definita in *main.storyboard,* e realizzata dall'*override* del metodo `prepare` del `MainViewController` che la esegue.

3.4 RISORSE

Le risorse utilizzate per il funzionamento del presente applicativo sono esclusivamente documenti di testo contenenti un insieme di associazioni.
La cartella *Resources* viene suddivisa nelle due sottocartelle *Informations* e *Dictionaries* che assolvono alle due funzioni di seguito descritte:

- **Informations**: contiene le corrispondenze tra un elemento e la sua descrizione. Tali associazioni vengono impiegate per la costruzione del manuale all'uso e delle informazioni sui singoli *buttons*, rappresentate alla pressione del pulsante *info*;
- **Dictionaries**: contiene le corrispondenze tra una *keyword*, che può rappresentare un comando od una parola del linguaggio Swift, ed un insieme di espressioni per pronunciarla nella lingua impostata.

Ogni documento viene dunque caricato da `DictionaryChecker` sulla base della lingua predefinita di sistema ed i suoi dati conservati in variabili del tipo `Dictionary` citato nel paragrafo 3.3.2. Nel caso non ci fosse un *file* associato, `DictionaryChecker` prova a caricare i dizionari in lingua inglese o altrimenti lancia un'eccezione.

[21] `UITableView` è il componente che consente la rappresentazione di una lista di elementi

I titoli di tutti i *files* si conformano alla seguente notazione[22]:

- [*Lingua Impostata*]_[*nomeFile*].txt

Ogni riga del contenuto delle risorse deve essere separata dalle altre mediante carattere di ritorno a capo e deve attenersi al seguente protocollo:

- [*Keyword*] - [*Espressione1, Espressione2, …, EspressioneN*]

Il sistema precedentemente descritto getta le basi per la realizzazione della localizzazione in lingua dell'app, nonché per l'aggiunta di nuove funzionalità in maniera rapida ed efficace.

3.5 AGGIUNGERE NUOVE FUNZIONALITÀ

Al fine di perseguire una buona flessibilità del software, il lavoro di implementazione è stato intrapreso tendendo conto della possibilità di aggiunta di nuove funzionalità.
Che si tratti di comandi vocali o costrutti di programmazione, il processo da seguire è approssimativamente lo stesso.
È stato, dunque, realizzato un sistema che faciliti questo passaggio, basandosi su un insieme ordinato di passaggi da rispettare per il perseguimento di tale scopo. Sono di seguito elencati i passi di tale algoritmo:

1. Aggiungere una nuova corrispondenza nel *file* delle risorse associato contenuto in *Dictionaries*;
2. Aggiungere, eventualmente, una descrizione del comando nel *file* delle info contenuto in *Informations*;
3. Aggiungere, se possibile, una nuova voce all'enumerazione relativa al comando o al costrutto, definita in `DictionaryChecker`;
4. Implementare i metodi associati alla funzionalità dove ritenuto più opportuno;
5. Aggiungere un nuovo metodo per l'esecuzione del nuovo comando o per la conversione del nuovo costrutto all'interno di `CommandInterpreter` o `CodeConverter`.

[22] Le espressioni poste in corsivo e tra parentesi quadre variano per ogni documento

4 CONCLUSIONI E SVILUPPI FUTURI

L'obiettivo di questo capitolo è trarre delle conclusioni sul contributo portato dal presente progetto alla soluzione del problema e su possibili sviluppi futuri.

4.1 CONCLUSIONI

Il tema dell'accessibilità è molto attuale ed affrontato con una risolutezza sempre maggiore dalle aziende, in tutti i campi di loro interesse. Le disabilità e le limitazioni, che affliggono una parte non indifferente della popolazione, sono e devono essere coprotagoniste nei processi che portano alla realizzazione di servizi e contenuti usufruibili dal pubblico. Lo scopo del progetto Mobile SAND è quello di portare un incentivo per quelle fasce altrimenti impossibilitate ad approcciare il mondo della programmazione. Sulla scia di altre risorse software, che sperimentano o propongono soluzioni accessibili a tutta la popolazione, Mobile SAND offre, come personale contributo, un ambiente di programmazione nativo per dispositivi *mobile*, utilizzabile mediante un approccio *hands free*. Avvalendosi delle tecnologie Apple, note anche per l'efficacia dei propri strumenti di accessibilità, il presente applicativo vuole portare una soluzione efficace ed immediata, che sia quanto più confortevole possibile, sia per chi è affetto da una disabilità, sia per chi crede nella comodità degli assistenti vocali e nel loro sviluppo. In particolare, Mobile SAND, fornisce un ambiente semplice ma completo di tutte le funzionalità necessarie all'apprendimento del linguaggio Swift e, più in generale, del mondo della programmazione.

4.2 SVILUPPI FUTURI

Alcuni dei possibili sviluppi futuri previsti per l'app Mobile SAND sono i seguenti:

- Migliorare l'interfaccia grafica;
- Arricchire il sistema di programmazione tramite voce;
- Introdurre la possibilità di gestire progetti;
- Integrare un sistema di *versioning and continuous integration*;
- Estendere le funzionalità dell'ambiente di programmazione;
- Migliorare il sistema di localizzazione;
- Agevolare l'interfacciamento con l'assistente vocale;

INDICE DELLE FIGURE

BIBLIOGRAFIA

Aho, A. V., Sethi, R., & Ullman, J. D. (2006). *Compilatori. Principi, tecniche e strumenti*. Milano: Pearson.

Apple Inc. (2020, Marzo 4). *Apple Store Review Guideliens*. Tratto da Apple Developer: https://developer.apple.com/app-store/review/guidelines/#design

Apple Inc. (2020). *Human Interface Guidelines*. Tratto da Apple Developer: https://developer.apple.com/design/human-interface-guidelines/

Apple Inc. (2020). *Technologies*. Tratto da Apple Developer: https://developer.apple.com/documentation/technologies

CDC. (2018). *Disability and Functioning*. Tratto da Centers for Disease Control and Prevention: https://www.cdc.gov/nchs/fastats/disability.htm

eurostat. (2019, Ottobre). *Functional and activity limitations statistics*. Tratto da Eurostat: https://ec.europa.eu/eurostat/statistics-explained/index.php/Functional_and_activity_limitations_statistics#Functional_and_activity_limitations

Freepik Company S.L. (s.d.). *Freepik*. Tratto da Freepik: https://www.freepik.com/

Kurose, J. F., & Ross, K. W. (2013). *Reti di calcolatori e Internet*. Milano-Torino: Pearson Italia.

Manifesto per lo Sviluppo Agile del Software. (2001). Tratto da Agile Manifesto: https://agilemanifesto.org/iso/it/manifesto.html

Sommerville, I. (2017). Ingegneria del Software. In I. Sommerville, *Ingegneria del Software* (p. 61-92). Milano - Torino: Pearson Italia.

WHO. (2011, Dicembre 13). *World report on disability*. Tratto da World Health Organization: https://www.who.int/publications/i/item/world-report-on-disability