Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica

A.a. 2021/2022
Sessione di laurea Ottobre 2022

UnPezzoAllaVolta: interfaccia multimodale per guidare gli addetti al prelievo in magazzino

Relatori:

prof. Luigi De Russis

Riccardo Sioli

Candidato:

Antonio Vespa

Ringraziamenti

Credo che questa tesi sia la degna conclusione del mio percorso universitario, frutto delle tante conoscenze apprese in questi anni, ma anche altrettante ansie, paure, passione. Ripensando a questo progetto e alle sue mille difficoltà, la mia gratitudine va al prof. Luigi De Russis, che mi ha sempre supportato con precisione, professionalità e gentilezza, e al mio referente aziendale Riccardo Sioli che, non solo mi ha dato l'opportunità di lavorare a un progetto molto interessante, ma mi ha anche sempre fatto sentire parte del team, accogliendomi con entusiasmo. Invece, per tutte quelle persone che in questi anni mi hanno sopportato e supportato, chi con la vicinanza, chi con un gesto d'affetto o anche con parole sincere, vi dedico un enorme grazie, di cuore. Vorrei ora rubare un paio di righe per ringraziare qualcuno in particolare:
Grazie ai miei genitori per l'amore incodizionato e la fiducia che avete sempre riposto in me; a te, mamma, per le premure che mi hai sempre donato e che, solo negli ultimi anni, ho imparato ad apprezzare; e a te, papà, per avermi mostrato l'uomo che voglio essere. Farò di tutto per rendervi orgogliosi.
Grazie a mio fratello Gabriele che mi accompagna per mano da che ho ricordo, il mio punto di riferimento che mi ha sempre dato consigli come solo un vero fratello maggiore potrebbe ... certo non posso dire che la pazienza ci sia stata, ma è da apprezzare lo stesso.
Grazie alla mia Martina, entrata nella mia vita grazie all' ingegneria, rimasta nonostante l'ingegneria, che ha reso tutto diverso da quando è al mio fianco e mi ha fatto scoprire molto di più, oltre ai doveri e università, compreso me stesso. Non vedo l'ora di conoscere cosa il futuro riserva per noi.
Grazie al mio migliore amico Luigi per l'amicizia sincera e il tuo esempio di passione e dedizione, che contagia tutti quelli che ti sono intorno. Spero tu diventi il medico che hai sempre sognato.
Grazie al mio unico e irripetibile amico Mattia, per avermi insegnato a guardare più in là e ad avere sempre presenti i propri obiettivi; l'augurio di soddisfarli tutti.
Grazie a Luca, l'amico più misterioso di tutti ma anche il più spontaneo, ad Angelo che, da quando non c'è più, casa Boves non è la stessa, e tutti gli amici speciali di vecchia data di Lucera.

Grazie agli amici Torinesi, per avermi tenuto compagnia quelle intere giornate in aula studio, spese tra chiacchiere e caffè, il ricordo più belli di questi anni universitari che non torneranno più.
Grazie infine al mio più bel angelo custode, nonna Giovina, per me una seconda mamma che rappresenta forza e responsabilità. Tu per prima mi hai fatto capire l'impegno di "andare a scuola", aiutandomi a vestire quelle mattine in cui non mi andava proprio di andarci, tu che ridevi e ti interessavi quando ti raccontavo quello che avevo imparato. Grazie di tutto.

Premessa

Questa tesi di laurea è stata scritta a conclusione del corso di Laurea Magistrale in Ingegneria Informatica al Politecnico di Torino, al termine di 5 mesi di lavoro trascorsi presso Reply, in particolare presso il laboratorio Area42 di Torino. Area42 è il laboratorio in cui si sperimentano e concretizzano le idee più creative, sfruttando il potenziale delle tecnologie più innovative nel campo della robotica, mobilità avanzata e virtual reality. Qui è stata predisposta un'area specifica a emulazione di un magazzino reale con scaffali, scatoloni, codici a barre, in modo da ricreare le condizioni di lavoro degli utenti target della tesi.

Il progetto, un sistema di prelievo vocale a supporto degli operatori di magazzino, è stato sviluppato da me, Antonio Vespa, con il supporto di Reply Logistics, rappresentata dal tutor aziendale Riccardo Sioli.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Il contesto in cui si colloca questa tesi è il magazzino moderno, ossia quella struttura logistica in grado di ricevere le merci, conservarle (stoccaggio) e renderle disponibili per lo smistamento, la spedizione e infine la consegna. I centri logistici non sono tutti uguali, bensì variegati in base a grandezza, configurazione degli spazi, livello di automazione, tempistiche e modalità di stoccaggio.

Esistono infatti magazzini, nell'accezione comune del termine, pensati per ricevere e stoccare materie prime/articoli su scansie e per tempi variabili, potenzialmente indefiniti, in attesa di lavorazione/spedizione; altri come i "Cross-Docking Points" che fanno del tempo la principale risorsa: qui si prevede che tutta la merce entrante nel magazzino esca in giornata, non occupandone gli scaffali; o ancora i cosiddetti "Magazzini Mobili", costruiti senza opere murarie per essere all'occorrenza montati e smontati, configurati in genere come coperture, tunnel o capannoni in acciaio inox e pvc, utilizzati strategicamente per aumentare lo spazio coperto per lo stoccaggio merci.

Il tipo di merce che potremmo trovare in magazzino è molto variegata: materie prime, prodotti intermedi o finiti, e in base alle loro caratteristiche si basano le modalità dello stoccaggio, i cui fattori chiavi sono: la *conservazione* e *l'allocazione*. Il primo fattore consiste nel controllare le quantità di prodotto stoccato in deposito, in modo tale che sia sufficiente per l'immissione periodica al consumo e non superiore alla domanda di mercato; il secondo invece riguarda la disposizione in magazzino della merce in modo strategico: per avere l'allocazione più efficiente possibile, è necessario tenere conto dei "tassi di rotazione", ossia il tempo medio impiegato a consumare una scorta e a rifornirla nuovamente. L'obiettivo di tutto ciò è far si che i prodotti maggiormente venduti siano sempre presenti in magazzino e quindi pronti per la spedizione, posizionati in posti comodi e facilmente accessibili per essere *prelevati*.

L'attività focus di questo progetto è il prelievo manuale della merce, in inglese "picking", la cui esecuzione è successiva allo stoccaggio e la prima delle operazioni

in uscita del magazzino. In particolare, il prelievo consiste nel collezionare colli, confezioni, pezzi sfusi, ecc. recapitandoli dalle relative posizioni in magazzino, univocamente identificate; gli oggetti raccolti sono posti in unità-di-carico[1] di trasporto (si può pensare a una scatola), avente una specifica *destinazione*, che a fine missione il pickerista dovrà raggiungere. L'attività di picking è svolta ogniqualvolta è necessario raggruppare pacchi, componenti, materiali che, una volta riuniti, verranno elaborati e spediti per soddisfare l'ordine di un cliente finale (che sia Business-to-Business o Business-to-Client) o per rifornire le scorte di altri reparti della supply chain[2], come quello di produzione e/o di lavorazione.

Il prelievo ha un ruolo chiave nel ciclo di operazioni di un centro logistico, ma altrettanto un grande impatto sull'efficienza generale, dal momento che i costi legati a questa attività, la più lenta e dispendiosa tra tutte, superano generalmente il 50% dei costi complessivi. Alla luce di quanto detto, risultano fondamentali la massima *tempestività* ed *accuratezza* nel picking, per evitare un "collo di bottiglia" per la qualità del servizio offerto al cliente, basilare per la reputazione di un'azienda.

Tradizionalmente il picking è sempre stato legato all'uso documenti stampati (le cosiddette bolle), per loro natura lenti e a volte scomodi da consultare soprattutto quando si tiene per le mani taglierini, oggetti pesanti come la scatola (udc) in cui viene posta la merce prelevata, oppure si è alla guida di muletti/carrelli sollevatori: tutto ciò rende pericolosi gli spostamenti in magazzino.

Negli ultimi anni però, i magazzini di tutto il mondo sono stati interessati dall'introduzione di nuove tecnologie, sia per la *digitalizzazione* dei processi che per l'*automatizzazione* attraverso l'introduzione di robot in grado di catalogare, imballare e depositare: quest'ultimo rimane però preclusa a grandi aziende con altrettanto grandi fatturati. In particolare, l'introduzione di un software gestionale[3] ha contribuito in maniera decisiva alla digitalizzazione di tutta la filiera all'interno del magazzino, e dunque anche il prelievo. I pickeristi, dopo tanti anni di solo cartaceo, sono stati equipaggiati da palmari che, seppur ingombranti come la carta per le mani, sono in grado di supportare e facilitare il lavoro di prelievo merce: interagendo con essi, è possibile infatti leggere le informazioni di prelievo e inserire dati per segnalare il termine dell'operazione al WMS, al quale sono collegati. In questa sinergia, ruolo fondamentale è svolto dalla *banca dati centrale*,

[1]Con unità di carico o UdC si intende l'unità di base di stoccaggio e trasporto posizionata su un supporto o imballaggio modulare (cassa, pallet, contenitore ecc.) al fine di ottenere una movimentazione manuale efficace.

[2]Per supply chain o catena di approvvigionamento si intende il processo che permette di portare sul mercato un prodotto o servizio, trasferendolo dal fornitore fino al cliente

[3]Il software gestionale di magazzino, in inglese Warehouse Management System (WMS) è un sistema di gestione informatizzato, che supporta l'azienda in tutte le fasi di organizzazione, coordinamento e controllo dei movimenti e dei processi logistici.

dove ogni articolo è catalogato e contrassegnato dal proprio codice, descrizione, quantità e posizione di stoccaggio (zona-corridoio-scaffale-ripiano) per poterlo sempre localizzare in modo rapido. Proprio su queste informazioni, si basa la tecnologia principe di un magazzino moderno, la chiave di ogni processo automatico: il *barcode*. Su ogni pezzo è infatti attaccata un'etichetta con su stampato il codice a barre[4], capace di identificare l'articolo univocamente. Tutti questi codici sono collezionati nel database che verrà aggiornato ogni volta che un articolo sarà venduto o spostato: con questo meccanismo è possibile diminuire al minimo gli errori relativi alla fatturazione o alla quantità nominativa di merce in giacenza, consentendo un'amministrazione semplice, centralizzata e efficiente.

L'ultima frontiera in questo settore è la sperimentazione di *tecnologie vocali* speech-to-text (STT) e text-to-speech (TTS), che rendono i classici dispositivi da magazzino sistemi *hands-free*: questo è un grande vantaggio, non solo per l'efficienza grazie a una continua interazione uomo-macchina, ma anche per la sicurezza dell'operatore che può così maneggiare attrezzi di magazzino senza doversi preoccuparsi di liberare le mani. Questi sistemi sono capaci di ascoltare la pronuncia di comandi (scelti appositamente per la loro semplicità), e in base a questi ultimi elaborare l'output vocale, che in ultima sintesi consiste nel dare indicazione agli addetti su quali *posizioni* raggiungere all'interno di un magazzino e dopodichè dire loro quali *prodotti* prelevare. Sfortunatamente, queste funzionalità continuano a essere esclusive di alcuni vendor, rilegate a costosi ecosistemi da acquistare in "pacchetto" e incompatibili con apparecchiature terze. In aggiunta, i palmari in dotazione ai pickeristi sono pensati per lavorare nel solo contesto del magazzino: ne risultano dispositivi specific-purpose con hardware specifico integrato, come batterie removibili, scanner di codici a barre, ganci per la cintola, ecc., ma ciò significa anche costi di cambio vendor/tecnologia pesantissimi e da ciò consegue poca spinta innovativa nel settore. Per tutti questi motivi la tecnologia vocale risulta essere poco accessibile ai piccoli magazzini.

1.1 Obiettivo

L'obiettivo della tesi è la progettazione, implementazione e test di un sistema hands-free e di supporto al lavoro degli addetti al prelievo manuale in magazzino. Lo sviluppo seguirà un approccio "Human Centered Design"[1], con lo scopo di mettere al "centro" l'utente in ogni sua fase.

[4]Il codice a barre è un codice di identificazione costituito da un insieme di elementi grafici a contrasto elevato destinati alla lettura per mezzo di un sensore a scansione e decodificati per restituire l'informazione in essi contenuta.

La soluzione proposta, sfruttando tecnologie sia vocali che mobile, si configurerà come applicazione Android di tipo stand-alone, basata su funzionalità vocali offerte da un framework indipendente (approccio modulare), capace di offrire un'esperienza plug&play per un facile uso. L'architettura software di alto livello dovrebbe dunque ricalcare a grandi linee quella disegnata in figura 1.1. Il punto focale di tutto il progetto sarà l'*usabilità*, un'interiazione uomo-macchina efficiente che permetta all'utilizzatore di migliorare le sue condizioni di lavoro. L'interfaccia sarà quindi disegnata sulla base delle *esigenze* raccolte dai lavoratori addetti al prelievo in magazzino.

In aggiunta, questo progetto vuole porsi come alternativa al monopolio sistematico delle aziende colosso nel settore e i loro dispositivi special-purpose brandizzati; alla luce di ciò, un ulteriore obiettivo che ci si pone è far si che la soluzione possa essere eseguita su un device Android di tipo general-purpose/consumer (che potremmo comprare nel negozio di elettronica sotto casa), quindi *non* dotato di hardware *specifico* a bordo.

Dall'uso del sistema sono attesi benefici come: autonomia del pickerista, modalità di prelievo intuitive, tempi di formazione minimi, bassa frequenza di incidenti in magazzino, produttività alte, costi di cambio particolarmente vantaggiosi in quanto i vecchi device Android da magazzino saranno senza dubbio riciclabili.

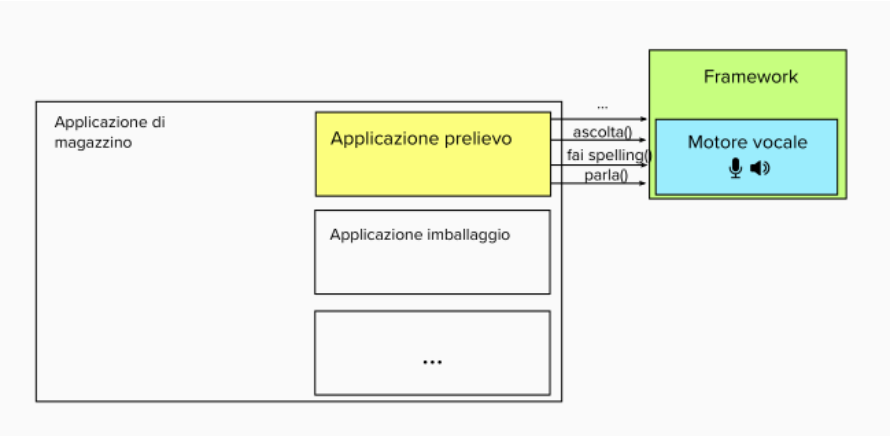

Figura 1.1: Architettura software obiettivo

Per raggiungere questi obiettivi, il problema sarà pragmaticamente spezzato in quattro fasi per approcciarlo in modo "incrementale":

- dapprima ci sarà la raccolta delle necessità degli utenti, e sulla base di questi la definizione dei requisiti del progetto;

- ricerca di un motore vocale tra quelli disponibili nel panorama Android, guidata dai requisiti fissati. Alcuni motori saranno quindi selezionati e tra loro confrontati, per capire quali tra questi è il migliore per essere implementato;
- progettazione e realizzazione di framework che offra funzionalità vocali "base", esposte come API logicamente slegate dal loro uso finale: la logistica di magazzino. Per rendere l'idea, si può pensare a una libreria che offra l'implementazione di metodi come "parla","ascolta","fai spelling". In più, questo framework si porrà come interfaccia tra le applicazioni e il motore vocale scelto con lo scopo di *disaccopiarle* e consentire in futuro la sostituzione dell'ultimo, modificando meno codice sorgente possibile delle prime. Per permettere questa generalizzazione, il framework dovrà definire *pattern di uso* generali, indipendenti dai metodi esposti direttamente dal motore selezionato al punto precedente;
- ideazione e implementazione di una applicazione che, implementando il framework descritto precedentemente, realizzi il sistema di supporto ai pickeristi durante il loro lavoro. Come descritto, non avrà nessun tipo di requisito stringente, se non le caratteristiche hardware di uno smartphone Android consumer/general-purpose di una decina di anni fa. In particolare l'app:
  - sarà caratterizzata da un'interfaccia multimodale, focalizzata sul canale percettivo sonoro: infatti l'utente potrà ascoltare e interagire parlando;
  - dovrà inoltre garantire un uso facile e efficiente, ottimizzando i movimenti e quindi riducendo le distanze percorse all'interno del magazzino;
  - essere stand-alone e non richiedere app terze di supporto: in uno scenario reale, infatti, è fondamentale che la manutenzione sia rapida e "centralizzata". Se così non fosse, l'amministratore tecnico del magazzino avrebbe un notevole carico di lavoro, non solo durante l'installazione su N dispositivi, ma anche durante gli aggiornamenti.

Per concludere, lo sviluppo dovrà svolgersi utilizzando il linguaggio nativo di Android, Kotlin, e integrarsi con l'ecosistema architetturale software di Reply Logistics. L'applicazione sviluppata, si configurerà come un modulo di una applicazione modulare più grande, preesistente e già in grado di supportare altri task di magazzino.

1.2 Struttura della tesi

Nei prossimi capitoli verranno approfondite le motivazioni di partenza questo lavoro, raccolti e analizzati i bisogni degli utenti finali, ripercorsi i vari step fino ad arrivare

alla fase dei test, passando dalla progettazione e implementazione, pesandone i risultati ottenuti. La tesi, sviluppata in più capitoli, sarà quindi cosi composta:

- Nel capitolo 2 “Background e Stato dell’Arte” verranno forniti concetti utili a contestualizzare il progetto e capire meglio i principi di funzionamento alla base.
- Nel capitolo 3 “Raccolta dei requisiti” verranno descritte le esigenze espresse dagli utilizzatori, le modalità di analisi usate e conseguentemente i requisiti che questa tesi si pone di raggiungere.
- Il capitolo 4 “Scouting e Selezione” descriverà il processo di scouting usato per selezionare i motori vocali disponibili nell’ecosistema Android nativo, e quindi quelli idonei alle finalità del progetto. Seguiranno le modalità di confronto e valutazione utilizzate per scegliere i migliori tra essi.
- Nel capitolo 5 “Progettazione e Prototipazione” si tratterà la progettazione software dapprima solo del framework vocale e poi anche dell’applicazione mobile, descrivendo nei dettagli le metodiche usate per la prototipazione di quest’ultima, durante le varie fasi. Saranno quindi ripercorse le alternative individuate, le motivazioni che hanno guidato le scelte fatte.
- Nel capitolo 6 “Implementazione” sarà descritta l’implementazione del framework e applicativo, soffermandosi sull’integrazione realizzata per far dialogare tutti i componenti del progetto;
- Il capitolo 7 “Test preliminari con gli esperti” presenterà la descrizione delle modalità scelte per effettuare i test, per capire i criteri con cui il progetto è stato testato e come sono stati ottenuti i suoi risultati;
- Nel capitolo 8 “Risultati e obiettivi raggiunti” ci sarà l’estrazione e analisi dei risultati ottenuti dai test, andando a confrontare gli obiettivi raggiunti con i requisiti che si erano prefissati a inizio lavoro. Questo paragone servirà a valutare la buona riuscita del progetto;
- Per concludere, il capitolo 9 “Conclusioni” ripercorrerà le varie fasi del progetto, arricchite da riflessioni a posteriori, andando infine ad indicare possibili sviluppi futuri da realizzare per un ancor più efficace utilizzo e migliori prestazioni.

Capitolo 2

Background e Stato dell'Arte

2.1 Processo di Picking

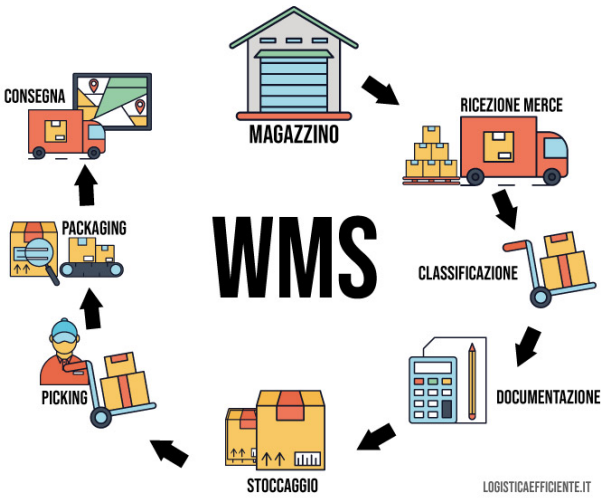

Figura 2.1: Operazioni base di un centro logistico

Il ciclo di attività che interessa i magazzini di stoccaggio, visibile in figura 2.1, può dividersi in maniera molto semplificativa in operazioni in *entrata* e operazioni in *uscita*. Se le operazioni in entrata riguardano l'accettazione della merce e il loro successivo deposito in magazzino, viceversa quelle in uscita, nel caso di preparazione ordini, sono il prelievo degli articoli (picking), imballaggio (packing), e infine la

distribuzione al cliente finale [2]. Più in generale il picking di magazzino è l'attività di prelievo, smistamento e ripartizione di materiale da un'unità di carico a diverse altre. Questa attività viene svolta per raggruppamento di materiali con il fine di elaborarli e spedirli.

Ci possono essere diverse strategie di attuazione picking:

- quando gli articoli presenti negli ordini hanno una consistente ripetitività, è possibile affidarsi al *picking massivo*, in inglese "batch picking", una modalità di prelievo che garantisce efficienza operativa e elevata produttività in quanto i prodotti sono presi a prescindere dagli effettivi ordini pervenuti.
- se l'operatore effettua giri di prelievo per un insieme limitato di ordini, possiamo parlare di *picking raggruppato* o multi-ordine.
- il *picking per ordine*, la modalità di prelievo più diffusa, consiste nel raggruppare gli articoli necessari a soddisfare un singolo ordine, nel quadro di una strategia "pick&pack": l'attività di handling ordine sarà fatta una sola volta, unendo la fase di prelievo a quella di impacchettamento.

Figura 2.2: Esempio di prelievo manuale

Il prelievo fisico del prodotto può avere un'accezione *manuale* se l'estrazione di un articolo dagli scaffali è compiuta mediante un addetto al picking che manualmente lo raccoglie, oppure *automatica* se invece l'estrazione avviene per mezzo di sistemi automatici, come ad esempio nei magazzini robotizzati.

Affinché il processo di picking si svolga in maniera efficace, è necessario pianificare in anticipo le attività di prelievo (preparazione della *picking list*), secondo una logica di *ottimizzazione* del percorso compiuto dagli operatori. A tal scopo, nei magazzini moderni si ricorre a un software gestionale (il WMS) che, prima di

avviare le procedure di picking, si occupa di calcolare le caratteristiche di *tutti* gli articoli da prelevare (numero, posizione, peso, dimensione, ecc...), in modo da trovare il percorso *ottimo* all'interno delle aree del magazzino, che minimizzi le distanze coperte dal pickerista esecutore e perfino i suoi movimenti tra gli scaffali e ripiani.

2.2 Voice Picking

Con voice picking [3] si intendono soluzioni di prelievo vocale, in genere sistemi hands-free che non hanno dunque bisogno di nessun tipo di supporto visivo, che sia cartaceo o l'interfaccia grafica di un dispositivo. Questo tipo di sistemi sono capaci, in input, di ascoltare comandi vocali scelti appositamente per la loro semplicità, grazie all'intelligenza artificiale, e sulla base di questi elaborare l'output che in ultima sintesi significa dirigere gli addetti in posizioni specifiche all'interno di un magazzino, e poi dire loro *quali* prodotti prelevare. In particolare possiamo definire una *posizione* in magazzino come una combinazione di coordinate, quali area - corridoio - scaffale - ripiano, come rappresentate in figura 2.3.

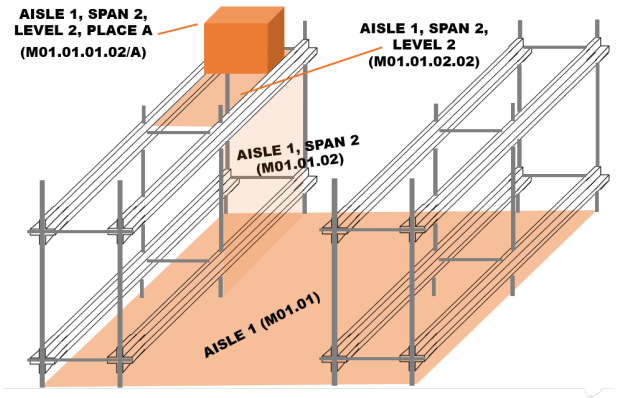

Figura 2.3: Sistema di coordinate usato per indicare una posizione

2.2.1 Come funziona

L'architettura software [4] di un WMS è di tipo *client-server* (nelle soluzioni più recenti può essere anche di tipo web-server): il server WMS è collegato ad un host e, a un livello inferiore, a degli access point posti all'interno del magazzino per consentire alle informazioni di essere scambiate in tempo reale tramite dispositivi mobili.

I raccoglitori portano con sé il proprio dispositivo mobile che, eseguendo un'applicazione con funzionalità vocali, importerà dapprima gli ordini dei clienti pendenti

Figura 2.4: Architettura tipica di un WMS

e, sulla base di questi, riferirà ai magazzinieri *dove* andare e *cosa* raccogliere. L'interazione del lavoratore con il sistema può essere tramite voce, immissione di testo o scanner di codici a barre: ad ausilio dell'interfaccia vocale, potremmo dunque trovare anche altre, supportate da dispositivi esterni come scanner barcode (guanti o anelli) o direttamente incorporati sul device.

In genere, per confermare il prelievo, basterà pronunciare un input *definito*, opportunamente validato dal sistema, grazie al supporto di un software di riconoscimento vocale, che trasformerà da parlato a testo ciò che è stato pronunciato: se è conforme a quanto atteso, allora il WMS saprà che l'operazione è andata a buon fine e potrà marcare la presa come "terminata". Se esistono prese successive, il sistema invierà il raccoglitore all'ubicazione di prelievo successiva, fornendo un percorso ottimale.

Figura 2.5: Esempio [5] di prelievo con tecnologia vocale

2.2.2 Vantaggi

Il picking è spesso il processo più costoso, in termini di tempo e denaro, per un magazzino/centro di distribuzione, il che lo rende un ottimo candidato all'ottimizzazione tramite tecnologia vocale. Quest'ultima infatti aiuta gli utenti a concentrarsi sulle proprie attività: mani e occhi sono liberi da fogli di istruzioni e/o dispositivi, e quindi possono individuare rapidamente gli oggetti, selezionare la giusta quantità e posizionarli nel posto appropriato. Una serie di noti [3] vantaggi raggiunti:

- aumento della *produttività* oltre il 30%
- maggiore *accuratezza* - Gli studi lo dimostrano che il tasso medio di errore di prelievo è compreso tra 1 e 3 o tra 10 e 30 errori per 1.000 prelievi. Grazie alla tecnologia vocale la precisione aumenta a 99,99 o a un solo errore circa ogni 1.000 selezioni. Meno errori significa più soldi.
- tempo di *formazione* ridotto all'85% - gli operatori sono guidati da istruzioni vocali semplici e di facile comprensione. Ogni compito è esplicato passo dopo passo e ciò significa che la formazione sulla lettura di documenti non è più necessaria. Essendo la formazione minima, il prelievo vocale è l'ideale per ambienti con molti lavoratori stagionali o con elevato turnover di magazzinieri.
- magazzino più *sicuro* - gli operatori spesso devono usare taglierini o sollevare oggetti pesanti, quindi è necessario usare le mani e il prelievo vocale consente di mantenerle libere.
- clienti *soddisfatti* - i livello del servizio offerto al cliente è migliorato direttamente dall'aumento della precisione e velocità di selezione.
- impiegati più *felici* - i lavoratori preferiscono usare la tecnologia vocale perché facile da usare, permette una breve formazione minima, e li rende più efficienti e produttivi. E' stato osservato che impiegando questo tipo di soluzione il tasso di abbandono è ridotto del 30%.

2.2.3 Dispositivi

I dispositivi, che possiamo trovare nella dotazione [6] di un pickerista, possono essere categorizzati in mobile e/o scanner barcode, cuffie e guanti touch. Le pecularità di questi prodotti è la *robustezza* agli urti, batterie facilmente removibili, accessori speciali con pin POGO[1], ma di contro hanno lo svantaggio di avere costo

[1]Un POGO pin è un tipo di connettore elettrico utilizzato per la sua resilienza a shock meccanici e vibrazioni.

elevato, scarsa *flessibilità*, nessuna *interoperabilità* tra vendor diversi. Per quanto riguarda il sistema operativo, se presente, può essere: specific purpose se sviluppato appositamente per operazioni nel warehouse (basato su Linux), o general purpose se pensato per supportare operazioni variabili e più complicate (basato su Android). Di seguito un breve campionario per capire meglio la dotazione di un comune operatore al picking in magazzino:

Dispositivo	Descrizione	Prezzo
	Dispositivo con sistema operativo Android e scanner barcode integrato con impugnatura ergonomica, tasti fisici.	~ 2500 €
	Dispositivo con sistema operativo Android, scanner barcode integrato ma senza impugnatura, schermo touch.	~ 1000 €
	Scanner barcode cablato o con tecnologia Bluetooth, senza display o display di piccole dimensioni.	~ 50 €
	Piccolo scanner barcode (ring-scanner) da indossare sulla mano, disponibili con display full touch (personalizzato) in modalità orizzontale o verticale per computer mini-mobile con LCD.	~ 900 €
	Headset con tecnologia Bluetooth, NFC, immunità al rumore, incluso fascia, paraorecchie, adatto per i dispositivi Android o soluzioni proprietarie.	~ 400 €

2.3 Motori Vocali

La tecnologia vocale è centrale nello sviluppo di soluzioni di voice-picking, in quanto offre due funzionalità basilari quanto complementari: conversione da parlato a testo (Speech-To-Text) e viceversa da testo a voce (Text-to-Speech). Grazie a queste sarà possibile sostenere un'interazione vocale/sonora, senza la necessità di altri input

sensoriali. In questo paragrafo capiremo meglio come sono a oggi implementate queste feature, il che tornerà utile per orientarsi nel capitolo dello scouting.

2.3.1 Speech-To-Text

Cos'è Speech Recognition

Il riconoscimento vocale [7], o sintesi vocale, è la capacità di *identificare* le parole pronunciate ad alta voce e *convertirle* in testo leggibile, utilizzando un'ampia gamma di soluzioni informatiche, linguistiche e ingegneristiche.

Non tutti i software di riconoscimento vocale sono uguali: alcuni sono più rudimentali, dispongono infatti di un vocabolario semplificato e/o di una capacità di identificazione di parole e frasi limitata a una pronuncia chiara e precisa, altri sono più sofisticati e per questo in grado di gestire il parlato naturale (NLU), accenti diversi o lingue diverse.

Funzionamento

I sistemi di riconoscimento vocale utilizzano algoritmi informatici per elaborare e interpretare le parole pronunciate e convertirle in testo. Un programma software trasforma il suono registrato da un microfono in un linguaggio scritto che i computer e gli esseri umani possono capire: l'audio viene registrato e diviso, pre-elaborato per la rimozione del rumore e digitalizzarlo, infine nuovamente elaborato per abbinarlo alla rappresentazione testuale più adatta. In aggiunta, è importante che il software di riconoscimento vocale si adatti alla natura altamente variabile e specifica del *contesto* in cui viene usato: basti pensare a quello di un magazzino qualunque, con addetti ai lavori stranieri o con un accento molto stretto. Per risolvere queste problematiche, gli algoritmi software che elaborano l'audio sono addestrati su diversi modelli di dialogo, stili linguistici, dialetti, accenti.

In generale, i metodi più usati per questi scopi sono quelli di *intelligenza artificiale* (AI), e più specificatamente quelli che prevodono un apprendimento automatico grazie al *deep learning* e alle *reti neurali*, schematizzati in figura 2.6. Questi sistemi utilizzano la grammatica, la struttura, la sintassi e la composizione dei segnali audio e vocali per elaborare il parlato. I sistemi di apprendimento automatico acquisiscono conoscenze ad *ogni* utilizzo, il che li rende adatti a sfumature difficili da schematizzare, come gli accenti.

Applicazioni e Vantaggi

I sistemi di riconoscimento vocale hanno molte applicazioni, tra le quali spiccano: negli smartphone assistenti vocali per l'accesso a tutte le sue funzionalità (chiamate, navigazione web, composizione di messaggi, ecc...), assistenti vocali artificiali (bot)

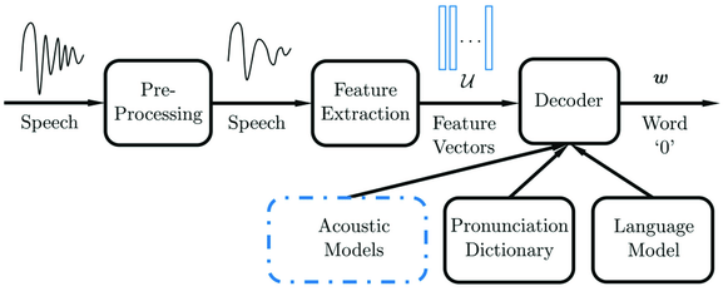

Figura 2.6: Diagramma base di un sistema [8] di riconoscimento vocale, basato su AI

capaci di fornire risorse utili a fronte di richieste, assistenza a disabilità uditive con traduttori voce/sottotitoli, comunicazione a mani libere con il computer di bordo durante la guida (aerei o navi, e ultimamente anche automobili).

I vantaggi che si possono sperimentare nell'uso di software con riconoscimento vocale sono: interazione uomo-macchina più *naturale*, *accessibilità* per i disabili, miglioramento delle performance continuo e automatico se basati su IA, mentre di contro gli svantaggi: elaborazione vocale che soffre di *alte latenze* se le richieste sono fatte in rapida successione, performance *incoerenti* di fronte a variazioni nella pronuncia (accenti), performance *dipendenti* dalla qualità dall'hardware usato per la registrazione (microfono).

2.3.2 Text-To-Speech

Cos'è Speech Synthesis

La sintesi vocale [9] consiste nella *riproduzione artificiale* del linguaggio umano, ossia il processo *inverso* del riconoscimento vocale; un qualsiasi sistema informatico che converta il testo nella lingua parlata, viene chiamato "sintetizzatore vocale". Le prestazioni di quest'ultimo sono giudicate dalla somiglianza dell'output con la voce umana e dalla sua semplicità di comprensione.

Funzionamento e Metodologie

Un sistema di sintesi vocale è composto da due fasi: una in entrata di pre-elaborazione (*front-end*) e una in uscita (*back-end*): la prima converte il grafema a fonema e contrassegna le unità prosodiche (frasi, clausole), utili insieme a creare una rappresentazione "*sonora*" dell'input testuale; la seconda, invece, è il "sintetizzatore" che, partendo dalla rappresentazione ottenuta precedentemente, crea il suono prestando particolare attenzione al "contorno" del tono.

L'audio sintetizzato può essere creato in due modi: il primo è un metodo "fisico" e consiste nel ricreare i fenomeni fisici che permettono l'emissione e articolazione della voce umana, attraverso modelli biomeccanici del tratto vocale umano, mentre il secondo, più vicino a noi, è di tipo "programmatico". Quest'ultimo tipo comprende la *concatenazione di tracce* di parlato archiviate in un database; tali sistemi differiscono per le dimensioni delle unità memorizzate: in genere quelli che memorizzano foni/difoni forniscono la gamma di uscita più ampia, ma da ciò non dipende la "chiarezza" che invece è influenzata da altri fattori come ad esempio la tonalità, frutto dell'algoritmo scelto per accostare i suoni. Per domini di utilizzo specifici (stazione dei treni), la *memorizzazione* di intere parole o frasi consente un output di alta qualità, ma per usi più generali potrebbe essere utile una sintesi vocale basata su *AI* e deep learning con reti neurali profonde (DNN), addestrate utilizzando una grande quantità di suoni registrati. Questi ultimi si stanno avvicinando alla naturalezza della voce umana ma soffrono di scarsa controllabilità, necessità di enormi quantità di dati.

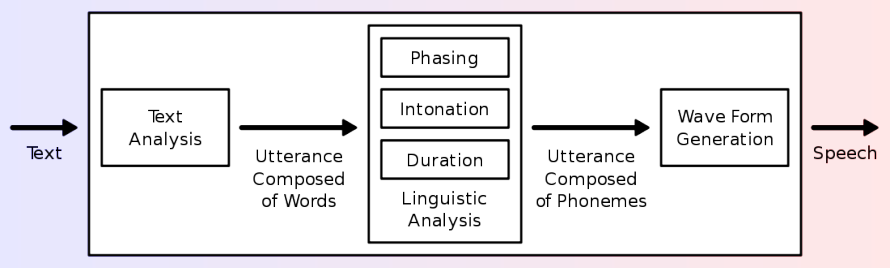

Figura 2.7: Diagramma base di un sistema di conversione testo-audio

Applicazioni

La sintesi vocale nasce come strumento di tecnologia assistiva, e a oggi la sua applicazione in questo settore è significativa e diffusa: consente di rimuovere barriere per persone con diversi tipi disabilità permanenti, temporanee o circostanziali. Esempio tipico di impiego è lo screen-reader[2] per persone con disabilità visive e quindi difficoltà di lettura (dislessia), nonché da bambini pre-alfabetizzati. Negli ultimi anni, la sintesi vocale sta trovando nuove applicazioni: è usata in produzioni di intrattenimento come giochi e animazioni, permette la creazione di video AI con teste parlanti che mimano fedelmente le espressioni facciali umane, come in figura 2.8.

[2]Uno screen-reader identifica ed interpreta il testo mostrato sullo schermo di un computer, presentandolo come output in sintesi vocale

Figura 2.8: Video AI [10]

Capitolo 3

Requisiti

In questo capitolo si definiscono i requisiti della tesi, sia in termini *tecnici* che di *usabilità*, evidenziandone i motivi di partenza e da dove essi nascono; successivamente gli stessi requisiti saranno schematizzati in una tabella, per essere facilmente consultati e confrontati con i risultati dei test condotti a progetto terminato: in questo modo si avrà una visione complessiva della riuscita del progetto, e sarà dunque possibile valutare se quest'ultimo è riuscito con successo o meno.

3.1 Le necessità di cui tenere conto

Un sistema per essere "usabile" [11] deve essere: utile, utilizzabile e usato. In poche parole il sistema dovrà consentire all'utente di realizzare un compito, portarlo a termine in modo piacevole, semplice e efficace, ed essere realmente usato. Per raggiungere quest'obiettivo, ho scelto di adottare un approccio che mettesse al centro della progettazione software le necessità, i desideri, i limiti dell'utente (Human Centered Design). Alla luce di ciò, le domande che mi sono posto sono state:

- Quali sono le *necessità*, *desideri* e *limiti* dei pickeristi?
- Come può l'introduzione di un sistema che coinvolga tecnologie mobile/vocali, *supportare* il lavoro quotidiano degli operatori addetti al prelievo in magazzino?

Per cercare di dare una risposta ai precedenti quesiti, non avendo la possibilità di intervistare/osservare i diretti interessati, si è fatto tesoro sia delle testimonianze riportate dagli esperti che da molti anni lavorano in questo campo all'interno di Reply, sia di articoli e filmati rappresentativi delle dinamiche di lavoro in magazzino.

In seguito alla valutazione di questo set di informazioni, quantificabile in 2 interviste, 3 annunci di lavoro con allegata descrizione e 5 filmati visionati, si è adottato il metodo di "immaginazione degli utenti", ossia una tecnica che consiste

nell'immaginare le cosiddette *personas* [12] nella veste di particolari ruoli, e su questa base descriverle dettagliatamente. In particolare, potremmo definire una *persona* come un vero e proprio identikit, un profilo fittizio, che rappresenta: i bisogni, i comportamenti, gli interessi e le aspirazioni di un utente tipo, più o meno accentuati a seconda degli scopi per cui è stato creato.

3.1.1 Testimonianze dirette

Nelle prime settimane di progetto, per meglio capire il contesto, le necessità, i problemi e gli strumenti usati dai pickeristi, ho avuto modo di parlare con due impiegati Reply che negli anni passati hanno maturato esperienze di *contatto diretto* con questi ultimi. In particolare, ho intervistato due figure lavorative tra loro ben diverse, ma complementari: Riccardo in qualità di "Tech Leader", ossia specialista di soluzioni tecnologiche innovative in campo logistico, e Andrea come manager in campo "Delivery", in altre parole la figura di riferimento nell'erogazione di servizi al cliente (in questo caso i proprietari dei centri logistici).

Le interviste sono state strutturate come *sessioni singole* di tipo faccia-a-faccia, della durata di 60 minuti, e hanno avuto luogo nel laboratorio "Area42" presso la sede Reply in via Nizza 250 a Torino. Si è deciso di mantenere i due colleghi separati affinchè fossero più liberi di esporre le proprie idee, senza condizionarsi a vicenda. Entrambi i colloqui sono iniziati con un'introduzione per spiegare le motivazioni dello studio e il perchè della necessità di conoscere meglio il lavoro di prelievo e i suoi addetti. Dopo una fase di conoscenza reciproca, in cui è stato chiesto di descrivere nel dettaglio il proprio lavoro, al partecipante sono state poste alcune domande aperte per guidare il dibattito.

Intervista all'esperto tecnico: Riccardo

Le domande poste a Riccardo sono state:

- Quali tecnologie si possono trovare in un magazzino di stoccaggio?
- Quali necessità soddisfano e quali difficoltà vengono riscontrate?
- Quali dispositivi proposti nel passato sono stati scartati, e quali invece sono piaciuti? Perchè?
- Cosa non ti ho chiesto?

Riccardo, che da anni si occupa di ricerca e sviluppo di soluzioni logistiche all'avanguardia, è abituato ad aver a che fare sia con i pickeristi, che con i responsabili tecnici del magazzino, in quanto la tecnologia ha più di una faccia: se da un lato deve risultare usabile per gli operatori, dall'altro deve anche essere facilmente

mantenibile e quindi il più possibile plug & play. Sono state poi ripercorse le tecnologie attuali più usate e i relativi strumenti (di cui è stato già discusso nel capitolo 2), i risultati raggiunti negli ultimi anni dalla moderna digitalizzazione, grazie al passaggio dai vecchi documenti cartacei ai dispositivi mobili. Per quanto riguarda i limiti, si è invece posto l'accento sulla sicurezza sul posto di lavoro, spesso inficiata da mani e occhi impegnati dall'interfaccia, sui problemi relativi a un'interazione di tipo "punta e clicca" che risulta essere poco naturale per utenti con scarsa familiarità con la tecnologia, poi ancora della stanchezza agli occhi provocata da interfacce grafiche con fondo bianco se usate in ambienti poco luminosi.

Tra le soluzioni innovative proposte nel passato, quelle maggiormente rilevanti sono: i droni da interno e i Vuzix [13] abbinati a tecnologie vocali. I primi offrivano il vantaggio del disimpegno della manodopera e, in più, la possibilità di raggiungere altezze sugli scaffali non possibili per i normali operatori, ma di contro potevano trasportare solo pesi leggeri e necessitavano di spazio per muoversi nel magazzino (poichè un po' più massicci rispetto ai droni così come li conosciamo). Furono scartati perchè pensati per lavorare da soli, nel buio di un magazzino fermo nelle ore notturne, al termine del turno pomeridiano dei pickeristi: trovarono poca applicazione, in quanto la quasi totalità di centri logistici è operativa 24h. I secondi invece (visibili in figura 3.1), erano smartglasses monoculari a controllo vocale, capaci di offrire un'esperienza di picking in realtà aumentata: le unità di carico da cui prelevare erano segnalate visivamente, e le azioni confermate vocalmente, ma senza risposta audio. I Vuzix ricevettero una larga sperimentazione, infatti piacquero molto poichè offrivano un'interiazione semplice, a mani libere, ma dopo un po' vennero scartati poichè scatenavano mal di testa non banali: porre infatti un dispositivo di fronte a un solo occhio, significa anche potersi concentrare su una sola profondità per volta: solo quella reale o solo quella virtuale, e non entrambe contemporaneamente. Di quest'ultima esperienza, è però rimasta la curiosità per il vocale: molti operatori sottoposti ai test affermarono di preferire questo tipo di interazione, alla UX classica, per la sua semplicità.

Figura 3.1: Vuzix-M300XL

Intervista all'esperto funzionale: Andrea

Le domande poste ad Andrea sono state:

- Che tipo di persone potremmo trovare a lavoro in magazzino? Com'è strutturata una giornata tipica di un pickerista?
- Quali esigenze hai potuto osservare? Quali le difficoltà più comuni?
- Nelle collaborazioni passate avete mai implementato tercnologie vocali? Cosa ricordi di quella esperienza?
- Cosa non ti ho chiesto?

Andrea, essendo una figura intermedia tra il prodotto e il cliente, negli anni ha maturato esperienza funzionale oltre che tecnica, seguendo da vicino le transizioni tecnologiche dei clienti: è suo compito facilitare la rimozione degli ostacoli che si potrebbero presentare nel processo, modificando la tecnologia in base alle esigenze espresse.

In magazzino possiamo trovare persone da poco entrate nel mondo del lavoro o alla loro n-esima esperienza, di origine italiana o straniera, di età compresa tra i 20 e i 55, persone con diverse attitudini con la tecnologia e livelli di istruzione molto differenti: alcuni hanno un'educazione superiore, ma molti altri non hanno frequentato neanche le scuole elementari, con conseguenti difficoltà nella lettura/scrittura.

Per quanto riguarda invece la routine, i turni di un pickerista possono essere al mattino, pomeridiani oppure notturni. Una volta arrivato sul posto di lavoro ed essersi sistemato per iniziare ad operare, il pickerista dovrà consultare la lista di prelievi per verificare l'ordine delle attività da svolgere e capire la sequenza di ubicazioni in cui dovrà dirigersi, in base alle priorità dei task. Dopo aver raggiunto gli articoli presenti negli ordini, sarà suo compito prelevarli dagli appositi scaffali, servendosi di tutti gli strumenti di cui ha bisogno (muletto, scala o altro), e dopo averli collezionati riporli nell'area di imballaggio. Seguiranno altri ordini fino a fine turno, intervallati solo dalla pausa pranzo.

Spesso i pickeristi lamentano difficoltà dovute a interfacce grafiche con font poco leggibili, tasti troppo piccoli per essere toccati con i guanti; durante le ore notturne stanchezza agli occhi causata da lettura di testi neri su sfondi bianchi. Rimarchevole l'incertezza espressa di fronte a errori particolari, che non hanno mai gestito: qualche volta preferiscono stare zitti pur di dare l'idea ai superiori di non avere dubbi sul lavoro, di sapere sempre cosa fare. Molte volte hanno bisogno solo di essere incoraggiati: succede che si sentano sminuiti da un lavoro così meccanico, e se sbagliano finiscono per buttarsi giù d'animo perchè gli errori in magazzino comportano ritardi molto gravi e chi sbaglia viene sempre richiamato.

Inoltre, l'addestramento rappresenta un momento critico per tutti gli addetti al prelievo, in cui la percentuale di abbandono è molto alta. Alcuni mostrano difficoltà nell'imparare a usare l'interfaccia dell'applicazione, soprattutto se non godono di particolare affinità con la tecnologia, e invece quasi tutti sono all'inizio impacciati nell'usare il mobile mentre si sta guidando o sollevando pesi: alcuni lo appoggiano sugli scaffali e lo dimenticano, altri cercano di tenerlo in mano ma finiscono con il far cadere tutto. Il tutto potrebbe essere complicato dall'eventualità di un software gestionale di magazzino che non gestisce l'ordine delle missioni di prelievo: i pickeristi sono pertanto chiamati a prendere decisioni autonome.

Per quanto riguarda infine le tecnologie vocali, in passato erano state vendute soluzioni basate su sistemi operativi chiusi, brandizzati, che obbligavano i magazzini a enormi spese di ingresso nella tecnologia. Il vocale fu apprezzato per i molti benefici apportati: aumento produttività, soddisfazione dei clienti, meno errori, inventari più precisi, ecc., ma si sperimentarono anche effetti negativi non prevedibili. In particolare ci furono difficoltà sia tecniche che di manutenzione: la prima consistette in ritardi di sviluppo dovuti ai contatti molto frequenti con l'azienda provider della tecnologia vocale, necessari per richiedere le modifiche da apportare al vocabolario, di base limitato a solo poche parole; la seconda invece si verificò in magazzino durante i mesi che seguirono la prima implementazione, a causa del supporto multilingua. Si permise infatti ai lavoratori stranieri di poter lavorare nella loro lingua madre, ma il sistema non performava ugualmente in tutte le lingue e non era stato "allenato" a riconoscere i relativi accenti, storpiature: il responsabile di magazzino non sapeva a chi dare la colpa, se al sistema o al dipendente.

3.1.2 Testimonianze indirette

Parallelamente alle interviste, è anche stato raccolto materiale che mi permettesse di conoscere meglio il contesto, i lavoratori, le dinamiche: si parla di testimonianze indirette, poiché lo scopo ultimo per cui questi articoli/filmati furono redatti è la promozione alla vendita di uno specifico prodotto o l'assunzione di personale. Sono qui riportati i più utili tra le 8 referenze consultate (appendice A.1), quelli che hanno contribuito ad arricchire le informazioni già precedentemente acquisite dalle testimonianze dirette:

Articolo Online - Questo articolo è scritto per far capire meglio a chi cerca lavoro, il tipo di possibilità lavorativa proposta. Allegata alla descrizione del lavoro, anche le qualità di un buon pickerista, richieste al candidato: ottima manualità e concentrazione, velocità di esecuzione, puntualità al lavoro e flessibilità di orari (con disponibilità a straordinari) per i periodi più caldi dell'anno, quando i ritmi in magazzino crescono a dismisura.

Video 1 - Questo filmato è stato creato per addestrare volontari in magazzini sociali. Qui vengono mostrati: come maneggiare carrelli elevatori, come riconoscere le aree in cui dirigersi per prelevare articoli, piccoli trick per la disposizione di imballaggi sui pallet, e si consiglia in particolare di dare un'occhiata, all'inizio di ogni prelievo, a quanti colli saranno prelevati, in modo tale da sapere esattamente che tipologia di udc sarà necessaria per collezionare gli oggetti. Oltre a ciò, si spiega anche la modalità di conferma prelievo con il mobile (minuto 7:35), piuttosto macchinosa per i tanti input da tastiera che facilmente portano all'errore, come in figura 3.2.

Video 2 - In questo video vengono pubblicizzati i Motorola-NC9090 e con esso una soluzione di picking vocale interattiva. Qui la testimonianza reale di un pickerista di nome Kumar Balvir (minuto 4:15), dove ricorda la sue vecchie difficoltà manuali nel gestire gli ingombri del magazzino e gli strumenti da lavoro, poi superate dall'introduzione della tecnologia vocale di cui invece apprezza molto la semplicità di interiazione (che a suo dire non lo fa mai sbagliare) e un conseguente incremento sensibile della produttività. Da notare il suo forte accento straniero.

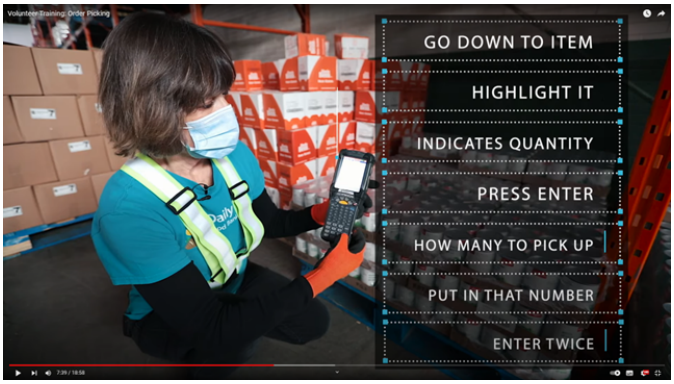

Figura 3.2: Esempio di conferma prelievo su mobile

3.1.3 Personas

Dopo aver consultato le diverse fonti, intervistato chi da anni supporta questo lavoro portando soluzioni tecnologiche innovative, ho individuato 5 diversi *modelli di comportamento*: ciascuno di questi è affidato a un preciso identikit fittizio (*persona*), ulteriormente arricchito da dettagli osservati e talvolta supposizioni generali per renderlo più "umano" e empatico. Di seguito le cinque personas immaginate, realizzate con lo scopo di facilitare la fase di *progettazione* della

soluzione: ognuna aiuterà il designer a concentrarsi sull'utente, sul suo preciso modello di comportamento e le sue esigenze.

Mario D'Azeglio - maschio di 34 anni, vive nelle periferie di Roma, diplomato, figlio unico e single. Nella sua carriera lavorativa ha cambiato molti lavori, tutti di breve durata e piuttosto manuali con basso uso di tecnologia. Ha da poco iniziato a lavorare nel magazzino centrale di Rebibbia, e dopo una breve formazione è stato assegnato al picking. Mario si sveglia presto la mattina, prende la macchina e arriva a lavoro. Attacca alle 8:00 e stacca alle 17:00, e un po' per la bassa dimestichezza con device mobile, un po' per la bassa esperienza, cerca di non sbagliare nulla per non dover gestire errori: questo lo costringe a impiegare più tempo, oltre che stancarlo per la costante attenzione ai dettagli. Mario vuole evitare di chiedere al capo squadra aiuto sull'uso dell'applicazione del picking, per non fare brutte figure e sembrare più bravo.

Marta Rossi - donna 28 anni, vive in Senigallia, non ha mai finito gli studi universitari, madre con 2 figli di cui uno appena nato. E' alla sua prima esperienza lavorativa in zona, ha iniziato in magazzino un anno e mezzo fa e ora è esperta nel picking, sa cosa fare e come. Marta ha il turno pomeridiano, attacca dopo pranzo alle 15 e stacca alle 23. A causa degli orari, è affaticata. Ultimamente le capita a lavoro di dimenticare le cose o farle parzialmente, ha una bassa concentrazione. Marta è capace e efficiente nel suo lavoro, ma non ha le forze e la volontà per prestare più attenzione ai dettagli degli ordini. Gli errori più frequenti che sta riscontrando sono due: prendere una quantità di prodotto per un'altra e, a inizio prelievo, sbagliare la scelta di tipologia di udc da portare con sè (per metterci dentro la merce): ciò la costringe a tornare indietro a prendere uno nuovo, quando l' attività è stata già iniziata, facendole quindi perdere tempo e efficienza.

Gianluca Di Virgilio - maschio di 53 anni, vive fuori Milano, diplomato, famiglia con moglie e 2 figli. Ha appena cambiato lavoro dopo che la sua vecchia ditta presso cui lavorava ha chiuso, e ha trovato ripiego come magazziniere. E' abbastanza confidente con la tecnologia, ma con l'avanzare degli anni è meno avezzo a cambiare strumenti informatici, la curva di apprendimento diventa sempre più alta. Gianluca lavora di notte, le luci a led fredde e ambienti con poca luminosità gli stancano gli occhi che non sono più quelli di una volta. Anche la memoria inizia ad avere qualche acciacco, gli capita di dimenticare il dispositivo in giro e riguarda spesso le informazioni di prelievo, ma è scomodo farlo se si guida sistemi di trasporto. Infine, dopo tanti anni spesi in un lavoro di contatto umano, ora si ritrova ad aver a che fare con documenti e device mobili, il che lo mette a disagio. Ultimamente sta sbagliando molto, ed è additato da tutti. Sta già cercando un altro lavoro che sia più adatto a lui.

Dakarai Ouedraogo - maschio di 25 anni, di origine senegalese, non ha frequentato studi se non quelli di base, ha una famiglia nel suo paese di origine. E' da pochi anni in italia e sta imparando l'italiano, ha un forte accento straniero. Lavora da un anno in un magazzino alle porte di Palermo, e dopo molti mesi a lavorare nel reparto packing, ora è stato addetto al prelievo ma è bravo e capace e in poco tempo si è adattato. Le sue uniche difficoltà sono il dialogo con i colleghi, in quanto non ha completa padronanza della lingua, e ciò si ripercuote nel lavoro quando un superiore gli indica una mansione da fare.

Rosa De Marco - donna di 45 anni, vive a Bari, diplomata, divorziata. Lavora in un magazzino nella Zona industriale fuori città e deve arrivare a lavoro prima di tutti gli altri lavoratori, in quanto ha un ruolo tecnico ed è sua responsabilità la manutenzione degli apperecchi informatici. Ogni mattina deve assicurarsi che tutti i dispositivi siano funzionanti e in caso di problemi, reinstallare tutti i programmi software; quando capitano quelle giornate in cui tutto va storto e più device crashano, o più banalmente bisogna fare degli aggiornamenti, non si riesce mai a iniziare lavoro in orario e tutta la filiera ne risente con forti ritardi. Oltre a ciò, deve supportare anche i settaggi delle applicazioni e a ogni cambio turno modificarli quando passano di mano.

3.2 Analisi dei bisogni dell'utente: i requisiti

Analizzando le personas, sono state estratte le necessità degli utenti. Tenendo bene a mente le motivazioni iniziali della tesi, i bisogni sono stati riformulati formalmente in *requisiti*: punto di partenza dapprima della progettazione, e poi dell'implementazione.

Uno dei bisogni primari, ben riconoscobile nella persona di Gianluca, è la necessità di un'interfaccia che non impegni le mani, così da poterle tenere libere di fare altro, come maneggiare supporti cartacei, guidare attrezzature di trasporto (muletti), ecc. Il sistema sviluppato non deve essere di intralcio per chi lo usa (ma allo stesso tempo non deve essere dimenticato) e inoltre deve essere funzionante ovunque siano posizionati gli operatori nel magazzino. Per questi motivi, il progetto prevederà un sistema che si adatti alle dinamiche del lavoro, che dunque avrà: un'interfaccia *multimodale* che permetta all'utente di interagire vocalmente/gesti della mano, e ricevere informazioni visivamente e/o con l'udito, in modo da supportarlo durante le sue attività anche se *in movimento*. Le funzionalità dovranno essere sempre accessibili in tutta l'area del magazzino: dovranno perciò essere implementate *offline* per non necessitare di linea internet, ballerina in ambienti così grandi. Sarà indispensabile in tal senso l'impiego di un motore vocale che sia capace di capire *accenti* diversi, per supportare lavoratori stranieri (come Dakarai) o nazionali ma con forte accento locale.

In aggiuntà è importante non stressare la vista (Gianluca) per raggiungere condizioni di lavoro più sostenibili grazie a occhi meno stanchi: in parte questa necessità è risolta da un'interfaccia multimodale che privilegi l'interazione vocale durante i movimenti (la maggior parte del tempo), ma è importante anche un'interfaccia grafica che *si adatti* alle condizioni ambientali di lumonisità.

Inoltre, per andare incontro alle necessità di tutti i possibili utenti, l'interfaccia non deve introdurre elementi inutili e complicati che potrebbero essere un deterrente per l'utilizzo del sistema: deve risultare *semplice* da usarsi, anche per quelle persone non abituate al contatto continuo con la tecnologia o a disagio nell'impararne una nuova (Gianluca). Tutti i dispositivi introdotti (possibilmente nessuno) devono essere facilmente portabili, plug & play (Rosa), senza procedure "di start" come potrebbe esserlo l'addestramento del motore vocale alla voce specifica dell'utilizzatore. L'usabilità si pone come fattore imprescindibile, e le interfacce dovranno essere quanto mai semplici e *intuitive*: chiunque dovrebbe poterla usare senza pensarci su (Marta) [14].

Infine, un sistema che dia confidenza di sè all'utilizzatore si configura come bisogno principale di tutti gli operatori, soprattutto se addetti da poco al picking (Mario). Le motivazioni sono sia di carattere pratico, in quanto meno errori significa maggiore produttività e efficienza, sia di carattere psicologico, poichè sentirsi sicuri del proprio lavoro si traduce in livelli di stress e ansia minori e quindi maggiore soddisfazione e felicità. In tale ottica, risulta fondamentale un sistema di *prevenzione errori*, un'esperienza d'uso che offra metodi semplici e veloci di error recovery, azioni sempre reversibili. C'è bisogno quindi di un sistema che riassumi quanti colli prendere, per guidare la scelta dell'udc da parte dell'operatore a inizio attività, e che eviti sviste su ciò che è prelevato (Marta).

3.3 I requisiti da soddisfare

I requisiti nascono dai *bisogni* dell'utente e dagli *obiettivi* che questa tesi si pone (descritte nell'introduzione): sintetizzano sia proprietà/vincoli richiesti al sistema, sia le funzionalità necessarie a fronte di particolari input e situazioni.

Essendo numerosi e di tipologia diversa, per facilità di consultazione sono riassunti nella tabella 3.1 qui di seguito. I requisiti sono raggruppati per macro aree, e ognuno di essi ha una relativa *priorità*, misurata su una scala che va da 3 (priorità minima) a 1 (priorità massima): nelle fasi successive di progettazione e implementazione del sistema se ne farà riferimento per tenerli ben presenti e non allontanarsene. Nel capitolo 9 si andranno poi ad analizzare i risultati raggiunti nei test con utenti reali, e confrontandoli con i requisiti prefissati, si potrà avere un riscontro sulla buona riuscita o meno del lavoro di tesi.

Categoria	Numero	Descrizione	Priorità
Usabilità	R 1.1	Il sistema deve permettere ai pickeristi di poter interagire con il proprio device mobile senza impegnare le mani durante i movimenti in magazzino, in modo tale da poter maneggiare veicoli e caricare gli articoli senza impedimenti.	1
	R 1.2	Il sistema non deve essere di intralcio nel lavoro quotidiano degli operatori, ma allo stesso tempo non deve essere dimenticato in giro.	2
	R 1.3	Il sistema deve essere facile da utilizzare per chiunque e deve fornire un'interfaccia più naturale possibile, intuitiva e semplice, in modo da non penalizzare l'utilizzo.	2
	R 1.4	Il sistema non deve affaticare gli occhi.	3
Supporto per i lavoratori di magazzino	R 2.1	Il sistema deve permettere al pickerista di portare al termine il processo con il minor numero possibile di errori, per raggiungere un'elevata accuratezza.	1
	R 2.2	Il sistema deve permettere al pickerista di decidere, in autonomia, la tipologia di udc da prendere, in modo che sia adatta a contenere tutti gli articoli da prendere nella missione di prelievo intrapresa.	2
	R 2.3	Il sistema deve permettere tempi di completamento prelievo rapidi.	1
	R 2.4	Il sistema deve riconoscere in automatico gli errori senza una esplicita richiesta, avviando le relative procedure di gestione errore.	3
	R 2.5	Il sistema deve essere in grado di ridurre gli errori commessi al minimo, facendo opportuna prevenzione.	2
	R 2.6	Il sistema deve permettere tempi veloci di formazione, grazie a procedure guidate e istruzioni semplici, di facile comprensione.	3
	R 2.7	L'installazione dell'applicazione deve essere "plug&play", pronta per essere subito usata.	2
Prestazioni e Efficienza	R 3.1	Nell'ottica di mantenere bassi i costi, il sistema deve supportare Android e non richiedere hardware specifico a bordo. In altre parole, il sistema deve poter girare su un device mobile android di tipo consumer.	2
	R 3.2	Il sistema deve essere in grado di supportare anche chi non ha una buona padronanza della lingua. Il text-to-speech deve essere facilmente comprensibile, e allo stesso tempo il speech-to-text reattivo e accurato.	1
	R 3.3	Il sistema deve impiegare le risorse del device mobile in modo efficiente	1
	R 3.4	Il sistema deve poter funzionare offline, quando ci si sposta in magazzino per il prelievo.	1

Tabella 3.1: Requisiti del Progetto

Capitolo 4

Scouting e Selezione: Motore vocale

In questo capitolo viene trattata la ricerca e selezione del motore vocale di riferimento. In particolare, sono state dapprima *raccolte* e catalogate tutte le soluzioni vocali attualmente disponibili in commercio per il sistema operativo Android, poi *filtrate* sulla base dei requisiti delineati nel capitolo precedente, infine *confrontate* in modo analitico per scegliere il motore più adatto agli scopi. Prima di parlare delle metodiche usate, bisogna però capire quali sono le possibili caratteristiche di un motore vocale e quali di queste quelle preferibili.

4.1 Caratteristiche desiderate

4.1.1 Caratteristiche di un motore vocale

La soluzioni vocali in commercio possono trovare una prima distinzione nelle funzionalità che offrono: esistono infatti soluzioni solo speech-to-text, solo text-to-speech, e in alcuni casi entrambi. Dovendo creare una libreria con funzionalità sia STT che TTS, senza precludersi nessuna potenziale alternativa, non ci si è limitati solo a soluzioni "finali" come quelle del terzo tipo, ma si sono considerate anche soluzioni *solo* STT o solo TTS, con la prospettiva di *combinarle* in un unico motore vocale al momento dell'integrazione nel framework. La ricerca è stata basata sulle seguenti proprietà di una soluzione vocale:

Componenti Esterni - potrebbe essere richiesta l'installazione *manuale* sul dispositivo di componenti terze, allo scopo di abilitare l'uso del motore vocale in applicazioni custom;

Tipo di vocabolario - il vocabolario può essere *espansibile* o *chiuso*, *limitato* a poche decine di parole (tavolta anche meno di una decina) oppure *molto ampio*, a seconda degli scopi per cui è stato creato: se infatti il motore è pensato per l'implementazione di un assistente vocale, sarà necessario un esteso pool di parole da usare nei N possibili scenari, ma se invece le casistiche di uso sono ridotte, allora basterà un ristretto set di parole. Tenendo a mente che le performance del speech-to-text sono *inversamente* legate alla grandezza del vocabolario, si può ben capire come la tipologia vocabolario chiuso sia adottata da molti motori vocali comunemente impiegati nel settore logistico;

Linguaggio Naturale - (solo STT) alcuni motori vocali potrebbero essere in grado di *comprendere il linguaggio* umano e fornire un output di conseguenza, grazie all'elaborazione del linguaggio naturale (NLP). E' quello che succede ogni giorno quando, senza pensarci troppo, inviamo comandi vocali ai nostri assistenti domestici virtuali come Alexa, Siri e Google Assistant, i quali rispondono alle nostre domande, aggiungono attività ai nostri calendari e chiamano i contatti richiesti nei nostri comandi vocali;

Allenamento - (solo STT) prima di iniziare il processo di riconoscimento, alcune soluzioni vocali *prevedono* un breve momento di allenamento per adattarsi alle caratteristiche vocali dell'utente, al suo accento: può durare da 1 a 30 minuti;

Connettività - le funzionalità di traduzione voce/testo e viceversa potrebbero essere implementate *in locale* o *"in cloud"*; da ciò dipende il tipo di connettività richiesta al dispositivo: nel primo caso non sarà necessaria (si parla di servizi "*offline*"), ma nel secondo si;

Licenza - è la licenza software a stabilire i termini d'uso del motore vocale, i limiti di responsabilità, il trasferimento dei diritti, ecc. Di queste, la tipologia "open source" è quella che permette di usare, copiare, modificare, ampliare, vendere liberamente, senza imporre obblighi al ricompenso economico degli autori.

4.1.2 Caratteristiche e requisiti a confronto

Per sapere quali parametri considerare sufficienti nella ricerca, i requisiti definiti (sezione 3.3) sono stati valutati sulla base delle possibili caratteristiche di un motore vocale, descritti al paragrafo precedente. In particolare si possono individuare proprietà "necessarie" e altre "preferibili":

Proprietà	Requisiti	Valore Desiderato	Descrizione
Componenti Esterni	R2.7	Non presenti	E' *necessario* un motore vocale che non richieda componenti esterni da installare e aggionare.
Vocabolario	R3.2	Limitato/Espandibile	Agli scopi del progetto è *preferibile* un vocabolario espansibile, per supportare eventuali richieste non previste al momento della creazione, e limitato, in quanto un minor numero di parole significa anche maggiore accuratezza e "resilienza" a accenti/rumore ambientale
Linguaggio Naturale	R3.3	Non presente	I motori vocali capaci di elaborare il linguaggio naturale sottointendono computazioni non banali, allo scopo di gestire input variabili. Tutto ciò è *preferibilmente* non necessario. I possibili input in ambito logistico sono definiti a priori e oltretutto un motore vocale più semplice, sotto questo punto di vista, ha il vantaggio di avere latenze mediamente meno elevate.
Allenamento	R2.7	Non necessario	In virtù di un'esperienza "plug&play", non *deve* essere richiesta una fase di allenamento vocale allo startup del sistema
Connettività	R3.5	Offline	Per permettere un uso continuo e performante all'interno del magazzino, le funzionalità vocali *devono* essere disponibili anche lì dove la linea internet potrebbe essere assente.
Licenza		Open Source	E' *preferibile* un motore vocale che non richieda permessi d'uso e che sia aperto a personalizzazioni, in modo da non rallentare il lavoro di tesi e sfruttarne al massimo le capacità.

Tabella 4.1: Caratteristiche desiderate del motore vocale

4.2 Metodo di ricerca

La ricerca di soluzioni vocali disponibili sul mercato è stata condotta *online*, usando come motore di ricerca di riferimento Google. La durata di questa indagine è durata un totale di *due settimane* ed è stata composta da due fasi, ciascuna di una settimana: la prima in cui si sono *cercati* più riferimenti possibili a librerie (in inglese Software-Development-Kit, SDK) con funzionalità vocali; nella seconda si

sono invece *raccolte le informazioni* relative alle librerie trovate, soffermandosi in particolare sulle loro caratteristiche, giudicandone il livello di *idoneità*.

La prima fase è stata gestita digitando sulla barra di ricerca risultati del tipo "Android SDK Speech Synthesis", "Android SDK Speech-To-Text", "Android SDK Text-To-Speech", "Voice Android SDK" o ancora "Framework Android Speech Synthesis"; da qui seguite le voci più pertinenti e annotati il nome e indirizzo url delle librerie trovate, senza dare più di una occhiata alle informazioni. Le caratteristiche sono poi state raccolte nella seconda fase, molto più sistematica rispetto a quella precedente, grazie al supporto di una *matrice* realizzata su un file Excel chiamata "*Sommario*".

Quest'ultima è stata realizzata mettendo sulle *righe* i motori trovati (uno per riga), e sulle *colonne*: tipo di funzionalità (STT e/o TTS), le caratteristiche (elencate nella sezione 4.1), sito web, un booleano che rappresentasse la possibilità di prova gratutita e infine un campo a testo libero "descrizione" per riportare le impressioni avute su prestazioni, flessibilità, compatibilità. Per favorire una rapida consultazione della tabella, la si è divisa in schede in base alla funzionalità (solo STT, solo TTS, entrambe) e si sono scelti dei colori a rappresentazione del livello di "idoneità" di una caratteristica: verde significa *idonea*, rossa *non* idonea, arancione invece *preferenza non rispettata*. Una sola caratteristica rossa significa motore vocale *non adatto*. I motori adatti sono stati portati nelle prime righe. Il risultato visivo può essere apprezzato in figura 4.1, dove è riportato un estratto dal file "*Sommario*".

OVERVIEW					REQUIREMENTS						LIN	
Data Bael	Solution (ASR)	Capabilities	OpenSou	Description	SDK android	Multilanguage	Training required	Vocabolario espandibile	Trial	Offline	Sito1	Sito2
apr-22	Vivoka	STT • TTS • NLU	Nope	demo -> https://www.youtube.com/watch?v=0l4Dg9D9GTs&ab_channel=Vivoka, it also allows voice-based authentication, general purpose, great versatility and customization possibilities like german autolabs through its IDE	Yes	Yes	Yes	Yes	Yes	Yes	AI Voice Technologies inside one powerful SDK for Companies – Vivok	Free Evaluation Period – Vivoka
apr-22	Android Recognizer(Native Android)	STT • NLU • TTS	Nope	https://www.tutorialspoint.com/how-to-integrate-android-speech-to-text	requires google services	Yes	No	Yes	Yes - https://github.com/mardenmadoli/speech-to-test	Yes	GitHub - livinoweb/continuousSpeechRecognit	hiteshsahu/Android-TTS-STT: One line solution for Android Text to
apr-22	IBM Watson Speech to Text	STT • TTS • NLU	Nope	X	X	X	X	X	X	No		
apr-22	Microsoft Cognitive Services	STT • TTS • NLU	Nope	X	X	X	X	X	X	No		
apr-22	Dialogflow (Google)	STT • TTS • NLU	Nope	The Dialogflow service itself cannot be deployed on-premises, however, your on-premises applications can access the Dialogflow cloud service. You can also host a webhook service on-premises, but it must be accessible through a public URL.	X	Yes - 14 languages	X	X	X	No	FAQ | Dialogflow Documentation | Google Clou	
apr-22	AndroidMaryTTS	TTS	Yes	based on https://github.com/marytts/marytts	yes - 10 langauges	No	Yes	No	GitHub - AndroidMaryTTS/AndroidMaryTTS: Android MARY TTS - an open-source, offline HMM-Based text-to-speech			
apr-22	ResponsiveVoice Text To Speech API	TTS	X	X	yes - 51 languages	X	X	No				
apr-22	ReadSpeaker	TTS	X	Very general purpose and with different types of software. From what I understand you can manually create audio files with ReadSpeaker speechMaker and then add them to the app. not very versatile.	110 voci in oltre 35	No	Yes	Yes	Demo Vocalizzazione Interattiva - ReadSpeaker	ReadSpeaker speechMaker ReadSpeaker		
apr-22	Voicepods	TTS	Nope	It allows you to add different kinds of "feelings" to the synthesized text	yes - 24 voices	X	Yes	No	Features ~ Voicepods			
apr-22	CloudPronouncer API	TTS	X	X	yes - 31 languages	No	X	No				
apr-22	Mimic	STT	X	X	X	X	X	No				

Figura 4.1: Estratto dal file Excel "Sommario"

4.3 Selezione del Motore Vocale

Dopo aver catalogato e classificato ben 21 motori vocali, la cui lista è consultabile in appendice A.2, il naturale passo successivo è stata la selezione delle soluzioni idonee da essere *integrate* nel sistema. Specificatamente, si sono considerate idonee le soluzioni vocali che *più* rispettassero le caratteristiche desiderate in tabella 4.1. Ma come gestire soluzioni solo speech-to-text o solo text-to-speech? Per ovviare a questo problema si sono considerate ipotesi di "*soluzioni complete*", costituite sia da librerie integranti da subito *entrambe* le funzionalità, sia da *combinazioni* di librerie solo STT con altre solo TTS.

Alla luce di quanto detto, sono state selezionate tre soluzioni idonee e da queste ipotizzate tre soluzioni *complete* riportate in tabella 4.2. Da sottolineare che il grosso discriminante di questa "scrematura" è stato il possesso di una licenza open source o equivalentemente l'opportunità di una prova gratuita a tempo indeterminato.

		Vocabolario	No Linguaggio Naturale	No allenamento	Connettività Offline	No Componenti Esterni	Licenza OpenSource	Periodo di Prova
Motori Vocali	Android Native	molto grande	✓	✓	✓		✓	
	Kaldi (STT) + A.Native (TTS)	limitato/espansibile	✓	✓	✓	✓	✓	
	Vo-ce ITWorks	limitato/espansibile	✓	✓	✓			✓

Tabella 4.2: Motori vocali selezionati

4.3.1 Metodo di confronto

Le prestazioni del motore vocale da implementare influenzeranno inevitabilmente quelle dell'intero sistema. Ricordando i requisiti R3.2 e R3.3 in tabella 3.1, vale la pena porre da subito l'accento su questi aspetti. Ma come *misurare* le prestazioni di una libreria software in modo sistematico?

Se da una parte abbiamo delle prestazioni direttamente misurabili come *reattività*, *numero di errori*, dall'altra dobbiamo anche considerare quelle lato hardware: non è riduttivo semplificarle in termini di *consumi risorse* CPU, RAM, batteria. Fortunatamente, è possibile collezionare in modo semplice queste misure su Android

grazie al tool "Profiler" offerto da Android Studio[1], capace di calcolare in tempo reale il *dispendio di risorse* causato da una sola applicazione. Con quest'ultimo è possible monitorare da una singola dashboard (in figura 4.2) tutti i parametri relativi a CPU, RAM, batteria del dispositivo.

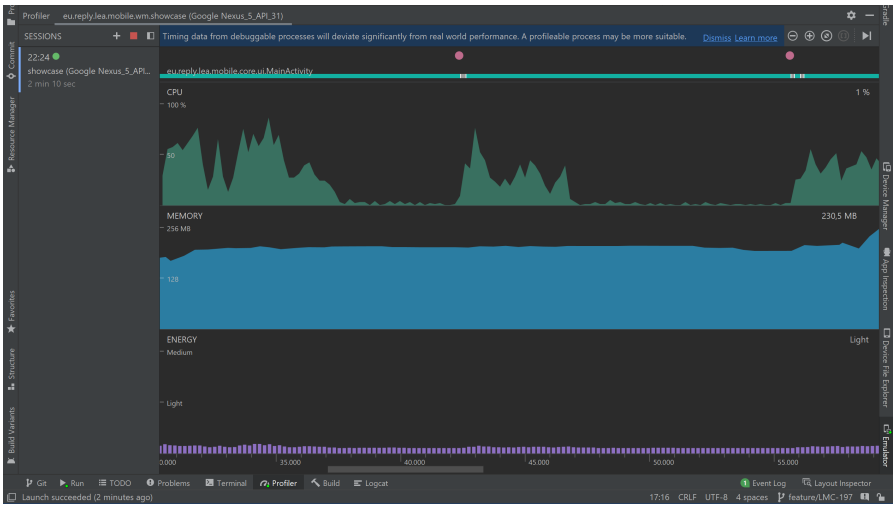

Figura 4.2: Dashboard del Profiler

Avendo a che fare però con librerie da implementare, e non di applicazioni fatte, l'idea in questo caso è stata quella di misurarne i dispendi a "*parità*" di app: se nell'applicazione il codice sorgente rimane uguale, ma cambiano solo gli endpoint a cui arrivano le chiamate API, i dispendi di risorse potranno essere direttamente confrontati in termini assoluti, senza curarci di considerare tutti i fattori esterni alle librerie implementate, poichè possiamo considerali, con buona approssimazione, "*annullati a vicenda*"!.

E' stata dunque realizzata un'unica applicazione Android "usa e getta" molto semplice, a partire dal progetto base "Empty Project" disponibile su AndroidStudio, che integrasse un motore vocale x intercambiabile (di volta in volta sostituito). L'unica interfaccia dell'app, riportata in figura 4.3, è dotata di due pulsanti e un campo di testo aperto non editabile direttamente: schiacchiando il primo bottone è possibile convertire il parlato e quindi leggere, ciò che si pronuncia, nel campo di testo, mentre il secondo permette di ascoltare quanto scritto.

[1]Android Studio è l'ambiente di programmazione ufficile (IDE) per lo sviluppo di applicazioni Android, creato da Google e basato su IntelliJ IDEA

4.3.2 Risultati

Dopo aver creato tre implementazioni diverse dell'applicazione, misurato il *dispendio* di risorse calcolato su un utilizzo medio, infine valutato *reattività* e *numero errori medio*, si sono potuti quindi confrontare complessivamente i risultati (in tabella 4.3) per capire quale delle tre soluzioni vocali avesse le prestazioni migliori. I test sono stati effettuati con l'ausilio dell'emulatore di AndroidStudio, configurato sulle caratteristiche (in figura 4.4) di un Nexus 5 API 31.

Da quest'ultima appare chiaro come il framework Vo-ce di ITWorks, pensato e creato per la logistica, sia più performante rispetto a una soluzione general purpose solo Android Native, la più carente delle tre, ma anche di quella mista Kaldi(STT) + Android Native (TTS). Considerando però anche le caratteristiche dei motori riportate in tabella 4.2, la resa prestazionale è stata valutata come "male minore" rispetto a caratteristiche *meno* aderenti a quelle necessarie a soddisfare i requisiti delineati. Sulla base di queste motivazioni, è stata dunque scelta la seconda soluzione, costituita dalla libreria speech-to-text Kaldi e il modulo di Android text-to-speech nativo, cioè *preinstallato* su tutti i dispositivi con a bordo questo sistema operativo.

		%CPU	RAM (MB)	Uso Batteria	Reattività	%Numero errori
Motori Vocali	Android Native	0.03	128	Bassissima	Media	0.15
	Kaldi (STT) + A.Native (TTS)	0.05	200	Bassa	Alta	0.10
	Vo-ce ITWorks	0.08	100	Bassa	Altissima	0.7

Tabella 4.3: Risultati test su motori vocali

4.4 Il Motore Vocale

Il motore vocale selezionato nello scouting offre sia funzionalità speech-to-text che text-to-speech. Nasce dall'unione di due soluzioni distinte ma compatibili tra loro: Kaldi per la parte di riconoscimento vocale, e in modo complementare Android nativo per quella di ouput vocale a partire dal testo.

4.4.1 Kaldi: Speech-to-Text

Kaldi [15] è un framework OpenSource che fa parte della suite Vosk, gestita dall'organizzazione Alpha Cephei. Funziona anche offline, è compatibile con Linux

Figura 4.3: Interfaccia grafica dell'app "usa e getta"

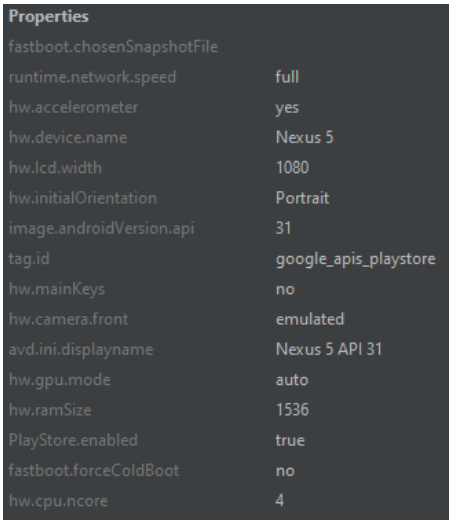

Figura 4.4: Proprietà emulatore usato nel test di confronto

e Windows, ma anche disponibile su device "pesopiuma" come Raspberry Pi, Android, iOS. Esistono modelli di lingua già pronti, supporta infatti più di 20 lingue e dialetti, ma è anche possibile riconfigurarne di nuovi per migliorare l'accuratezza del riconoscimento vocale.

Per la realizzazione del progetto "usa e getta" già citato, ci si è basati sulla demo Android presente su Github [16]. Da quest'ultima è stato importato il modulo integrante la libreria Kaldi (in figura 4.5), e con essa il modello di lingua italiana "model-it"[2] allenato dalla community di Vosk.

Nel modulo possiamo trovare il file *build.gradle* necessario per l'automazione della compilazione del package Android, la cartella *model-it* contente i file necessari al riconoscimento vocale e infine il *manifest Android* usato per descrivere le informazioni essenziali sul modulo e permettere quindi al sistema operativo di far girare il codice.

[2]Il modello della lingua italiana è scaricabile dalla pagina di Alpha Cephei al link `https://alphacephei.com/vosk/models/vosk-model-small-it-0.22.zip` e occupa appena 48MB

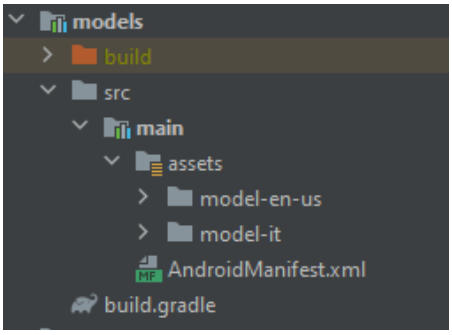

Figura 4.5: Visione dell'organizzazione del modulo Kaldi

4.4.2 Android Native: Text-to-Speech

Questo modulo [17] è installato di default con il sistema operativo Android, per supportare le funzionalità base dello stesso come quelle di accessibilità per disabili. Essendo parte di Android, non è necessario copiare e incollare nessun package all'interno dell'applicazione e basterà semplicemente dichiararla tra gli "import": ci penserà Android a fare *dinamic linking* della libreria a compile time. Non è richiesta quindi nessuna installazione manuale, tuttavia è necessario scaricare il pacchetto della lingua usata (nel nostro caso l'italiano) per sfruttarne la funzionalità anche offline. A tal scopo è possibile gestire le lingue installate sul dispositivo, tonalità e velocità di sintesi vocale, andando in Impostazioni>Lingua e Inserimento>Sintesi Vocale.

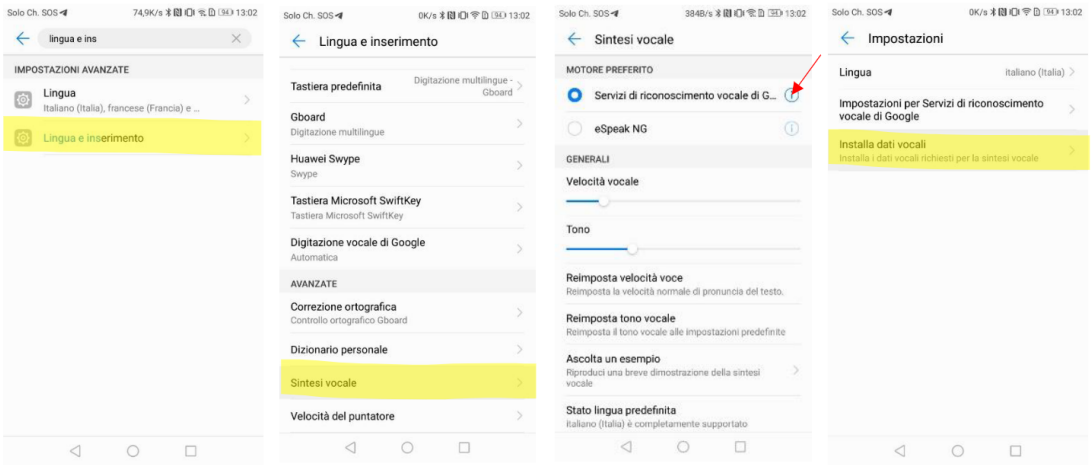

Figura 4.6: Percorso di installazione pacchetti lingue offline

4.4.3 Vantaggi e Svantaggi

I vantaggi offerti da questa soluzione sono: il *supporto* da una vasta comunità online, possibilità di reperire facilmente la *documentazione* in rete, possibilità di *personalizzazione* grazie all'opensource, possibilità di creazione di nuovi *modelli di lingua* personalizzati, *licenza* di uso gratis a tempo indeterminato. Di contro, gli svantaggi: è richiesta una versione di Android che integri i *Google Play Services* per poter usare Android STT nativo, è richiesta l'*installazione manuale* dei pacchetti di lingua offline (da fare però solo una sola volta).

Capitolo 5

Progettazione e Prototipazione

In questo capitolo si descrive la progettazione del sistema, regina nel lavoro di tesi. Questa fase si pone come scopo la realizzazione di un sistema efficiente e in linea con i requisiti precedentemente delineati nel capitolo 3, mantenendo quindi il focus sugli utenti e i loro bisogni. La scelta di questo approccio *user centered* permetterà di disegnare un'interfaccia che sia al tempo stesso gradevole, coerente, facile da apprendere e funzionale, da usare cioè in contesti operativi reali: in poche parole *usabile*.

Per fare ciò, verrà prima studiato il processo di prelievo manuale *solo* mobile, in modo da ridisegnarlo e ottimizzarlo attraverso l'impiego della tecnologia vocale, poi descritta l'architettura generale, con attenzione alle componenti da scegliere e la sinergia da costruire, infine esposti i passaggi di prototipazione che hanno portato alla scelta dell'interfaccia utente.

5.1 Processo di prelievo manuale

Nella suite di applicazioni di magazzino (gestionale WMS) di Reply, era stata già stata in precendenza sviluppata un'applicazione di supporto per il prelievo, ma in versione *solo* mobile. Per motivi di *compatibilità* con l'infrastruttura sottostante il modulo applicativo (il backend), non di mia competenza, si è deciso di partire dal vecchio processo, e quindi ridisegnarlo al fine di ottimizzarlo e adattarlo alle funzionalità vocali. Il segreto industriale impedisce di descrivere i dettagli del processo *as is*, però può essere presentato un diagramma di flusso operazionale

BPMN[1] semplificato che lo riassume, in figura 5.1.

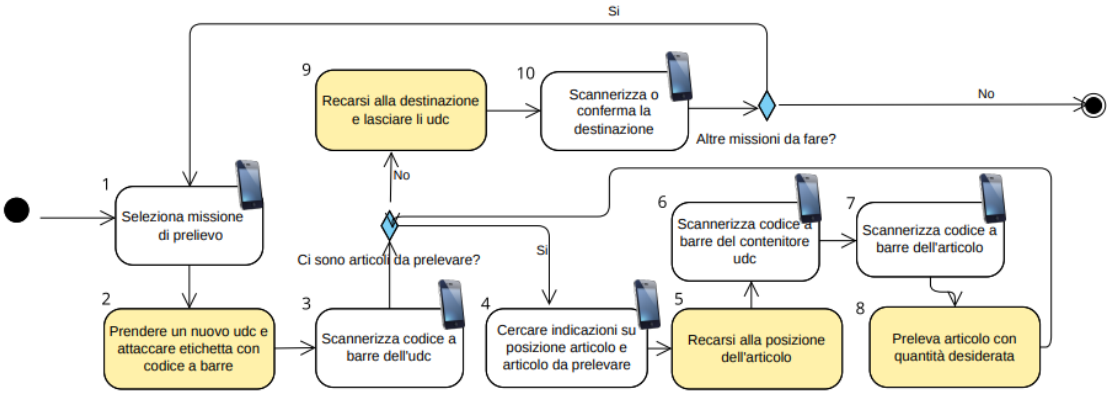

Figura 5.1: BPMN del processo di prelievo *solo* mobile

Nel grafico si possono notare diversi tipi di unità operative: le *azioni fisiche*, in giallo, e *azioni digitali* da eseguire invece con il mobile. La progettazione del nuovo processo riguarderà esclusivamente quest'ultime, in quanto non si vuole modificare il workflow operativo di base richiesto in magazzino.

Iniziando per ordine dall'azione#1, non è necessario che questa venga modificata e adattata al vocale, in quanto è previsto che la selezione della missione di prelievo venga fatta *da fermi*, che sia la prima della giornata oppure l'n-esima (R1.1). Tuttavia, si è anche valutato di usare il vocale, che però risulterebbe poco adatto in questa situazione: non essendo la lista di missioni di lunghezza ben definita, potrebbe essere tediosa descriverla tutta e per il pickerista ricordarne i dettagli per una scelta finale che sia ponderata.

Tra la l'azione 1 e l'azione 2 potrebbe esserne aggiunta un'altra, per aiutare la scelta della tipologia di udc da prendere per iniziare il prelievo (R2.2). In particolare il sistema dovrà fornire *informazioni utili* (quantità, dimensioni merce) a far scegliere il tipo di scatola più adatta a contenere tutti gli articoli da prelevare nella missione iniziata.

Per quanto riguada lo step successivo#3, non potrà più prevedere un barcode scanner, in quanto si tratta di hardware specifico, in questa tesi non supportato (R3.1). L'azione, anche se non eseguita in movimento, potrà essere eventualmente adattata al speech-to-text, dando la possibilità di *pronunciare il barcode* in input, con opportuni metodi di *roll-back* in caso di errore.

[1]Il Business Process Modeling Notation (BPMN) è un metodo per diagrammi di flusso che modella dall'inizio alla fine le fasi di un processo aziendale pianificato.

L'azione seguente #4, è fortemente legata a un'interfaccia grafica in cui cercare le informazioni utili: sarà *ottimizzata* per un'esperienza più fluida e naturale (R.1.3) e *spezzata* in sotto-azioni per renderla più semplice da seguire e permettere al pickerista di concentrarsi su un problema per volta (R2.6). Sarà inoltre opportuno prevedere un sistema di controllo per accorgersi di un eventuale *errato posizionamento*, se ad esempio non si conoscono bene le zone del magazzino. A tal proposito si sono valutate due soluzioni diverse, ma entrambe robuste allo stesso modo:

- un check di tipo *ripeti posizione* da parte dell'operatore, che consiste nel confrontare quanto dice con l'effettiva ubicazione di magazzino da raggiungere;
- un check basato su *cifre di controllo*, ossia una stringa numerica di lunghezza variabile da 1 a 4 caratteri, assegnate randomicamente[2] a una posizione, stando però attenti che le stesse non siano già state assegnata a un'altra nelle vicinanze.

Poichè le posizioni potrebbero raggiungere lunghezze non banali, e pronunciare stringhe troppo lunghe significherebbe rallentare il processo, la prima possibilità è stata scartata a favore della seconda.

Continuando la sequenza, anche le azioni #6 e #7 dovranno essere modificate per gli stessi motivi della #3: per velocizzare il prelievo, non si dovrà più sparare il codice a barre di udc sorgente e articolo, bensì semplicemente ascoltarli e dare un feedback al sistema quando si sono trovati visivamente nello scaffale. In particolare, dal momento che memorizzare e cercare stringhe alfanumeriche potrebbe essere difficoltoso, *dividere i codici* in sottostringhe (da ascoltare un pezzo per volta) sarà utile a prevenire errori di comprensione e agevolare in generale la ricerca.

Per quanto riguarda invece l'azione #8, vale la pena riconsiderarla sotto il punto di vista della prevenzione (R2.5). E' possibile difatti prelevare una quantità di prodotto diversa da quella richiesta, ma ci sono casi limite da distinguere:

- la quantità di prodotto prelevato *può* essere minore rispetto a quella aspettata, se non è disponibile in giacenza la quantità richiesta (errore di inventario). Si procederà a raccogliere gli articoli presenti per soddisfare quanti più ordini possibile;
- la quantità di prodotto prelevato non può *in nessun caso* essere maggiore.

Il sistema dovrà distinguere i casi autonomamente, ed attivare le relative contromisure a supporto del pickerista. Infine, l'operatore dovrà ricevere indicazioni sulla

[2]Le cifre di controllo possono essere create utilizzando programmi randomici, oppure con trasformazioni numeriche a partire da numeri primi e valori ASCII di lettere e numeri della posizione.

destinazione da raggiungere, generalmente sempre la zona di imballaggio. Essendo una meta *comune* e *ricorrente*, non si ritiene opportuno aggiungere eventuali controlli, che appesantirebbero l'attività di prelievo. La schematizzazione del nuovo processo, così come appena descritta, può essere osservata in figura 5.2.

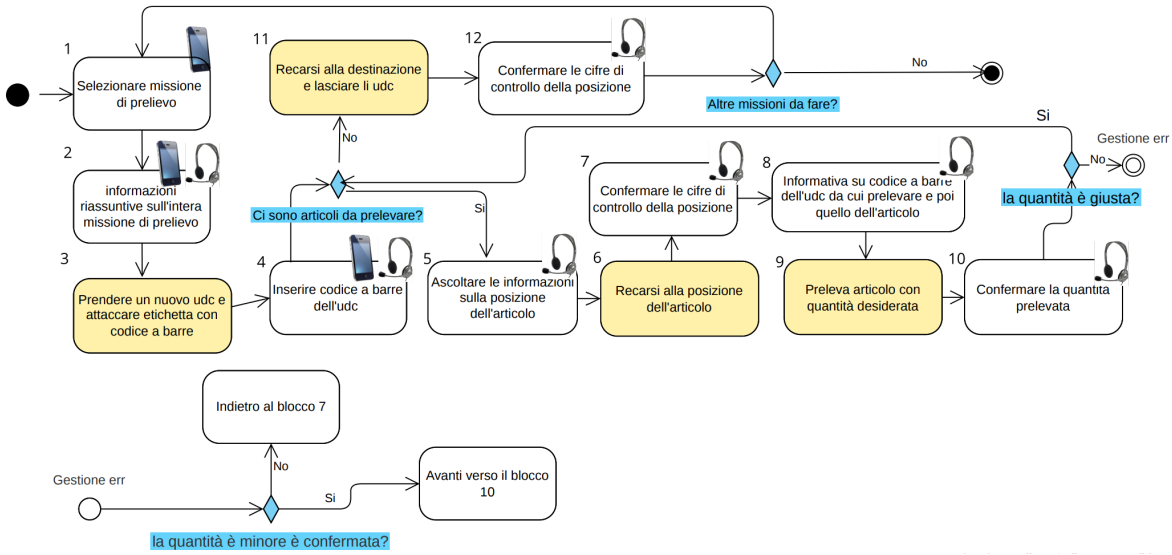

Figura 5.2: BPMN del processo di prelievo vocale

5.2 Architettura generale della soluzione

Dopo aver ridisegnato il processo in versione vocale, ci si è contrati sulla definizione dell'architettura della soluzione. Nella prima parte di progettazione, si è cercato di capire quali componenti *fisici* fossero utili al mobile, per l'ergonomia del sistema.

Per ottenere un sistema che non sia di intralcio nel lavoro di un pickerista (R1.2) e che allo stesso tempo permetta di interagire durante i movimenti in magazzino senza impegnare le mani (R1.1), è necessario che il sistema possa essere messo in un posto *comodo*, ma allo stesso tempo *vicino*, in modo da consentire un'interazione uomo-macchina prettamente vocale/sonora. Questi due requisiti sono in naturale contrasto tra loro, in quanto è naturale appoggiare il dispositivo sul muletto, o sullo scaffale, o ancora metterlo in tasca: la comunicazione sarebbe però di certo disturbata da eventuali rumori e le prestazioni fortemente condizionate dalla distanza del microfono/speaker. Considerando il contesto di un magazzino, potenzialmente un ambiente molto rumoroso, e l'impossibilità ad avvicinare con le mani il dispositivo alla bocca/orecchie durante l'uso, si rende necessaria la dotazione di cuffie senza filo, per parlare e ascoltare.

Quest'ultime dovranno comunicare con Android usando un protocollo più possibile standard, che non richieda hardware specifico a bordo (R3.1). La scelta naturale è quindi ricaduta sul Bluetooth, lo standard tecnico-industriale supportato da pressochè tutti i dispositivi mobili, capace di offrire un metodo economico e sicuro per scambiare informazioni tra dispositivi diversi attraverso una frequenza radio sicura a corto raggio, di circa 10 metri, più che sufficiente per lasciar lavorare il nostro pickerista libero da ingombri.

Analizzando inoltre i scenari d'uso appena citati, ci si accorge subito delle possibili ottimizzazioni (R3.3): sapendo quando lo schermo non è osservato, potremmo abbassare i consumi di batteria spegnendolo. Ma come fare? La prima possibilità considerata è stata far si che fosse lo stesso pickerista a informare il sistema attraverso un gesto (un tocco, o la pressione di un tasto), la seconda invece sfruttare il sensore di prossimità[3]. Quando infatti il cellulare è posto in tasca, oppure su una qualsiasi superficie con lo schermo rivolto verso il basso, si può assumere che il suo schermo non sia visualizzato dall'utente in quanto nascosto: in entrambi i casi il sensore di prossimità si attiverà, tornando disattivo quando il mobile sarà ripreso in mano e lo schermo nuovamente visibile. E' stata scelta la seconda alternativa sia perchè la prima richiede un costante livello di attenzione da riservare al sistema, e ciò incide negativamente sull'usabilità, sia perchè offre un ulteriore vantaggio di prevenzione errore (R2.5): se il dispositivo è posto in tasca, non si corre il rischio di tocchi accidentali sullo schermo.

Per quanto riguarda invece la comunicazione client-server tra operatore e WMS, non ci si è preoccupati di progettare da sè il *backend*, in quanto l'infrastruttura da usare era già esistente (gestionale di Reply), come da obiettivo. Il sistema, dal punto di vista hardware, può essere schematizzato visivamente in un grafilo UML, come in figura 5.3.

Per quanto riguarda invece l'architettura *software*, i componenti erano già stati definiti in partenza negli obiettivi di tesi, e così riassunti: il *motore vocale*, il *framework* interfaccia per disaccoppiare le logiche, il *modulo* di prelievo vocale e infine l'*applicazione* mobile di Reply con cui il progetto dovrà integrarsi. La domanda più difficile a cui rispondere è stata senza dubbio come generalizzare e astrarre il framework vocale per *disaccoppiare* l'applicativo dal motore vocale implementato, e permettere un cambio di quest'ultimo nel modo meno impattante possibile per l'ecosistema generale. La schematizzazione è visibile in figura 5.4.

[3]Il sensore di prossimità è un sensore capace di rilevare ostacoli. E' generalmente posizionato in alto nella parte anteriore dei dispositivi mobili ed è usato per capire quando si sta avvicinando lo smartphone all'orecchio, in modo da spegnere il display e evitare tocchi accidentali.

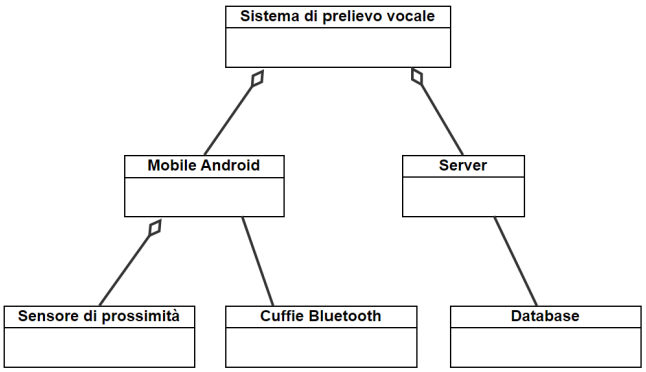

Figura 5.3: Architettura hardware del sistema vocale

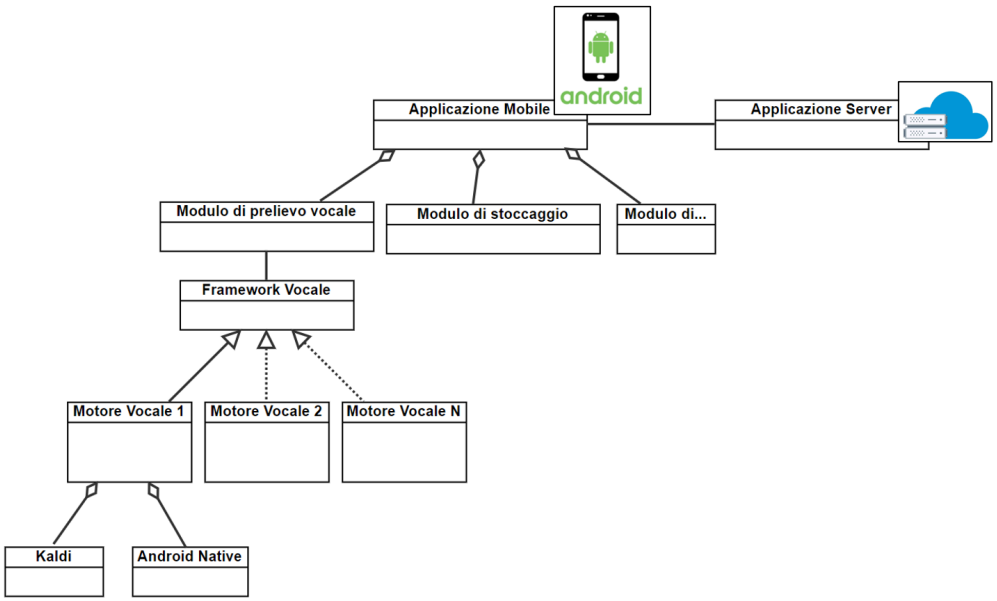

Figura 5.4: Architettura software del sistema vocale

5.2.1 Framework Vocale

Il framework vocale è stato pensato per essere *un'interfaccia* tra le applicazioni e il motore vocale scelto durante la fase di scouting, in modo da *disaccoppiare* la logica e guadagnarne in *modularità*. Cosi facendo, sarà possibile, in eventuali sviluppi futuri, sia una *manutenzione* più semplice del sistema, in quanto i pezzi di codice a cui è affidata la funzionalità vocale sono ben distinti da quelli puramente

applicativi, sia un aumento di prestazioni immediato grazie a un possibile cambio di motore, in tempi minimi. Ma quali *pattern* comuni possono essere individuati nei motori vocali ed essere generalizzati? Di quali *funzioni base* ha bisogno esattamente l'applicazione per funzionare? Prima di capire come disegnare l'interfaccia, è stato necessario rispondere a queste domande.

Pattern comuni nei motori vocali

In genere un motore vocale si presenta sotto forma di *interfaccia*[4], da estendere nella nostra personale implementazione. Possiamo pensare che estendere un'interfaccia sia come a una sorta di "promessa" che una classe si impegna a mantenere, che consiste nell' usare gli attributi e implementare i metodi astratti definiti nella prima: non è tanto importante come verranno implementati tali metodi all'interno della classe ma, piuttosto, che la denominazione ed i parametri richiesti siano assolutamente rispettati. In aggiunta, per loro natura i motori vocali sono anche dei *listener*: classi che rimangono in ascolto di particolari eventi, i cui metodi vengono "triggerati" e quindi attivati quando il relativo evento per il quale sono in ascolto si verifica.

In pratica estendere l'interfaccia di un motore vocale, che sia solo speech-to-text o solo text-to-speech o entrambi, significa definire *ciò che deve accadere* quando si verifica un evento o qualcuno richiede un'azione in particolare. I metodi più importanti e generali che sono definiti nell'interfaccia di un motore vocale *completo* e di tipo *general purpose*, da implementare nella nostra classe per contenere il comportamento desiderato, sono:

Init - questo metodo sarà il primo a essere chiamato. E' il responsabile di un opportuno setup, grazie a eventuali parametri passati alla chiamata. Restituisce un'istanza o inizializza il valore dell' attributo della classe che costituisce l'oggetto *motore vocale*;

OnStart - questo metodo sarà invocato dalla logica sottostante l'interfaccia (a noi trasparente), quando il motore vocale sarà *pronto* per essere usato;

Speak - questo metodo riceve come parametro una stringa e eventualmente altre opzioni che condizionano la sintesi vocale. E' responsabile dell'*output vocale*: la stringa passata verrà convertita in parlato;

StopSpeaking - *ferma* istantaneamente l'output vocale;

[4]un'interfaccia, nel paradigma di programmazione orientata agli oggetti, è un gruppo completamente astratto di membri.

onResult - questo metodo viene attivato automaticamente quando il motore vocale *ascolta* qualcosa di comprensibile, e ciò che ha capito passato come parametro sottoforma di stringa;

StopListening - *ferma* istantaneamente il processo di riconoscimento vocale. Siamo sicuri che a partire da questo momento onResult non verrà più invocata;

onError - questo metodo è attivato quando si scatena *un errore* di qualsiasi genere nel motore vocale. Di solito il codice errore è passato sottoforma di intero;

Come si può notare, le funzioni che iniziano con "on" sono metodi *reattivi*, cioè attivati e eseguiti dal sistema operativo quando si scatena un certo tipo di evento a cui sono associati.

Funzionalità richieste dall'applicazione

Ma cosa esattamente servirà alla nostra applicazione di prelievo per funzionare? Sicuramente sarà necessario dire qualcosa all' operatore, come ad esempio dargli indicazioni su cosa fare e come farlo (R2.1): in questo caso basterà semplicemente chiamare il metodo *speak()* e passare la frase che desideriamo venga pronunciata. Ma quando invece l'operatore dovrà rispondere e darci un *feedback*? Poichè i motori vocali considerati sono programmati per restuire *tutto ciò* che riescono a ascoltare, e non solo un particolare input di cifre, parole, o frasi, c'è la necessità di creare da sè un meccanismo che informi l'applicazione quando un particolare evento è accaduto, e farlo il più velocemente possibile per permettere all'applicazione di proseguire con il prossimo step e non perdere tempo prezioso che diminuisca l'efficienza dell'attività di picking (R2.3).

Inoltre, quest'applicazione, come ogni altra, sarà costituita da diverse sezioni e, come naturale che sia, ci sarà anche una *navigazione* in app, da supportare opportunamente. E' dunque preferibile che quando ci si sposta da una schermata all'altra, eventuali comandi ancora in esecuzione come il riconoscimento di uno specifico input, o lo speak() di una determinata frase siano stoppati qualora non più necessari. Sarà quindi necessario un meccanismo che permetta, il prima possibile, la notifica della transizione al framework vocale, per *fermare i servizi* e non avere *sprechi di risorse* inutili (R3.3).

Disegno dell'interfaccia

Come già detto, il framework vocale vuole porsi come interfaccia tra modulo applicativo e motore vocale, attraverso la definizione di un *pattern d'uso indipendente* dalla specifica implementazione di quest'ultimo. In poche parole, anche cambiando motore a monte, l'interfaccia potrà rimanere invariata funzionalmente (invariata anche l'applicazione a valle), in quanto si basa su quei *pattern comuni* definiti poco

più su. Conviene inoltre che il comportamento "comune" di questo framework sia codificato in una porzione di codice *generalizzato*, sia per escludere inutili ripetizioni di codice, sia per essere sicuri che i metodi esposti non cambino nome: in tal modo l'applicazione non subirà modifiche, e allo stesso tempo il cambio motore sarà agevolato da importanti pezzi funzionali già *testati* e *robusti*, che potranno essere riutilizzati senza riscriverli da capo.

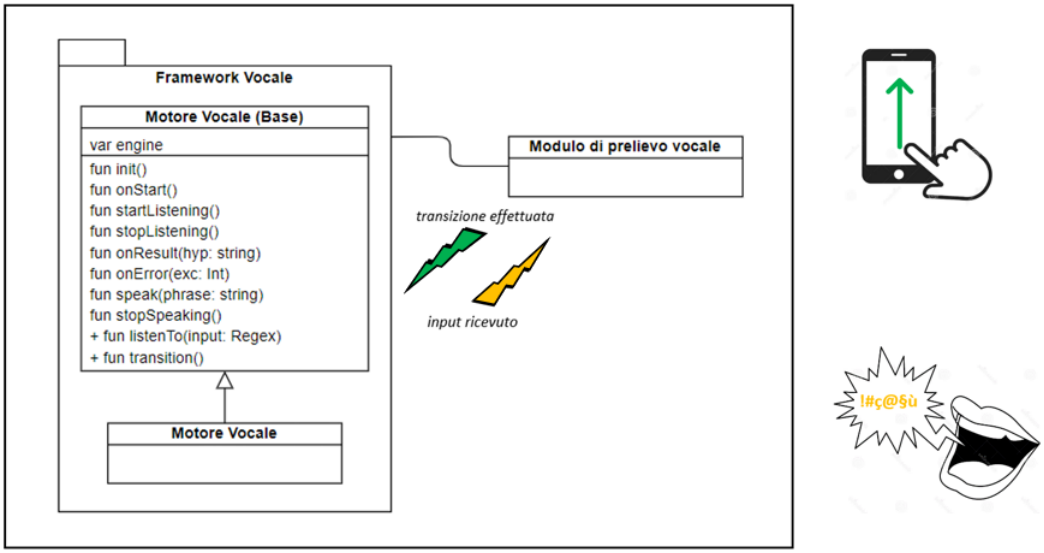

Figura 5.5: Schema base dell'interazione applicativo/framework vocale

Ma come adattare i metodi che potremmo trovare in ogni motore vocale di tipo general purpose, alle funzionalità richieste dall'applicazione di prelievo? Considerando le necessità dell'app, sarà necessario:

- **meccanismo 1** - il framework dovrà definire la logica da attuare in casi *ricorrenti e indipendenti* dal task applicativo supportato. Dovrà essere implementato un metodo che, a fronte di notifiche di avvenuta *transizione di navigazione* nell'applicazione, chiami i metodi opportuni nel motore vocale. In particolare dovrà essere capace di distinguere i diversi casi (cambio schermata, riapertura app, chiusura app) e reagire di conseguenza (invocando ad esempio stopSpeaking()/stopListening() nel caso di chiusura task), per un'esperienza d'uso più naturale possibile (R1.3);
- **meccanismo 2** - un metodo che faccia da *filtro* per tutto ciò che viene ascoltato e che si accorga che un determinato input, precedentemente *richiesto* dall'applicazione, è stato pronunciato: dovrà quindi reagire di conseguenza e *notificare* l'evento.

La schematizzazione del framework in figura 5.5.

5.2.2 Applicazione

L'applicazione di prelievo vocale si configurerà come un *modulo* all'interno di una applicazione più ampia, pensata per supportare i magazzinieri in vari tipi di attività, come inventario, stoccaggio, imballaggio. Questa integrazione però non si limiterà a un banale collegamento, bensì prevederà una *modellizzazione* comune di entità base astratte e da generalizzare: tutti i moduli hanno a che fare con l'entità magazzino, posizione, articolo, missione, etc. seppur con sfaccettature diverse. Questa astrazione ha permesso l'implementazione di pattern comuni nella comunicazione frontend/backend, implementati in sezioni di codice condivise, da non riscrivere: non sarà pertanto necessaria una progettazione a tutto tondo della comunicazione con il database, dal momento che molte funzionalità sono già disponibili di default. Ma *come propagare* le informazioni provenienti dal server all'interfaccia?

Il paradigma Model-View-ViewModel

Il paradigma scelto per questo progetto è il Model-View-ViewModel (MVVM), preferito ad altre opzioni come il Model-View-Presenter (MVP) o il classico Model-View-Controller (MVC), dal momento che:

- *separa* la logica di business da quella di presentazione, disaccoppiando così le componenti dell'applicazione e altrettanto i vari livelli di astrazione di un'applicazione;
- è più facile da *mantenere*, in quanto le view non sono direttamente coinvolte nella computazione di ciò che avviene dietro le scene: il codice sarà maggiormente *testabile*;
- conferisce una *struttura ben definita* al progetto e conseguentemente: facilità di *navigazione*, *comprensione* del codice, dal momento che ogni sezione ha una specifica responsabilità, e *modularità*, capace di semplificare l'aggiunta o rimozione delle funzionalità esistenti.

MVVM non è una libreria, bensì un pattern, un approccio allo sviluppo di applicazioni che permette quindi di *riusare* gran parte del codice. Consiste nella separazione degli aspetti della nostra applicazione in tre componenti:

- il **Model** rappresenta il punto di accesso ai dati. Trattasi di una o più classi che leggono dati dal DB, oppure da un servizio Web, per memorizzarli;
- la **View** rappresenta la vista dell'applicazione, ossia l'interfaccia grafica che mostrerà all'utente lo stato corrente del sistema;

- il **ViewModel** è il punto di incontro tra la View e il Model: i dati ricevuti da quest'ultimo sono elaborati per essere presentati e passati alla View. ViewModel e View sono in genere collegati da DataBinding e/o LiveData.

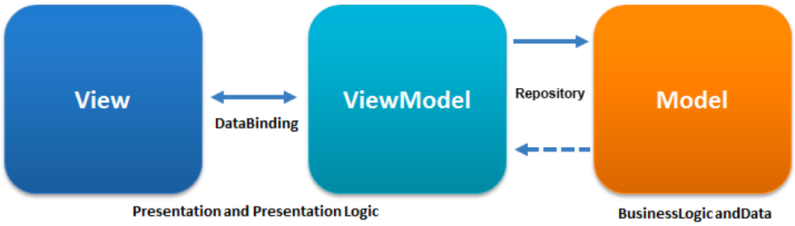

Figura 5.6: MVVM

Il principio del funzionamento di questo pattern è il ViewModel, il componente che ha il ruolo di *rappresentare* tutte le informazioni della corrispondente View, mentre quest'ultima si limiterà a *visualizzare* graficamente quanto esposto dalla prima, a riflettere i cambi di stato, attivare i comportamenti di risposta.

Più specificatamente, quando l'utente interagisce con la View, la variazione di stato viene comunicata al ViewModel tramite Binding o l'attivazione di un metodo, il quale in risposta "propaga" il cambiamento sul Model e aggiorna il proprio stato. Il nuovo stato del ViewModel si riflette infine sulla View.

E' da sottolineare il fatto che il ViewModel mantiene nel proprio stato non solo le informazioni recuparate dal Model, ma anche lo stato attuale della visualizzazione: ciò gli consente di essere del tutto disaccoppiato dalla View, e quindi operare indipendentemente dal ciclo di vita dell'activity: sarebbe infatti un problema conservare puntatori a memoria deallocata in seguito all'arresto momentaneo dell'applicazione. Il processo descritto in precedenza risulta essere un "two way", funziona cioè in entrambe le direzioni, ed è schematizzato in figura 5.6.

5.3 Prototipazione

La buona riuscita di un sistema che sia *usabile* dagli utenti, dipende molto dalla qualità dell' interfaccia offerta all'utilizzatore, in termini di soddisfazione dei bisogni, semplicità, intuitività d'uso. Molti sono i fattori da considerare e, a tale scopo, la fase di prototipazione rappresenta il punto più importante di una progettazione *user-centered* secondo l'ISO 13407: *Human-centered design process* [1].

In particolare, la prototipazione permette non solo di *generare* e *organizzare* le idee su come un'interfaccia utente può essere disegnata, ma anche *valutare* anticipatamente la qualità di una soluzione, grazie alla raccolta di *feedback* preliminari,

ottenuti dalla *simulazione* di comportamenti e funzionalità. È in questa fase, infatti, che si stabilisce *cosa* il sistema deve fare e *come* lo deve fare. Gli strumenti a supporto della prototipazione sono diversi e da usare in base allo *stadio* di design (precoce, ..., avanzato, finale) e il *pubblico* a cui sottoporlo (utenti, clienti, managers, etc.).

In questo capitolo, saranno quindi esposti i passaggi che hanno portato alla scelta e alla successiva realizzazione dell'interfaccia utente per il sistema. In particolare, partendo dagli step del processo vocale al paragrafo 5.1, verranno proposti due prototipi alternativi di interfaccia, in seguito valutati da esperti sulla base di *euristiche*[5] pensate per identificare *problemi di design*. Sulla base dei risultati ottenuti, saranno infine descritti i cambiamenti apportati alla UI e la realizzazione di un nuovo prototipo, questa volta *più fedele* al sistema reale.

5.3.1 Paper Prototypes: due modalità di interazione

Un prototipo è una rappresentazione o implementazione concreta ma *parziale* di un progetto di sistema, facilmente modificabile ed estensibile, che probabilmente include l' interfaccia e funzionalità di input/output. I prototipi sono usati per valutare il ruolo di un prodotto nella vita reale di un utente, valutare l'interazione dell'utilizzatore, ragionare su aspetti tecnici, senza aspettare di costruire interamente la soluzione, e si differenziano per *completezza funzionale* (orizzontale se prevede molte features, verticale se invece poche), livello di *fedeltà* (bassa con i disegni, alta se computerizzata), *durabilità* (se buttato subito, o duraturo se estensibile con altri moduli).

Più specificatamente, in questa fase si è scelta una prototipazione iniziale basata su *paper prototypes*, ossia prototipi su carta ... o meglio tablet. I prototipi su carta consistono in mock-up[6] *disegnati a mano* su fogli di carta di diversa dimensione, caratterizzati da bassa fedeltà nel look e esperienza d'uso, ma al contrario alta nella simulazione del backend, grazie al *facilitatore*: una persona che simula le operazioni del computer, ad esempio muovendo e unendo fogli, scrivendo sullo schermo, descrivendo effetti difficili da mostrare su carta. In particolare, la peculiarità del disegno a mano permette da un lato di concetrarsi su *aspetti generali*, perdendo di vista i dettagli, e dall'altro *incoraggiare* i feedback, in quanto questi prototipi sembreranno più "temporanei" e aperti a cambiamenti, anche radicali. La scelta di sostituire la carta con una tavoletta grafica è stata dettata dal beneficio strategico di

[5]Una euristica consiste in un insieme di strategie, tecniche e procedimenti inventivi per ricercare un argomento, un concetto o una teoria adeguati a risolvere un problema dato.

[6]Un mockup è una rappresentazione statica del progetto da realizzare, con la funzione di rappresentare nel dettaglio i vari contenuti, dimostrare le funzionalità base in maniera statica e mostrare anche il lato grafico del progetto.

una maggiore pulizia e velocità di disegno delle maschere, consapevoli del fatto che così facendo sarebbe stato molto più difficile tenere d'occhio le dimensioni: aspetto dopotutto secondario nel nostro progetto, dal momento che il mondo Android è vastissimo, ed è pensato per supportare schermi con enormi differenze in termini di dimensioni, risoluzione, screen ratio, orientamento.

Di seguito sono descritti i due prototipi di interfaccia realizzati. Oltre a una diversa disposizione delle informazioni, la sostanziale differenza tra le due interfacce consiste nelle modalità di *gestione dei casi limite*, come lo step 4 o gli errori. Se nel secondo sarà favorita un'interazione prettamente vocale, in modo da permettere una certa *rapidità di azione*, nel primo invece sarà prevista anche un'interazione tramite tocco su schermo, più affidabile grazie alla cattura di una *completa attenzione* dell'operatore, dal momento che è previsto che si riprenda il dispositivo in mano quando qualcosa va storto. La prima interfaccia sarà indicata con il nome *"interfaccia sicura"* e la seconda come *"interfaccia rapida"*.

Interfaccia sicura

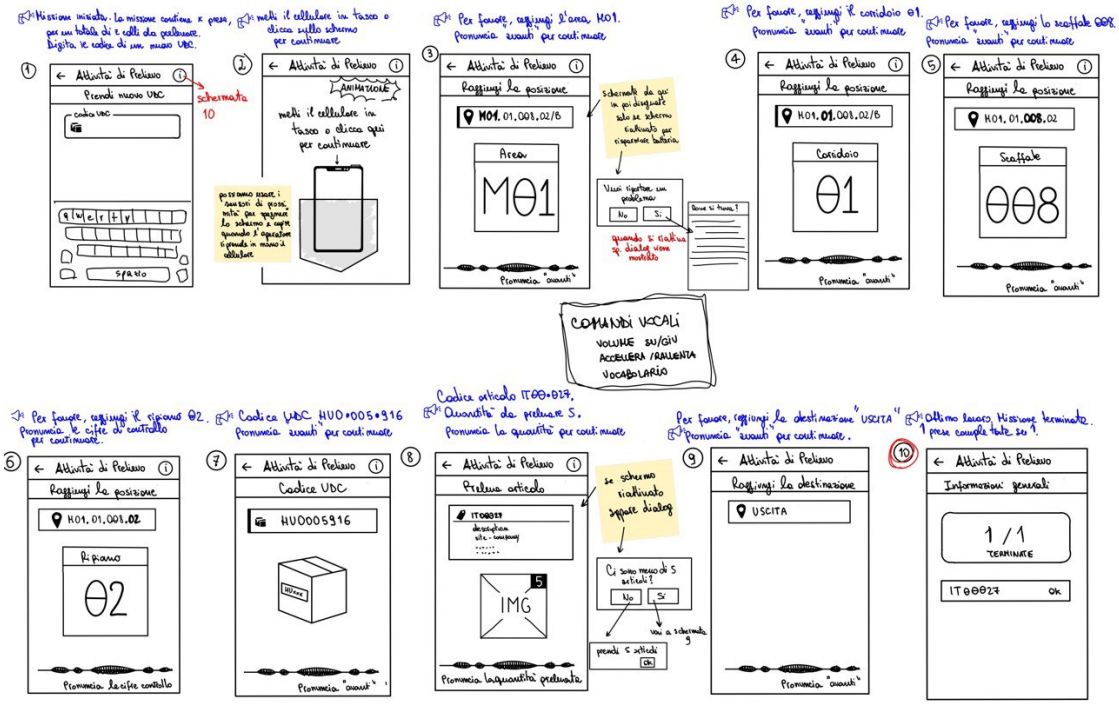

Figura 5.7: Paper Prototype 1

L' interfaccia in figura 5.7 è disegnata per supportare una interazione vocale durante le *normali azioni in movimento* (azioni successive alla 4), mentre, in tutte

quelle situazioni in cui è richiesta maggiore attenzione e precisione, un'interazione basata su *tocco su schermo*. Fanno parte di quest'ultima categoria: l'azione di inserimento codice (da tastiera) e tutti i casi di gestione errore, in cui è previsto un uso *proattivo* del sensore di prossimità, per accorgersi di quando il cellulare è posato, oppure ripreso. In particolare, dopo che il pickerista metterà il dispositivo in tasca, potrà riprenderlo in mano tutte le volte in cui è necessario gestire qualcosa di *anomalo* come quando gli viene detta una posizione di magazzino che non sa raggiungere o quando la quantità di prelievo aspettata è maggiore rispetto a quella in giacenza; in questi casi, quando lo schermo verrà riacceso, si presenterà una *finestra di dialogo* che aiuterà l'operatore a risolvere/segnalare il problema.

Più in generale, ogni schermata di questo prototipo corrisponde a un'azione del processo vocale, ed è accompagnata da una frase introduttiva (riportata in *blu* nel prototipo) che permette l'uso dell'applicazione anche *senza guardare* direttamente lo schermo, in cui invece sono raffigurate le informazioni *salienti* senza ulteriori dettagli superflui. E' importante sottolineare che è stato fatto un lavoro di *ottimizzazione* delle frasi tts: se il pickerista preleva un articolo nella posizione x.y.z.j e l'articolo successivo è in x.y.z.i, riceverà direttamente indicazione di spostarsi in i, senza riascoltare le coordinate comuni, poichè sono state già raggiunte. Inoltre, da notare che i codici sono pronunciati a gruppi di simboli, con pause nell'intermezzo (rappresentate da puntini), e che nel protitipo mancano i feedback audio tts in risposta ad azioni dell'operatore: se quest'ultimo sbaglia a pronunciare le cifre di controllo della posizione, il sistema lo renderà noto dicendo *"cifre di controllo errate"*.

Oltre a ciò, essendo questo prototipo pensato anche per situazioni "touch" *oltre* che vocali, su ogni maschera è stata inserita, in basso, una barra di progresso ad indicare lo stato del sistema *in ascolto* e poco più giù le parole che si aspetta per passare al prossimo step.

Per finire, esistono alcuni comandi vocali di supporto che si possono pronunciare in *qualsiasi* momento, utili al settaggio del motore vocale secondo le proprie preferenze:

- **volume sù/giù** - alza/abbassa il volume dell'output text-to-speech;
- **accellera/rallenta** - accellerare/rallentare la velocità dell'output text-to-speech;
- **vocabolario** - con questo comando è possibile ascoltare i comandi vocali sempre validi che si possono pronunciare (tutto questo elenco).

Interfaccia rapida

Come anticipato, nel secondo prototipo in figura 5.8 sarà prevista un'interazione prevalentemente vocale anche nei casi *limite*, per favorire un'esperienza completamente

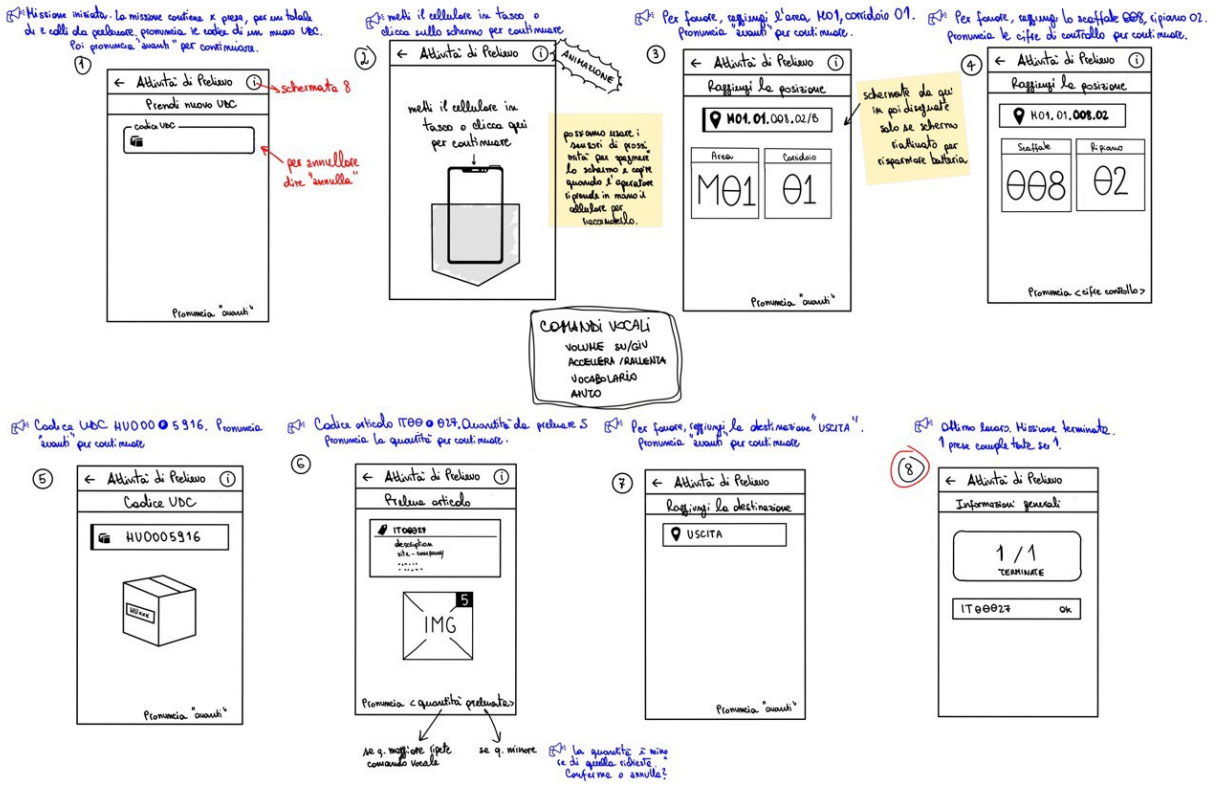

Figura 5.8: Paper Prototype 2

hands-free di prelievo.

Rispetto al prototipo precedente, sarà possibile pronunciare il codice dell'udc vocalmente, per intero o eventualmente a gruppi di cifre, ricevendo di volta in volta un feedback tts sulle cifre ascoltate dal motore vocale (per assicurarsi che sono state capite), e in caso di qualsiasi tipo di errore annullarle pronunciando "annulla".

Altra differenza importante, la scelta di *raggruppare le coordinate* della posizione da raggiungere in magazzino, in modo da permettere un *trade-off* tra memoria richiesta all'operatore nel ricordarle, e la velocità di esecuzione: l'operatore non dovrà fermarsi per dare un feedback al sistema in ogni coordinata raggiunta.

Inoltre, i codici non saranno più solo scanditi da pause nel mezzo di ogni gruppo, bensì il sistema attenderà di ricevere un feedback vocale dall'utente (rappresentato in figura dalla F), prima di far ascoltare il prossimo gruppo di simboli, in modo da prevenire errori di comprensione.

Molto diversa anche l'eventuale gestione errore al momento del prelievo, essendo del tutto *automatizzata* e vocale: quando il pickerista dichiarerà la quantità prelevata, se quest'ultima è maggiore, allora il sistema darà messaggio di errore, se invece minore, il sistema chiederà conferma sull'intenzionalità della sua azione,

pronunciando “conferma” o al contrario “annulla”.

Infine è stato aggiunto un nuovo comando vocale a quelli sempre validi, ossia “aiuto”: sarà possibile pronunciarlo per avere maggiori indicazioni quando non si sa bene cosa fare, non si sa come raggiungere una posizione.

5.3.2 Valutazione euristica dell’interfaccia

Cos’è la valutazione euristica

La valutazione euristica è un processo strutturato, in cui ci sono dei *valutatori* esperti che testano l’usabilità, funzionalità, accettabilità di un’interfaccia utente, eseguendo vari compiti step-by-step, al fine di individuare *problemi di design* precocemente e risolverli prima che costi troppo farlo, per un’esperienza utente più agevole. Più specificatamente, la valutazione euristica implica avere un piccolo insieme di valutatori (3-5), che esaminano l’interfaccia, adottando lo stesso punto di vista e obiettivi degli utenti, e giudicano la sua conformità con i *principi di usabilità* conosciuti come le “euristiche” di Jakob Nielsen (in appendice B.1). Secondo Jakob, infatti, la valutazione è più efficace quando più di un valutatore ispeziona l’interfaccia in modo indipendente: più specialisti UX sono coinvolti, più i problemi di usabilità possono essere scoperti.

Modalità di esecuzione

I test sono stati eseguiti presso il già citato laboratorio “area42”, in sessioni della durata di 30 minuti, con la partecipazione (individuale) di tre esperti diversi nel ruolo di valutatore e me in qualità di facilitatore e contemporaneamente osservatore.

Inizialmente il facilitatore ha fornito una breve descrizione introduttiva dell’applicazione di prelievo vocale e delle funzionalità implementate; successivamente è stata data una lista di task (in appendice B.2) al valutatore, poi lasciato libero di esplorare i prototipi *stampati su carta*, per capire se ogni singola azione fosse visibile e intuitiva. Se nel mentre qualcosa risultava difficile da capire, il facilitatore veniva in aiuto dando alcuni suggerimenti per completare l’attività. Al termine della valutazione, il facilitatore ha chiesto al valutatore di eseguire tutte le azioni mancanti che non ha visto o non ha provato. Le singole valutazioni si sone concluse con un feedback generale sul sistema dai valutatori.

Dopo il termine dei test, questi ultimi hanno dapprima dato una valutazione *individuale*, e poi una *aggregata*, dopo un reciproco confronto. La valutazione consiste nella schematizzazione di: problematiche registrate durante la prova, quali euristiche sono state violate e perchè, grado di severità (misurata su una scala con base 5).

Risultati: modifiche progettuali alla luce dei test

Alla luce dei risultati della valutazione (in appendice B.3), è stato scelto il prototipo numero 2 poichè il più apprezzato dai valutatori. L'esperienza di quest'ultimo prototipo è stata più intuitiva e fluida, mentre nel primo sono emerse più perplessità, soprattutto riguardo la possibilità di riprendere in mano il dispositivo: ciò significa liberare le mani da ingombri, e quindi maggiore possibilità di incidenti.

Il punto di forza del prototipo scelto è la comodità di un processo completamente hands-free (R1.1), di una gestione errori rapida (R2.3) e automatizzata (R1.3), ben *integrata* nelle operazioni di routine, mentre nell'altro è richiesta più attenzione nel distinguere i casi limite e contemporaneamente concentrazione nell'interazione con l'interfaccia, ricordandosi di prendere in mano il dispositivo: tutto ciò va a discapito dell'usabilità.

Sulla base dei problemi riportati dai valutatori, i cambiamenti da apportare all'interfaccia del secondo prototipo sono:

Cambiamento #1 - per risolvere il problema 4 (riportato in basso in grigio), e allo stesso tempo migliorare il controllo del sistema (H3), sarà *emesso un feedback* audio ogni qualvolta il sistema inizia il riconoscimento vocale;

> Nella prima schermata, in cui va inserito il codice dell'udc, non è chiaro se il sistema è da subito in ascolto oppure se si debba attendere.

Cambiamento #2 - come soluzione al problema 5, sarà richiesto un *ulteriore feedback* alla fine dell'ultimo gruppo di codici a barre, in modo da essere sicuri che l'operatore abbia individuato l'articolo, prima di continuare il prelievo;

> Durante la pronuncia dei codici a barre, l'ultima parte non è seguita da nessuna pausa e anzi, parte subito la parte successiva della descrizione tts. E' poco chiaro se l'utente abbia individuato l'articolo in così breve tempo.

Cambiamento #3 - su ispirazione del problema 6, verrà aggiunto un nuovo comando vocale da pronunciare durante la fase di prelievo: "descrizione", grazie al quale sarà possibile *ascoltare la descrizione* dell'articolo da prelevare, in modo da favorire un rapido riconoscimento visivo;

> Indicare solo il codice a barre di un articolo da prelevare non è agevole quando ci sono numerevoli oggetti, di piccole dimensioni, in una sola scatola.

Cambiamento #4 - conseguentemente al problema 7, verrà data la possibilità di dichiarare una *quantità prelevata nulla*, pronunciando "zero";

> Nelle operazioni di gestione errore durante il prelievo, non è previsto il caso in cui non ci sia del tutto giacenza in magazzino.

Cambiamento #5 - come soluzione al problema 8, verrà aggiunto un ulteriore comando vocale sempre attivo: "ripeti", grazie al quale sarà possibile *riascoltare* l'ultima indicazione data dal sistema;

> Se la descrizione tts termina e l'utente non ha testato attenzione (o ha dimenticato qualche dettaglio), non c'è modo di risentirla.

Cambiamento #6 - su suggerimento del problema 9, si è deciso di dare solo le *informazioni essenziali* all'operatore, rimuovendo la parte "pronuncia avanti per continuare" dalle indicazioni. Il comando vocale "avanti" sarà la parola chiave standard per avanzare di step nel processo, a meno che il sistema non dia indicazione di pronunciare altro (cifre di controllo, quantità).

> Ripetere ogni volta "pronuncia avanti per continuare" può risultare noioso, oltre che far perdere tempo ogni volta all'operatore.

Ispirandosi al prototipo 1 e all'esperienza derivante dal test effettuato, altri cambiamenti apportati sono:

Cambiamento #7 - nella prima schermata sarà previsto una doppia possibilità di input: sia vocale che da tastiera (come nel prototipo 1); essendo il pickerista non ancora in movimento verso una posizione x, potrà usare l'interazione che preferisce;

Cambiamento #8 - in basso allo schermo sarà aggiunta la *barra di progresso* (già presente nel prototipo 1), in modo da rappresentare graficamente lo stato del sistema: "attualmente in ascolto", coerentemente al feedback audio già descritto nel cambiamento #2;

Cambiamento #9 - saranno aggiunti altri comandi vocali sempre attivi, con effetto contrario: "pausa" e "pronto". Questi comandi sono stati pensati per interrompere e riavviare a proprio piacimento il riconoscimento vocale, in modo da dare maggiore libertà e controllo all'utente (H3). Questa modifica su ispirazione della prova di laboratorio, quando ci si è accorti che è molto utile sospendere la funzionalità di speech-to-text per fare altro, come parlare con un collega, o in questo caso il facilitatore.

5.3.3 Wireframe

Dopo la prototipazione su carta, in cui il focus principale erano le funzionalità, il naturale passo successivo è stata la realizzazione di prototipi più "fedeli", che

aiutassero a definire meglio la struttura delle interfacce utente. Sono quindi stati usati i *Wireframes*, rappresentazioni delle interfacce utente in bianco e nero, piuttosto imprecise, ondulate e ispirate dal disegno a mano, in modo da esprimere l'idea di un design ancora *preliminare*. In particolare, questo strumento fornisce, insieme a un'idea approssimativa sull'organizzazione delle informazioni, maggiori dettagli sugli output forniti all'utente e i comportamenti del sistema a fronte di input.

Per creare i wireframes si è scelto Balsamiq, uno strumento di progettazione design che offre un gran numero di elementi grafici di vario tipo, alcuni dei quali ispirati direttamente da Android, con uno stile "disegnato". Il punto forte di questo tool è la possibilità di "schizzare", in un solo foglio di lavoro, un numero indefinito di maschere (schermate) collegate da archi, attraversati a fronte di precisi input.

La creazione dei wireframes, in figura 5.9, è stata guidata dai disegni del secondo paper prototypes, arricchiti da tutte le modifiche introdotte dopo i risultati della valutazione euristica. In particolare, è stata ridisegnata l'interfaccia grafica della schermata #1 per implementare il cambiamento #7: essendo possibile un doppio tipo di input, sia vocale che tocco, si sono rappresentati due campi testo: il primo *editabile* (dove digitare il codice) e il secondo, al contrario, *non editabile*, in cui sarà visibile il codice inserito. Il risultato è che se si decide di digitare il barcode dell'udc e si preme invio sulla tastiera, questo sarà inserito automaticamente nel campo non editabile, mentre se si decide di pronunciarlo, si potranno leggere le cifre man mano che vengono dette. In entrambi i casi, continua a esserci lo stesso feedback visivo e sonoro: i due tipi di input sono funzionalmente equivalenti.

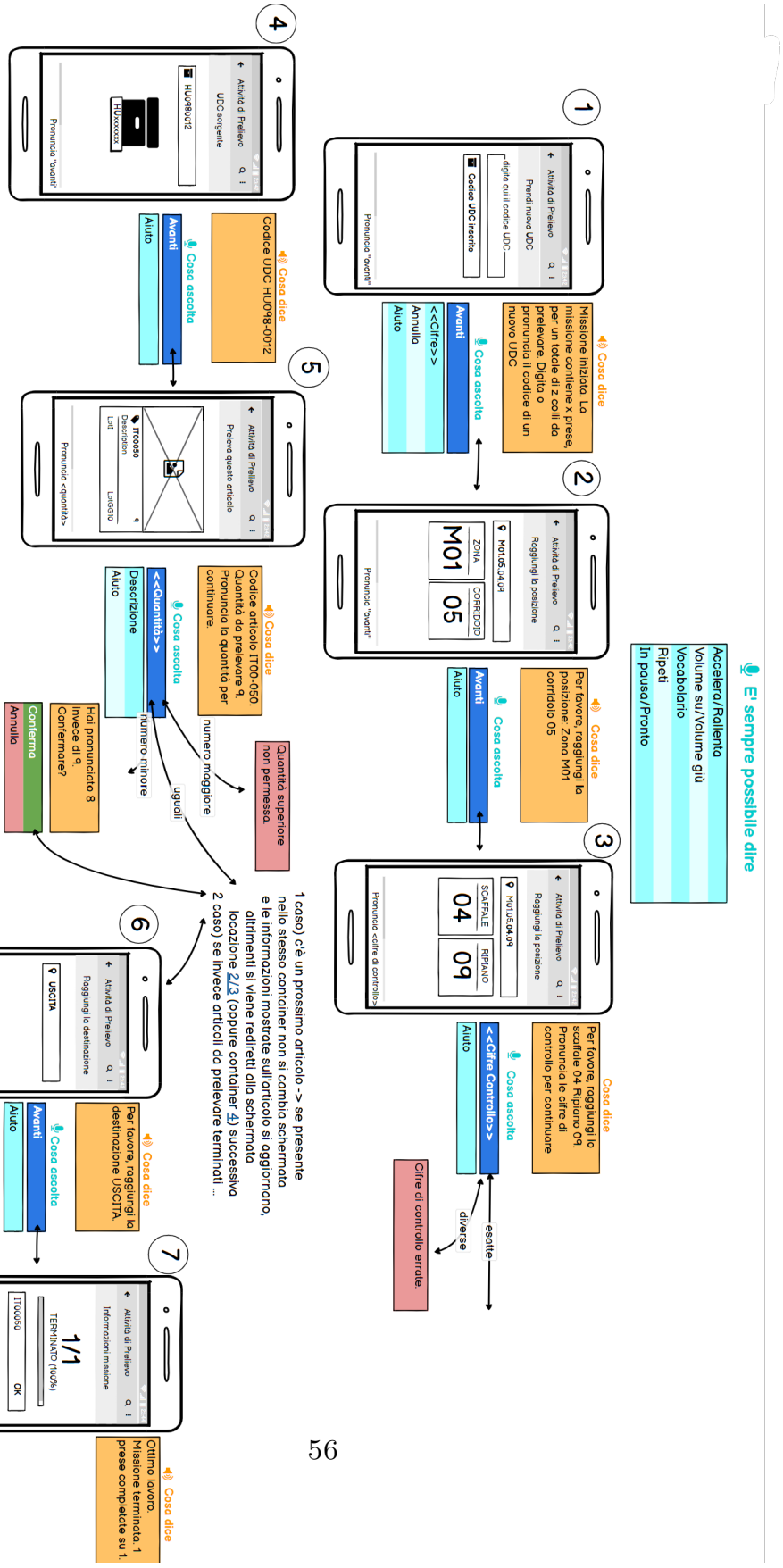

Figura 5.9: Wireframes realizzati con Balsamiq

Capitolo 6

Implementazione

Alla fase di progettazione, descritta nel capitolo precedente, fa seguito quella di implementazione, il cui scopo è quello di realizzare un'applicazione funzionante, che soddisfi i requisiti raccolti nel capitolo 3. Partendo dalla presentazione dell'architettura generale del sistema, in questo capitolo si descriveranno i componenti, le loro funzionalità e come questi comunicano e scambiano dati tra loro, evidenziando le scelte implementative fatte e le motivazioni da cui sono scaturite.

6.1 Architettura generale del sistema

In figura 6.1 è possibile osservare, ad *alto livello*, come è stata organizzata l'architettura generale del sistema e la comunicazione tra i diversi componenti. L'utente interagirà con alcuni componenti *di facciata*, come l'interfaccia grafica dell'applicazione o il componente che gestisce la conversione da parlato a testo (e viceversa), ma la maggior parte dell'applicativo, costuita dalla logica del processo vocale e la comunicazione con il server, gli sarà nascosta.

I componenti (o moduli) che formano il sistema di prelievo vocale implementato sono:

Backend - al backend sono legate le funzionalità di autenticazione dell'operatore, persistenza e restituzione dei dati, in seguito a richieste HTTP. E' costituito a monte da un *load balancer*, responsabile del bilanciamento del carico in entrata, grazie allo smistamento delle richieste sul network di *server* attivi. Questi ultimi sono costituiti da due moduli, uno apache per la gestione dell'autenticazione del client e l'altro di tipo applicativo, responsabile di far girare un'applicazione Java EE che implementa la logica di *domain business*. In aiuto dell' appication server troviamo un message broker che si occupa dell'incodamento di messaggi e richieste, permettendo così una comunicazione asincrona client-server. A monte troviamo infine il database MySQL di tipo

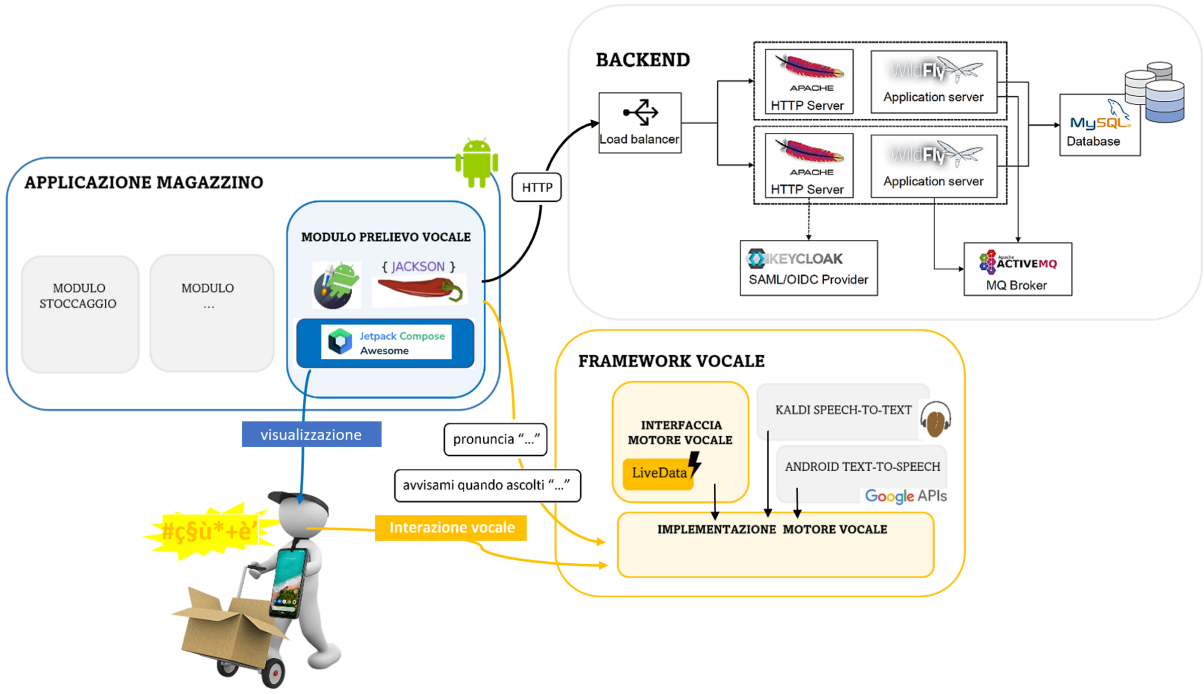

Figura 6.1: Architettura generale del sistema realizzato

relazionale, responsabile della persistenza dei dati. Questa architettura è disegnata per essere affidabile e scalabile, ed è del tutto *indipendente* dal lavoro di tesi.

Framework vocale - questo componente integra le funzionalità base offerte dalle librerie speech-to-text di Kaldi e text-to-speech di Android Native, come ampiamente descritto nei capitoli precedenti, per esporre metodi che possono essere usati direttamente dal client per sostenere un dialogo con l'utente. All'interno del framework è presente tutta la logica di notifica *cambio di stato* necessaria al supporto delle API vocali, e in più la generalizzazione di comportamenti da attuare a fronte di eventi standard nell'applicativo client, indipendenti cioè dallo *stato attuale* del motore vocale.

Applicazione di prelievo vocale - si tratta dell'applicativo Android vero e proprio, parte di una suite più ampia, che lo integra. Sfruttando il paradigma MVVM, questo componente è in grado di dialogare sia con il backend, per la collezione dei dati necessari al processo, sia con il motore vocale, a cui chiederà una sorta di "notifica" nel momento esatto in cui qualcosa di *specifico* accade. Quì sarà contenuta la logica di *presentazione*, come i comandi da dare

al pickerista per completare una missione, o la logica di prevenzione/gestione errori.

Come è evidente, questa applicazione è caratterizzata da una alta *modularità* a più livelli, non solo nei componenti (appena descritti) che implementano *logiche diverse*, ma anche all'interno degli stessi, grazie a *generalizzazioni* che permettono ulteriori divisioni di responsabilità. Tutto ciò permette maggiore *leggibilità* e *testabilità* del codice, ma anche una *manutenzione* più semplice del sistema, in quanto a ogni sezione di codice è affidata una funzionalità ben distinta e riconoscibile. Prima di entrare più nel dettaglio, vale la pena soffermarsi sulle caratteristiche di Android, la piattaforma target di questo progetto, per capire meglio i principi base di questo sistema.

6.2 Android

Android non è solo un *sistema operativo*, ma anche una *piattaforma software* pensata per supportare lo sviluppo di nuove applicazioni: è installato su più di un miliardo di dispositivi nel mondo, quali telefoni, tablet, orologi, TV, auto e altri. Android è realizzato da Google e licenziato come open-source Apache, il che permette libertà di apportare modifiche a tutti gli strumenti dell' ecosistema: le applicazioni non sono soggette ad alcun tipo di limitazione e di fatto, *non esiste distinzione* tra app native e quelle create da terze parti, poichè entrambe usano le API esposte dal Software-Development-Kit Android per l'accesso all'hardware sottostante.

Ogni applicazione Android è modellata da un singolo processo, istanziato dentro una macchina virtuale ART (Dalvik), concettualmente molto simile alla Java-Virtual-Machine (JVM), ma ottimizzata per girare su dispositivi mobile. Ruolo chiave in questa sinergia è svolto dal kernel Linux, che costituisce un *livello astrazione* tra lo stack Android e i componenti fisici dell'architettura.

I principali linguaggi di sviluppo Android sono C++, Java e Kotlin, il linguaggio scelto per questo progetto.

6.2.1 Kotlin

Kotlin è un linguaggio di programmazione open-source sviluppato dalla JetBrains, tipizzato staticamente[1], e multi-paradigma, in gran parte ispirato a tecniche di

[1]Un linguaggio tipizzato staticamente è un linguaggio (come Java, C o C++) in cui i tipi di variabili sono noti in fase di compilazione. Nella maggior parte di questi linguaggi i tipi devono essere espressamente indicati dal programmatore; in altri casi (come OCaml), l'inferenza del tipo consente al programmatore di non indicare i loro tipi di variabili.

programmazione funzionale: funzioni di ordine superiore, chiusure, immutabilità, funtori, monadi, ecc.

Kotlin è stato pensato per compilare primariamente sulla JVM, su cui vanta un 99.9% di compatibilità, grazie a una struttura del bytecode molto simile a java. Per tale ragione, sviluppando in Kotlin si possono tranquillamente riciclare classi/librerie scritte in Java (o anche compilati .jar), e persino fare l'opposto: ovvero sfruttare codice scritto in Kotlin da applicativi Java. Si dice dunque che Kotlin e Java sono linguaggi *interoperabili*: non c'è discriminazione tra una classe o una libreria scritta in Kotlin, da una scritta in Java.

Inoltre, Java è il linguaggio su cui è basato l'SDK Android, e per questo anche il linguaggio utilizzato da sempre per scrivere applicazioni *native* Android: per *transitività* è possibile fare la stessa cosa con Kotlin; nel Google I/O 2017, Google ha infatti dichiarato Kotlin come linguaggio ufficiale Android, tanto quanto Java. Dopo questo evento, improvvisamente, Kotlin ha visto una rapida espansione, e lo sviluppo di nuove versioni che lo hanno reso il linguaggio *ideale* per lo sviluppo di app: si è ritrovato ad essere il linguaggio preferito non solo da Google, ma anche da importanti realtà aziendali famose in tutto il mondo.

Ma perchè preferire Kotlin a Java per lo sviluppo? La risposta si può trovare nella *sinteticità*, dal momento che Kotlin è un linguaggio di programmazione più conciso e rapido rispetto a Java, che è più prolisso. Detto in modo semplice: con Kotlin è possibile scrivere meno codice che con Java. Facciamo un esempio pratico per capire le differenze di codice tra i due linguaggi:

```java
//Esempio scrittura "Hello World" in codice Java
class HelloWorldApp{
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}
//Esempio scrittura "Hello World" in codice Kotlin
fun main(){
    println("Hello World")
}
```

Scrivere meno codice significa anche: minore probabilità di bug, accelerazione dei tempi di sviluppo, riduzione dei costi. Ecco perché molti sviluppatori preferiscono Kotlin a Java.

6.2.2 Componenti di un'applicazione

Un'applicazione Android si può scomporre in due parti: la prima è costituita da codice che implementa la logica dietro le quinte, responsabile della *gestione dello stato* del sistema e delle funzionalità *dinamiche* dell'applicazione; la seconda è *statica*, scritta in XML (o in alternativa toolkit UI come Jetpack Compose), si occupa invece della definizione dell'interfaccia grafica con cui l'utente potrà interagire.

Un'applicazione Android non ha un singolo *entry point* (una main function), bensì può essere vista come un software package (apk-Android Package) costituito da diversi componenti, attivati dal sistema operativo in seguito a specifici *eventi*: una chiamata, l'interazione dell'utente su un'icona, una gesture, etc, e ognuno di essi si occupa di gestire la reazione a un diverso tipo di interazione utente/sistema operativo diversa. Possiamo quindi distinguere i seguenti componenti:

Activity - questo componente corrisponde a una schermata di un'app, ha un'interfaccia grafica con cui l'utente può interagire per reperire informazioni, per esempio una lista di mail da leggere, ed è *responsabile di un task*, come può esserlo la scrittura di un messaggio. La creazione delle activity avviene attraverso la descrizione di view (unità minime della UI) e la definizione delle modalità con cui queste si passano le informazioni.

Intent - rappresenta un'azione astratta (descritta da azione, dati allegati e categoria) che dovrà essere portata a termine da *un'altro componente*, attivato nel momento più opportuno per l'esecuzione (ottimizzazione di risorse). Gli intent sono utilizzati principalmente nella comunicazione tra più activity: ognuna può dichiarare, attraverso gli intent filter, un insieme di intent (task) che è in grado di gestire e portare a termine. Gli intent filters offrono dunque un metodo chiaro e semplice per rendere accessibili le funzionalità di un'app a disposizione di tutte le altre.

Broadcast Intent Receiver - è la componente del sistema Android in grado di *attendere l'esecuzione* di determinati eventi (intent lanciati in broadcast appunto), e reagire in modo opportuno: attivando una applicazione, mostrando una notifica, vibrando o altro ancora.

Service - i componenti di tipo *service* vengono usati quando è necessario eseguire funzionalità che potrebbero *durare a lungo* (dowloand di file, riproduzione musicale), e non direttamente legate ad aspetti visuali: non è prevista un' interfaccia grafica con cui interagire. I service sono quindi in grado di garantire l'esecuzione di alcuni task in background, cioè in modo indipendente da ciò che l'utente sta facendo e visualizzando sul display in quel momento.

Content Provider - è il componente in grado di *gestire i dati memorizzati* dall'applicazione, rendendoli disponibili ai propri client attraverso un'interfaccia che espone un set di operazioni CRUD (Create, Retrieve, Update, Delete). Questo meccanismo è necessario in Android, in quanto essendo un sistema Linux-based, ogni app ha un suo *userid* e un suo spazio di memorizzazione privato e protetto (la sua directory */data/nome_package*): hanno bisogno dei content provider per comunicare tra loro e scambiarsi dati.

La creazione, l' esecuzione, e anche la distruzione, costituiscono il cosiddetto ciclo di vita di un componente, caratterizzato da fasi diverse a seconda della categoria di quest'ultimo. In figura 6.2 quello di una activity.

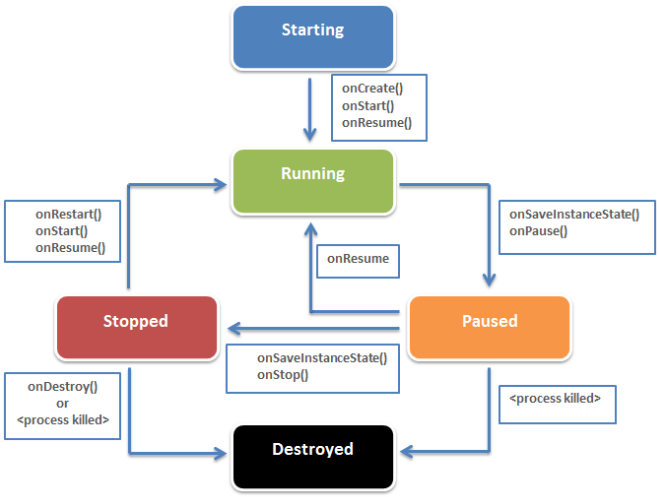

Figura 6.2: Lifecycle di una activity Android

Il ciclo di vita è gestito autonomamente dal sistema operativo, ma deve anche essere supportato dal programmatore che è *responsabile* della definizione del comportamento del componente in ogni sua fase. Il ciclo di vita è un concetto chiave nell'ecosistema Android, poichè i processi che qui hanno luogo sono *fluidi*: fintanto che un componente è usato, è garantita la continuità del processo, altrimenti se non lo è più (va cioè in background), *potrebbe essere arrestato* per far spazio ad altre applicazioni. Contestualmente a queste azioni, Android invia *notifiche* per avvisare dell'*evoluzione* nel ciclo di vita: sarà responsabilità del programmatore reagire a tali messaggi, rilasciando e riacquisendo risorse all'evenienza, al fine di evitare *perdite di memoria* o addirittura *arresti anomali* dell'applicazione.

Dopo questa rapida descrizione sulle caratteristiche della piattaforma Android, trattiamo ora l'implementazione della libreria vocale, in inglese "framework", realizzata per offrire funzionalità già "pronte all'uso", che saranno dunque integrate nell'applicazione di prelievo.

6.3 Framework Vocale

In informatica, una libreria è un insieme di *funzioni* e *strutture dati* predisposte per essere collegate ad un programma software attraverso un opportuno collegamento di tipo statico o dinamico. Lo scopo delle librerie software è fornire una collezione di entità di base pronte per l'uso e che permettono il *riuso di codice*, evitando in tal modo al programmatore di dover riscrivere ogni volta le stesse funzioni o strutture dati e facilitando così le operazioni di sviluppo e manutenzione. I vantaggi principali derivanti dall'uso di un simile approccio sono i seguenti:

- *Separazione* della logica applicativa da quella necessaria per la risoluzione di problemi specifici, come ad esempio calcolo di funzioni matematiche o la gestione di collezioni;
- Le entità definite in una libreria possono essere *riutilizzate* da più applicazioni;
- Si può modificare la libreria *separatamente* dal programma, senza limiti di spazio o capacità funzionali.

Inoltre, le librerie possono integrarne a loro volta altre librerie in modo *incrementale*: il framework realizzato, per la corretta implementazione dei suoi molti compiti, ne *integra* infatti molte altre, tra le quali spiccano, Kaldi per le funzionalità speech-to-text, Android Native per il text-to-speech e Android Jetpack per un supporto attivo e *real time* all'applicazione (gestione del lifecycle e delle notifiche).

Il framework vocale, in definitiva, vuole porsi nel sistema come *livello astrazione* tra l'applicativo di prelievo e il motore vocale, non solo per *disaccoppiare* relativi compiti e logiche, ma anche per il vantaggio strategico di un'infrastruttura *indipendente* dalla soluzione scelta per le funzionalità vocali (in questo caso Kaldi + Android Nativo), la cui *sostituzione* sarà il primo cambiamento da fare per raggiungere migliori prestazioni del sistema in sviluppi futuri. Per concludere, questo componente è realizzato pensando a un pattern d'uso "generale", che può essere riassunto in *"è in grado di supportare un dialogo uomo-macchina con risposte predefinite date dall'utente"*, dunque non legato a logiche attuative del prelievo manuale, ma sufficiente a implementarle: lo scopo è, ancora una volta, la possibilità di *riuso* futuro nell'implementazione di altri task di magazzino in versione vocale.

6.3.1 Soluzione: meccanismi

Prima di parlare in dettaglio dei vari componenti del framework e della loro implementazione, è necessario prima discutere le scelte fatte per realizzare i *meccanismi chiave* descritti nel paragrafo 5.2.1.

Meccanismo 1

Il framework dovrà definire la logica da attuare in casi *ricorrenti e indipendenti* dal task applicativo supportato. Dovrà essere implementato un metodo che, a fronte di notifiche di avvenuta *transizione di navigazione* nell'applicazione, chiami i metodi opportuni nel motore vocale. In particolare dovrà essere capace di distinguere i diversi casi (cambio schermata, riapertura app, chiusura app) e reagire di conseguenza, per un'esperienza d'uso più naturale possibile.

Nel nostro caso in particolare, l'obiettivo è lo sviluppo di un'*applicazione* standalone, che non abbia bisogno di applicazioni terze: sarà pertanto composta *solo* da componenti di tipo activity. Tenendo a mente questo, prima di parlare di come è stato progettato e pensato il *meccanismo 1*, sarà bene parlare di come funzionano le *transizioni* quando si passa da una activity all'altra, in seguito a interazioni dell'utente o altri eventi.

Per ogni applicazione avviata dalla schermata "home", il sistema operativo crea uno *stack di attività* inizializzato con la corrispondente activity *principale*. Durante il ciclo di vita di quest'ultima, l'activity può richiedere al sistema di avviarne una nuova: verrà creata e inserita in cima allo stack, e diventerà visibile e interattiva. Al tempo stesso però, l'activity precedente verrà spostata di una posizione indietro e rimarrà in background fino a quando la nuova rimarrà attiva.

Se la nuova activity termina in modo esplicito o se l'utente preme il pulsante "indietro", la precedente raggiungerà la cima dello stack e diventarà nuovamente visibile e interattiva. Ma cosa succede se invece la nuova activity ne crea una terza o una quarta? L'activity originale, andando troppo indietro nello stack, potrebbe essere arrestata per salvaguardare l'uso di risorse del dispositivo. Questo è ciò che succede in un'applicazione con più schermate e coerentemente più activity, riassunto in figura 6.3.

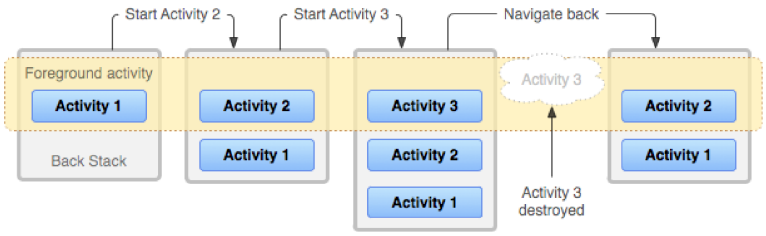

Figura 6.3: Stack di activity in Android

Ritornando a ragionare sul nostro sistema, come far arrivare queste notifiche al framework vocale, in modo che possa reagire nel modo più opportuno? La soluzione proposta da Google a questo problema è far si che il nostro framework diventi un *DefaultLifecycleObserver*, ossia una classe capace di osservare un ciclo di vita di un componente, a cui è stato legato esplicitamente dal programmatore, e dunque

ricevere anch'esso le notifiche inviate dal sistema, in modo da reagire alle *evoluzioni del ciclo di vita* osservato.

In particolare, cio significa che il nostro framework dovrà implementare l'interfaccia *DefaultLifecycleObserver*, e con esso i relativi metodi contenenti la *logica di reazione* ai vari tipi di eventi (chiusura app, cambio schermata, etc.). Per un'opportuna riuscita del meccanismo, durante la fase di inizializzazione del sistema, il motore vocale (in qualità di *osservatore* del ciclo di vita) riceverà la referenza al ciclo di vita dell'activity del nostro modulo applicativo che lo implementa e usa. Dopo aver implementato l'interfaccia, la nostra classe dovrebbe presentarsi così:

```
1  //bind al ciclo di vita
2  viewLifecycleOwner.lifecycle.addObserver(EngineLifecycleObserver)
3  class EngineLifecycleObserver: DefaultLifecycleObserver {
4      //callback eseguita quando va in pausa
5      override fun onPause(owner: LifecycleOwner) {
6          super.onResume(owner)
7          onPauseCallback()}
8      //Le funzioni di cui è possibile fare l'overridde in modo simile sono:
9      //onCreate, onStart, onResume, onPause, onStop, onDestroy.
10 }
```

Meccanismo 2

> Un metodo che faccia da *filtro* per tutto ciò che viene ascoltato e che si accorga che un determinato input, precedentemente *richiesto* dall'applicazione, è stato pronunciato: dovrà quindi reagire di conseguenza e *notificare* l'evento.

Per far funzionare un *meccanismo di filtro* su tutto ciò che viene pronunciato, sarà utile permettere al client (in questo caso l'applicazione) di indicare *quale* input è atteso, per ricevere coerentemente una "notifica" quando arriverà: ad esempio attraverso una chiamata a funzione *listenTo(something)*.

Ma esattamente come sarà fatto quest'input e come filtrarlo? L'input, dovendo essere pronunciato, sarà identificabile in una *stringa alfanumerica*, e quindi in linea di massima basterebbe un'uguaglianza tra stringhe per sapere se l'input ascoltato è anche quello richiesto. Tuttavia, in virtù di un' astrazione più alta possibile, e supponendo di dover supportare richieste del tipo "*ascolta tutti i numeri a due cifre*", sarà necessario usare una regular expression[2] a rappresentazione dell'input

[2]Un'espressione regolare (abbreviata in regex o regexp) è una sequenza di caratteri che specifica un modello di ricerca nel testo.

richiesto: nel caso ci sia match allora saremo sicuri che è stato pronunciato. A complicare il tutto, ci pensa però la *formattazione*: non esiste infatti uno standard nel riconoscimento vocale, e "cinque" potrebbe essere tradotto in parola o cifra. Alla luce di questo problema, sarà necessario, a monte del metodo "filtro", una normalizzazione della stringa, che le formatti univocamente: i numeri a parole saranno tradotti in cifre e eventuali spazi troncati.

A questo punto non resta che capire il sistema di notifica da implementare. In Android esistono diversi strumenti per una *comunicazione asincrona*: RxJava, Observable, StateFlow, etc., ma nel caso in questione si è ritenuta più opportuna la scelta della classe LiveData, per le sue proprietà e garanzie. LiveData è infatti una classe contenente dati osservabili, capace di notificare al suo osservatore (una classe *subscribed*) il cambiamento del valore dei dati sottostanti, in modo *consapevole* rispetto al ciclo di vita del componente che lo osserva (la classe osservatrice): saranno aggiornati solo gli osservatori che si trovano in uno stato del ciclo di vita *attivo*. I benefici apportati dall'uso di LiveData sono: nessuna perdita di memoria, nessun arresto anomalo dovuto all'interruzione delle attività, nessuna gestione manuale del ciclo di vita, dati sempre aggiornati.

Sarà possibile definire una *azione di risposta* negli osservatori, a fronte di cambiamento nel LiveData: se ad esempio in un LiveData è contenuto un intero di valore 5, e questo viene modificato da un terzo in 7, l'osservatore riceverà una notifica e, coerentemente al valore, potrebbe decidere (o meno) di attivare una callback di reazione al cambiamento. Il codice, nel motore vocale, dovrebbe avere questo aspetto:

```
class VoiceEngine{
    //chiamando questa funzione, si chiede al motore vocale di ascoltare la
    // regular expression recognize. Sarà ritornato un LiveData contenente
    // false, perchè l' input non è ancora stato ascoltato.
    fun listenTo(recognize: Regex): LiveData<Boolean> {
            currentRegex = recognize
            result = MutableLiveData<Boolean>(false)
            return result}
    fun onPhraseRecognized(phraseListened: String){
        var phraseNormalized = mapNumbers(phraseListened). //normalization
            replace("\n", "").lowercase()
        if (currentRegex != null && result != null)
            if (currentRegex.containsMatchIn(phraseNormalized)) //input riconosciuto!!!
                result.postValue(true) //threadSafe
    }
}
```

mentre nell'activity che usa il framework:

```
val regex: Regex<Boolean> = Regex("^input") //tipo di input atteso
val myInputWasPronounced = voiceEngine.listenTo(regex)
//chiamando observe() sul LiveData, è possibile diventare osservatore e
// registrare una callback di reazione al cambiamento
myInputWasPronounced.observe(viewLifecycleOwner,Observer<Boolean>{ pronounced ->
    if (pronounced == true)
        /*do something*/
})
```

6.3.2 Struttura e Componenti

Struttura gerarchica

In figura 6.4 è visibile la gerarchia di generalizzazione creata per *ottimizzare* al massimo la modularità del framework e *massimizzare* la quantità di codice riusabile.

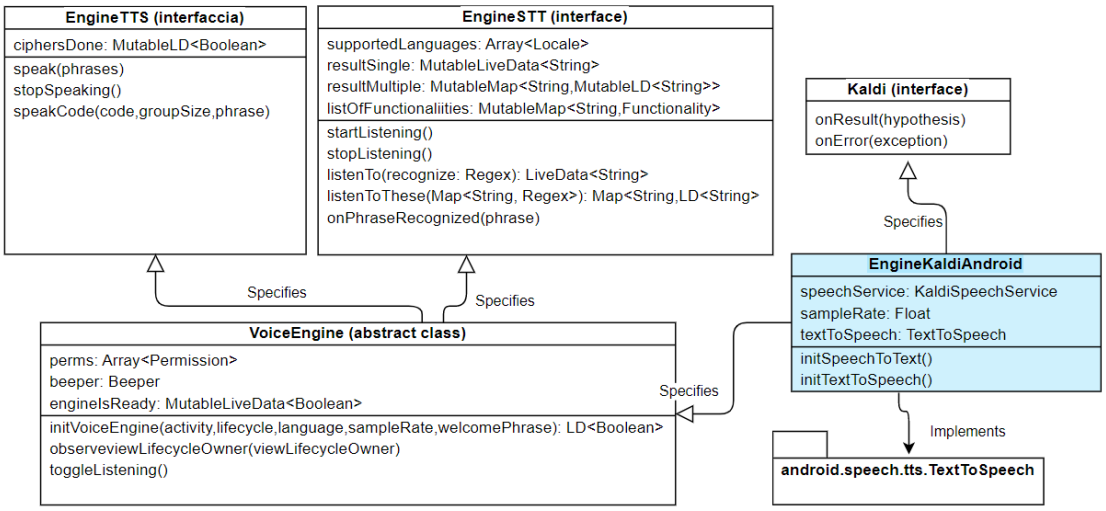

Figura 6.4: Visione dell'organizzazione delle gerarchie all'interno del framework vocale

Ma cos'è una *generalizzazione* e cosa implica? Si ha una generalizzazione (detta anche specializzazione) quando ci sono delle entità legate da una dipendenza gerarchica. In questi casi si ha un'entità genitore e una o più entità figlie (o sotto-entità): si dice che l'entità genitore *generalizza* le entità figlie e che le entità figlie *specializzano* l'entità genitore. Specializzando un'entità genitore, una entità figlia

eredita sia gli attributi, che le associazioni del padre, ma può anche avere attributi propri (non presenti nel genitore) e associazioni proprie (a cui non partecipa l'entità genitore). Da specificare che, il linguaggio di programmazione usato, Kotlin, pone un grosso limite alle generalizzazioni: una classe può specializzare infinite interfacce (eredità multipla), ma solo una classe. Questo vincolo ha *condizionato*, e non poco, la definizione della struttura della gerarchia: sono stato costretto a implementare in classi separate (nella cartella /utilities) alcune funzionalità che potevano essere, a rigor di logica, generalizzate. Nella gerarchia troviamo:

ENGINETTS - è l'interfaccia che si occupa di definire e *standardizzare* le funzionalità text-to-speech, poi implementate dalle classi figlie che ereditano da questa (VoiceEngine e EngineKaldiAndroid). Qui troviamo il metodo *speak()* che, dato l'input sottoforma di array di stringhe, è responsabile della sintesi vocale delle stesse; il metodo *stopSpeaking()* l'opposto, usato per fermare l'output vocale; *speakCode()* da usare per la pronuncia di codici per gruppi di simboli (di dimensione *groupSize*), con feedback dall'utente nel mezzo, eventualmente preceduti da una frase di start, *phrase*, letta per intero: la fine del codice è segnalata dal valore di *ciphersDone* a true.

ENGINESTT - in modo simile a EngineTTS, questa interfaccia definisce le funzionalità speech-to-text e, insieme ad essi, anche gli attributi necessari: in particolare *supportedLanguages* per rendere noto al client le lingue supportate per la conversione da parlato a testo. In particolare, il metodo *startListening()* permetterà l'avvio del riconoscimento vocale (di tutto ciò che è comprensibile al motore stt), *stopListening()* per stoppare il processo, per ascoltare un *preciso input* le due funzioni *listenTo*.

Quest'ultime (*listenTo()* e *listenToThese()*) implementano due pattern diversi, da usare in situazioni diverse:

- *listenTo()* prende in input una Regex, ed è utile quando si vuole ascoltare tutto (".*"), oppure sapere quando uno specifico input viene pronunciato, e nient'altro: regular expression uguale a "[0-9]{2}", e nel momento in cui succede *resultSingle* modificato con il valore *esatto di ciò che viene ascoltato* (ad esempio 12);
- *listenToThese()* riceve invece come argomento una mappa chiave-valore, in cui sono mappati una serie di possibili input da ascoltare: si è interessati a sapere quali tra questi viene pronunciato (*resultMultiple[i]* settato con quanto realmente ascoltato, in modo similare a *listenTo()*).

Infine *onPhraseRecognized()* è un metodo privato, non esposto, che risponde alla necessità di *riuso del codice*: l'idea dietro questo metodo è che, una volta distinta una qualsiasi parola, la logica di *filtro* e di *avviso* sia uguale per qualsiasi

implementazione di motore vocale. Questa funzione sarà implementata dalla classe astratta *VoiceEngine* e richiamata internamente dalle classi figlie, in questo caso solo *EngineKaldiAndroid*, non appena ascoltano *qualcosa*.

VoiceEngine - questa classe astratta è il contenitore di tutto il codice *riusabile*, che può essere generalizzato e condiviso dalle varie implementazioni di motore vocale: grazie a questo è possibile crearne uno nuovo in tempi minimi. Contiene i permessi (*perms*) che l'applicazione client dovrà chiedere all'utente per essere eseguita, la classe *Beeper* definita in /utilities per dare un feedback uditivo all'utente quando il motore vocale *ha capito* ciò che ha detto (e di conseguenza l'input dato corrisponde a quanto *atteso* dal sistema, poichè ha passato il filtro), e il LiveData *engineIsReady*, ritornato al client dopo l'esecuzione di *initVoiceEngine()*, settato a true a inizializzazione terminata.

In particolare, la funzione *initVoiceEngine()* contiene del codice generale nella sua implementazione nella classe VoiceEngine (come il bind dell'EngineLifecycleOwner al ciclo di vita dell'activity), ma sarà *EngineKaldiAndroid* ad estenderla con le opportune operazioni di inizializzazione specifiche, poichè dipendenti dall'implementazione finale.

Infine *toggleListening()* funziona da interruttore: se chiamata, inverte lo stato (da attivo a pausa e viceversa) del riconoscimento vocale, ed è uno shortcut messo a disposizione del client, che non dovrà gestire direttamente le logiche di cambio stato con start/stopListening().

Kaldi - interfaccia del *reale* motore vocale speech-to-text, implementato come classe *listener*. In particolare, i metodi dichiarati non saranno *mai chiamati* direttamente dal programmatore, bensì eseguiti dal sistema operativo in seguito allo scatenarsi di un determinato evento: *onResult()* abbinato al riconoscimento vocale (quanto ascoltato passato come stringa *hypothesis*), *onError()* se invece perviene una qualsiasi eccezione.

EngineKaldiAndroid - Effettiva implementazione del motore vocale, ultima specializzazione delle varie interfacce e della classe astratta. Grazie al codice in gran parte generalizzato e portato a fattor comune nella classe *VoiceEngine*, qui rimane ben poca logica: possiamo trovare l'effetiva inizializzazione e i collegamenti necessari tra i metodi (come quelli di Kaldi e il metodo *onPhraseRecognized()*).

Package

Diamo ora uno sguardo all'organizzazione nei packages del framework vocale, in figura 6.5 e 6.6. Innanzitutto si possono distinguere due moduli diversi: *engine* e *kaldi-android-engine*, entrambi strutturati sulla base del "nuovo modulo standard"

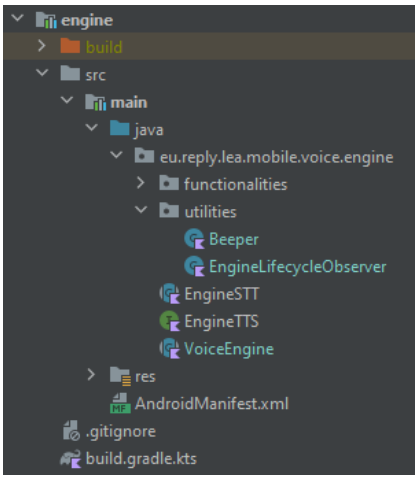

Figura 6.5: Modulo *engine*

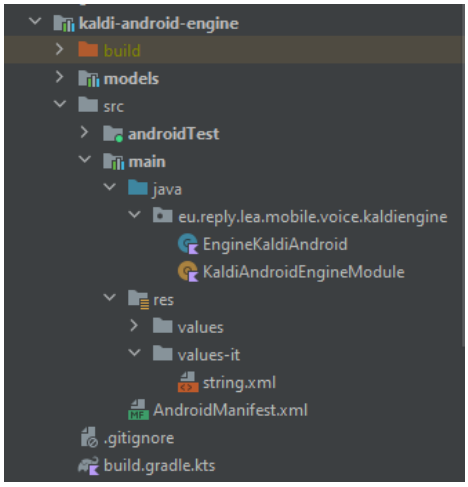

Figura 6.6: Modulo *kaldi-android-engine*

offerto da AndroidStudio. Il primo contiene tutto ciò che può essere definito *generale*, mentre il secondo lo implementa e specializza per creare un vero e proprio motore vocale fatto e finito.

Nel root dei due package, possiamo notare dei file .gradle, necessari a gestire la compilazione automatizzata dei vari moduli. E' stato scelto per questi scopi il tool *Gradle* rispetto a Maven, poichè il primo offre un modello più flessibile che supporta l'intero ciclo di vita dello sviluppo software (creazione, test, pubblicazione e distribuzione), e su qualsiasi tipo di piattaforma. La parte più importante nei file .gradle è quella in cui si specificano le dipendenze del modulo; ad esempio, il file *build.gralde* (nel modulo *kaldi-android-engine*), dal momento che la classe EngineKaldiAndroid usa l'interfaccia VoiceEngine, ma anche Kaldi e il modello di lingua italiana, apparirà con le seguenti dipendenze:

```
...
dependencies {
    api(project(":engine"))
    implementation(project(":kaldi-android-engine:models"))
    implementation (group = "com.alphacephei", name = "vosk-android")
    ...
}
```

Oltre alle risorse (/src/main/res), perlopiù stringhe in *string.xml*, e il manifesto Android *AndroidManifest.xml* (basico), troviamo il codice effettivo (/java). Sono qui

distinguibili le classi precedentemente descritte nella *gerarchia* di generalizzazione, ma anche:

engine/.../functionalities - qui è contenuta la modelizzazione di quei comandi *sempre attivi* come: ripeti, vocabolario, aiuto, volume sù/giù, accellera/rallenta, e la loro implementazione;

engine/.../utilities - qui è contenuto ciò che non si è potuto generalizzare nella gerarchia, a causa dei limiti dell'ereditarietà in Kotlin. In particolare la classe *Beeper* è responsabile della riproduzione del "beep" quando il motore vocale filtra con successo quanto ascoltato, mentre *EngineLifecycleObserver* è la classe che osserverà l'evoluzione del ciclo di vita dell'activity che implementa il motore vocale;

kaldi-android-engine/.../KaldiAndroidEngineModule - questo file permette l' *injection* del motore vocale, ossia la referenzazione di Kaldi-Android senza *mai* nominarlo esplicitamente. Sarà molto utile per il disaccoppiamento dell'applicativo di prelievo vocale dall'implementazione finale.

Queste parti saranno descritte funzionalmente più avanti nel corso del capitolo, quando si parlerà di funzionalità e integrazione.

6.3.3 Funzionalità: sguardo all'implementazione finale

Diamo ora uno sguardo all'implementazione concreta del motore vocale. Si procederà alla descrizione delle singole funzionalità, soffermandosi sui pezzi di codice *più critici*, che pertanto hanno richiesto un'attento sviluppo.

Inizializzazione

La procedura di inizializzazione può essere avviata chiamando la funzione initVoiceEngine(), che dovrà ricevere informazioni come la *lingua* (in questo caso solo l'italiano è supportato), la referenza alla *activity* Android, *sampleRate* del riconoscimento vocale, una *frase di benvenuto* da pronunciare a inizializzazione terminata. In particolare, quest'ultimo evento non è sequenziale alla chiamata del metodo, bensì può avere durata variabile (multithreading).

Si è scelto di segnalare al client la fine dell'inizializzazione attraverso una *notifica* di cambio valore del LiveData *engineIsReady* a true: si tratta della variabile tornata a fine esecuzione del metodo, al client, e nel contempo salvata nell'engine per modificarla al momento opportuno.

Ma qual è esattamente il momento in cui il motore è *pronto* ad essere usato? La risposta è quando lo sono sia il motore kaldi, che android tts: i listener saranno

attivati, e potremo usare una variabile *semaforo* (*engineIsReadyIfThisEqualTwo*) per aggiornare lo stato del sistema.

Da notare che il motore va inizializzato *una sola volta*, e, poichè potrebbe essere condiviso da *più fragment* che chiamano indipendentemente l'init, sarà necessario un check iniziale:

```
override fun initVoiceEngine(...): MutableLiveData<Boolean> {
        //check se il motore deve essere inizializzato
        if (currentLanguage == null || currentLanguage != newLanguage) {
            engineIsReady.value = false
            initSpeechToText(newLanguage)
            initTextToSpeech(newLanguage)
            currentLanguage = newLanguage
            initFunctionalities() //comandi sempre attivi
            ...}
        //in ogni caso va osservato il ciclo di vita del fragment. Se ne
        //occupa il codice generalizzato nella classe padre
        return super.initVoiceEngine()}
private fun initTextToSpeech(language: Locale) {
    TextToSpeech(activityContext) { speak(engineWelcomePhrase) }.also {
        tts?.setSpeechRate(actualSpeechRate)
        postReadyValue()}}
private fun initSpeechToText(language: Locale) {
    ... StorageService.unpack(activityContext, "model-it", "model",
        { model: Model? ->
            this.model = model
            speechService = Recognizer(model, mSampleRate) ...
            postReadyValue()})}
private fun postReadyValue() {
    engineIsReadyIfThisEqualTwo++
    if (engineIsReadyIfThisEqualTwo == 2)
        engineIsReady.postValue(true)}
```

Comandi sempre attivi

Dopo aver inizializzato il motore vocale, saranno attivati dei comandi vocali *sempre attivi*, utili sia al setup del motore vocale tts: *"accellera"*/*"rallenta"* e *"volume sù"*/*"volume giù"* , che all'usabilità con *"ripeti"*, *"vocabolario"* e *"in pausa"*/*"pronto"*. In virtù di una generalizzazione e modularità *più ampia* possibile,

si sono modellizzati i comandi sulla base di una classe padre interfaccia: "Functionality". Questa classe è modellizzata per rappresentare contemporaneamente due azioni (funzioni lambda): una con effetto *positivo* e una con effetto *negativo*, che rispondono rispettivamente a due parole chiave (Regex) diverse. In aggiunta è stato aggiunto *nativeSupported*, ossia un flag (Boolean), che sarà true nel caso il motore vocale scelto supporti nativamente la funzionalità x.

A titolo esemplificativo, è di seguito riportata l'implementazione della classe "volume sù/giù", su sfondo giallo per segnalare l' accezione di codice generalizzato e condiviso, per mezzo del package *engine*.

```
class Volume constructor(
    nativeSupported: Boolean = false,
    volumeUpKeywordToListen: Regex,
    volumeDownKeywordToListen: Regex,
    volumeUpCallback: (volume: Int) -> Unit,
    volumeDownCallback: (volume: Int) -> Unit,
    var actualVolume: Int = 0
) : Functionality(
    //parametri ricevuti dal costruttore, passati alla classe padre...
)
```

Le azioni (callback) sono definite all'esterno e passate al *costruttore* della classe *Volume*(così come gli altri parametri), che a sua volta specializza la classe *Functionality*: questo meccanismo permette un'ampia flessibilità.

Ma dove avviene, dunque, la definizione di tutte le funzionalità? Queste sono definite al momento dell'init() del motore vocale, quando c'è la chiamata a funzione *initFunctionalities()*: questo metodo è responsabile dell'inizializzazione della lista di oggetti "Functionality", passando a ogni costruttore le azioni desiderate e le keyword scatenanti. In poche parole, il "cerchio" del meccanismo si chiude quando, dopo che il motore vocale avrà ascoltato qualcosa e onPhraseRecognized() sarà chiamata, si cercherà di matchare (*hypothesis*) con le *keyword* (positiva e negativa) della funzionalità i-esima: se il confronto è *positivo* allora l'azione corrispondente potrà essere eseguita.

Speech-to-Text

Ma esattamente come funziona l'ascolto, ossia il la traduzione da parlato a testo? Il riconoscimento vocale inizia a fronte della chiamata a metodo *listenTo()* o *listenToThese()*: il primo è da usare nei casi *esclusivi*, il secondo nei casi *alternativi* (come già descritto in precedenza).

```
override fun listenTo(recognize: Regex): MutableLiveData<String> {
        startListening() //start stt
        currentSingleRegex = recognize
        resultSingle = MutableLiveData<String>("")
        return resultSingle!!
    }

    override fun listenToThese(recognize: Map<String, Regex>):
            Map<String, MutableLiveData<String>> {
        startListening() //start stt
        currentMultipleRegex = recognize
        resultMultiple = mutableMapOf()
        for (k in recognize.keys) resultMultiple!![k] = MutableLD<String>("")
        return resultMultiple!!
    }
```

Dopo questo setup, il metodo listener di Kaldi *onResult()* sarà triggerato ogni qualvolta il motore stt ascolta qualcosa di comprensibile (*hypothesis*): c'è subito dopo la chiamata al metodo generalizzato *onPhraseRecognized()* nella classe *VoiceEngine.kts*.

```
override fun onPhraseRecognized(phrase: String){
    //normalization...
    for (f in listOfFunctionalities.values) { /*prima prova di match*/ }
    //prova di match nella opzione esclusiva
    if (currentSingleRegex!!.containsMatchIn(phraseNormalized))
        resultSingle!!.postValue(phraseNormalized)
    //prova di match nelle opzioni multiple
    for ((k, v) in currentMultipleRegex!!){
        if (v.containsMatchIn(phraseNormalized)){
            resultMultiple!![k]?.postValue(phraseNormalized);
            break; //esco direttamente
        }
    }}
```

Il metodo *onPhraseRecognized()* è responsabile del matching di quanto ascoltato (*phrase*) con tutte le possibili alternative. Da notare due aspetti:

- si è scelto di tipizzare i mutableLiveData su *String*, ma avrebbe funzionato anche il tipo *Boolean* ai fini di una notifica al client. Nonostante entrambe le soluzioni abbiano un risultato simile, non sono *funzionalmente equivalenti*: nel

client, usando String, sarà possibile un *ulteriore confronto* tra quanto richiesto e quanto ascoltato (basti pensare alla regex [0-9]{2} e 12), non possibile con un Boolean, al prezzo di opererazioni computazionali più onerose;

- *listenToThese()* potrebbe essere tradotta nella chiamata di n volte *listenTo()* e la gestione di n LiveData da osservare. Ma perchè si è scelto di offrire questo doppio metodo? La risposta è nella *semplicità* di programmazione del caso alternativo, grazie alla restituzione di una mappa chiave-valore.

Text-to-Speech

Sono due le varianti a disposizione del client. La prima, più semplice, prevede la pronuncia dell'array di frasi ricevute (dopo la normalizzazione), inserendo tra di esse piccole pause. Nella classe KaldiAndroidEngine troviamo:

```
override fun speak(phrases: List<String>) {
    for (phrase in phrases) {
        //...normalizzazione
        tts?.speak(str, TextToSpeech.QUEUE_ADD, null, uttenceId) //pronuncia
        tts?.playSilentUtterance(300, TextToSpeech.QUEUE_ADD,null) //pausa
    }
}
```

La seconda (*speakCode()*) permette invece la funzionalità di pronuncia di codici *per gruppi*, con *attesa* del feedback dell'utente nel mezzo: grazie al supporto della prima, si è potuto generalizzare completamente il codice (in *VoiceEngine.kts*). In questa funzione viene dapprima creata la lista di chunks, ossia pezzetti singoli da pronunciare, e in seguito viene chiamata *resumeSpeaking()*, dove effettivamente avviena la pronuncia del pezzo i-esimo. Quest'ultima verrà anche chiamata ogni volta che il feedback (avanti) è ascoltato: il relativo check è presente nella funzione onPhraserecognized() di cui si è già abbondantemente parlato.

```
private fun resumeSpeaking() {
    //se non ci sono più feedback rimanenti, trigger dell'osservatore
    if (remainingFeedBacks == 0 && ciphersDone?.value == false) {
        ... ciphersDone?.postValue(true); return
    }
    speak(phrasesWaitingForFeedback!![i]) //altrimenti pronuncia i-esimo
    listenTo(Regex("^avanti")) //il sistema deve ascoltare il feedback
}
```

Gestione del Lifecycle

Per la gestione del ciclo di vita che il framework vocale dovrà osservare, è stata definita una classe che implementa l'interfaccia DefaultLifecycleObserver, in modo simile a quanto descritto al paragrafo 6.3.1. In particolare, per implementare una corretta gestione delle evoluzioni del ciclo, è fondamentale *intercettare* due momenti:

- la pausa - quando il fragment viene sostituito o l'app messa in pausa, si scatena la relativa notifica *pause*: è necessario stoppare un eventuale tts e ovviamente il riconoscimento vocale, oltre che resettare variabili di stato;
- l'avvio - quando il fragment torna/diventa interattivo o l'app è riavviata dal multitasking, arriva la notifica di *resume*: è necessario avviare il riconoscimento vocale qualora sia stato precedentemente fermato;

Il bounding del motore vocale al ciclo di vita del Fragment avviene durante l'inizializzazione, con la chiamata alla funzione *observeViewLifecycleOwner()*, dove viene chiamata la funzione addObserver() sulla proprietà lifecycle del viewLifecycleOwner passato come parametro dal client.

6.3.4 Integrazione: Gradle

Per una libreria software è essenziale che l'integrazione lato applicativo sia il più semplice possibile. Come già accennato, la compilazione di questo progetto, coerentemente alla maggior parte del mondo Android, è affidata a Gradle: uno strumento di automazione della compilazione (build automation tool) per progetti mono/multi repo, in grado di supportare le varie fasi di sviluppo di un software (dalla scrittura, fino al build, testing e pubblicazione su repository online).

Per usare questo framework in un'applicativo, il modo più semplice per l'integrazione è copiarlo e incollarlo all'interno del repository dell'app, includerlo nella pre-compilazione gradle dichiarandolo nel *settings.gradle* del package e infine aggiungere la dipendenza sul *build.gradle*. Dopodichè, sarà possibile dichiarare la variabile di tipo VoiceEngine nell'Activity o Fragment:

```
1 import eu.reply.lea.mobile.voice.engine.VoiceEngine
2 @AndroidEntryPoint class PickingMainFragment : Fragment(){
3   @Inject lateinit var voiceEngine: VoiceEngine
4   override fun onCreateView(...): View {
5      launcher.launch(voiceEngine.perms) //chiedere permessi all'utente
6      engineIsReady = voiceEngine.initVoiceEngine(  //initialization
7         activity = activity as MainActivity,
8         viewLifecycleOwner = viewLifecycleOwner, ...
```

```
9     ).observe(viewLifecycleOwner){ isReady ->
10        if (isReady == true)
11            voiceEngine.listenTo(Regex("something")) ... //reaction
12    }}}
```

Da questo script si possono notare due cose: la prima è la necessità di assicurarsi che l'utente abbia dato il suo *consenso* ai permessi richiesti dal framework vocale per funzionare (basta farlo una volta per sempre), la seconda è il nome della classe istanziata "VoiceEngine". A rigor di logica infatti ci si sarebbe aspettati di dover istanziare una classe chiamata "KaldiAndroidEngine"! Questo piccolo trucco è permesso dalle notazioni in **viola** e ci permetterà di dover modificare ancora meno codice per un *cambio di implementazione* del motore vocale integrato: la classa dichiarata VoiceEngine rimarrà uguale, cambierà però la reale dipendenza "iniettata" (KaldiAndroid in questo caso). Il meccanismo appena descritto è gestito dalla libreria di iniezione di dipendenze *Hilt*, capace di ridurre a minimo il *boilerplate code* del progetto: ogni volta che sarà presente una notazione **@Inject**, cercherà, nelle dipendenze dichiarate nel build.gradle, un file che espliciti come risolvere l'iniezione. Il file in questione è *KaldiEngineAndroidModule*, visibile nel package in figura 6.6. Per concludere, si è voluto facilitare l'uso del framework accompagnando ogni metodo esposto (public) da una *java documentation*, leggibile facendo hover del mouse su di esso:

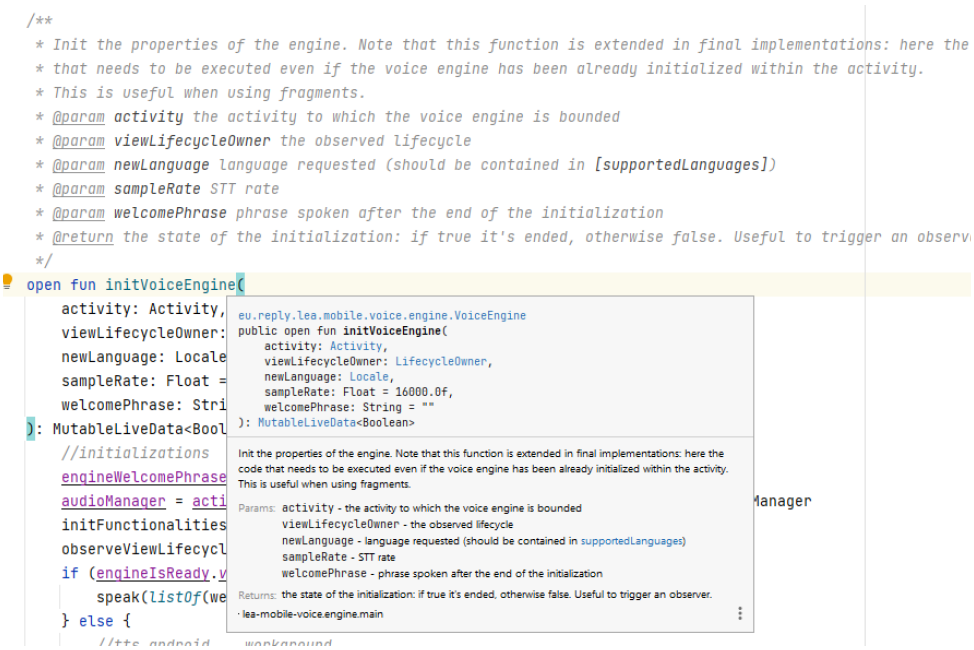

Figura 6.7: Esempio di uso della javadoc nel framework vocale

6.4 Applicazione

Dopo aver descritto l'implementazione del framework che abilita alle funzionalità di sintesi vocale, è arrivato il momento di capire come è stata creata l'applicazione del prelievo vocale. Come già detto in precedenza, questa applicazione sarà configurata come *modulo* di una suite più ampia, contenente altri moduli che implementano altrettanti processi di magazzino, come stoccaggio, imballaggio, spedizione, ecc... Il modulo applicativo realizzato, dunque, si integra non solo a un'applicazione più generale, bensì anche a un'intera infrastruttura pre-esistente, di cui fa parte il *Backend* precedentemente descritto. Come tale, la scelta di tutta la parte *database* e *server* è stata condizionata dallo stato dell'arte: esistono già gli *endpoint* a cui far arrivare le richieste HTTP.

Come descritto nella paragrafo 5.2.2 della progettazione, verrà implementato il paradigma MVVM per la separazione delle logiche di *business* da quella di *presentazione*: coerentemente a questa struttura si cercherà ora di spiegare le scelte fatte nello sviluppo dell'app.

6.4.1 Model

Il Modello è responsabile del recupero di dati dal server remoto, in modo da metterli a disposizione dell'applicazione. In questa sezione non potremo scendere troppo nei dettagli, poichè ciò significherebbe descrivere la metodica backend usata dai sistemi di logistica REPLY. Se ne darà comunque un'overview generale.

La comunicazione applicazione/server si basa su *modellizzazioni* di entità nel database; quelle con cui il prelievo ha a che fare sono: *task* di prelievo, l'*udc* di servizio (in cui mettere gli articoli prelevati), l'*articolo* da prelevare, la *posizione* di magazzino in cui recarsi. Prima di andare avanti, è bene dare uno sguardo a come sono fatti.

Entità

Il modello entità/relazione può essere riassunto nelle sue caratteristiche più salienti come in figura 6.8. Nel diagramma si distinguono:

Posizione - rappresenta una posizione di magazzino (*stabilimento*), e individuata univocamente dalle coordinate: *zona-corridoio-scaffale-ripiano*. Nel WMS la posizione è identificata da un *barcode*, e a ognuna sono assegnate delle *cifre di controllo*: cifre il più possibile randomiche per non essere uguali a quelle di altre posizioni adiacenti.

Missione di Prelievo - rappresenta la missione di prelievo manuale da effettuare (*status="TODO"*), identificata nel WMS da un *ID* unico, un *nome*

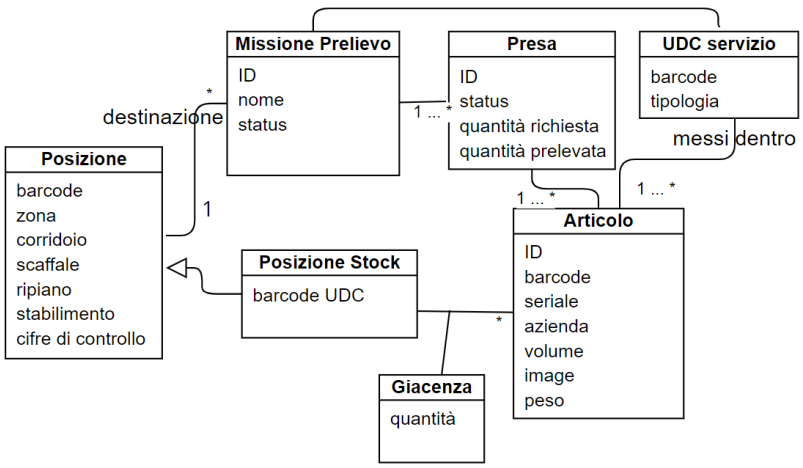

Figura 6.8: Diagramma Entità-Relazione UML

matricola. La missione è associata a un *udc di servizio* e a una *destinazione* da far raggiungere a quest'ultimo, e può avere da 1 a N prese da fare/in sospeso.

UDC DI SERVIZIO - rappresentà l'unità di carico (come può esserlo una semplice scatola) che dovrà contenere gli articoli prelevati in tutte le prese previste, identificata da un *barcode* nel WMS e caratterizzata da una *tipologia* (materiale, dimensioni, ecc...), da abbinare opportunamente al trasporto di merce con un certo volume e peso.

PRESA - rappresenta il singolo prelievo, di una certa quantità (*quantità richiesta*) di uno spercifico articolo, identificata da *ID*, e avente uno stato attuale (*status* a "TODO" o "DONE"). Le informazioni della presa verranno aggiornate su richiesta HTTP durante il prelievo, quando l'effettiva *quantità prelevata* sarà dichiarata.

ARTICOLO - rappresenta la merce stoccata in magazzino e da prelevare in questa missione, un singolo prodotto, identificato da un *ID*, *seriale*, *azienda*: queste informazioni sono riassunte nel *barcode* associato; oltre a queste, anche volume, peso, immagine. Un articolo può avere varie posizioni in magazzino.

POSIZIONE STOCK - rappresenta la posizione di stoccaggio di un articolo, che a sua volta può avere N stoccaggi. Specializza l'entità Posizione, a cui aggiunge la proprietà *barcode UDC* dell'unità di carico in cui è posto l'articolo.

GIACENZA - rappresenta la *quantità di giacenza*, in altre parole la massima quantità prelevabile di un articolo in una certa posizione di stock.

Componenti

Dopo la descrizione delle entità e delle loro proprietà, si può parlare di come il Modello fetcha i dati dal server remoto, e li mette a disposizione dell' applicazione di prelievo. Il modello può essere diviso in sotto-componenti:

- **Repository** - una classe interfaccia (*façade*) che espone metodi che permettono di reperire i dati da remoto e in modo simil "sincrono", ma che in realtà nascondono codice *asincrono* e molto spesso anche complicate logiche di elaborazioni dati e mix di chiamate API al server.
- **Remote Data Source** - rappresenta la sorgente primaria di informazione, contiene il codice necessario ad accedere al server remoto, contattare gli endpoint e inviare le richieste HTTP.
- **POJO** - i Plain Old Java Object (POJO) sono implementati da classi di tipo *data*, che in quanto tali usufruiscono di metodi standard per tutte le proprietà, come *set()* e *get()*. Questi oggetti ricalcano fedelmente le colonne delle tabelle della base dati: sono usati per modellizzare le entità ritornate (in seguito a GET) e inviate (in seguito a POST) durante la comunicazione con il server.

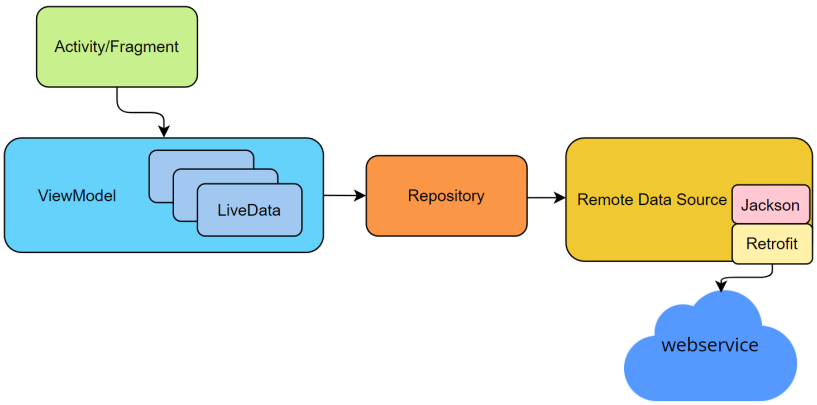

Figura 6.9: Repository pattern

API: passaggi necessari per contattare il server

Per osservare più da vicino l'iter necessario a recapitare i dati dal server, e renderli disponibili all'app, prendiamo ad esempio la prima chiamata del processo: il retrieve delle informazioni sulla missione di prelievo selezionata dall'operatore (identificata da un ID). In questo caso, il primo metodo da chiamare è la funzione esposta dal repository *retrieveJobs()*, la quale torna una *lista* di *JobDetail*: si tratta della

modellizzazione generalizzata di una missione di prelievo, contente una lista di prese rimanenti da fare.

```
class PickingRepository @Inject constructor(
    @ApplicationContext val context: Context,
    private val pickingClient: PickingClient){
    override suspend fun retrieveJobs(taskId: String): List<JobDetail> {
      apiCall {
         pickingClient.pickingDetails(id = taskId, ...)
      }.let { result ->
         if (result.isSuccessful()) return result.pickingJobList.mapToJobList()
         else {
           Log.e("Picking info fetching failed")
           return exception
        }
    }}}
```

La prima cosa che saltà all'occhio guardando il codice sorgente e la keyword *suspend*, che sta a indicare la capacità di questo metodo di *sospendere* la computazione e *riprenderla* al momento opportuno (quando i dati saranno disponibili dal backend): questo meccanismo è realizzato dalle *coroutines*, capaci dunque di separare l'idea di task da quella di thread, dal momento che nessun thread sarà messo in attesa e nessun thread è ritenuto responsabile della computazione. Le coroutine sono molto utili in questo contesto, in quanto offrono la possibilità di scrivere codice *sequenziale* mentre si affrontano nelle retrovie calcoli asincroni, ben più difficili da controllare: il tutto è gestito automaticamente dal compilatore.

Verrà quindi usato apiCall per la "costruzione" della coroutine: qui la chiamata a *pickingDetails()* sul PickingClient (Remote Data Source), il metodo responsabile dell'implementazione dell'API di fetch dei dati. Di seguito l'implementazione:

```
interface PickingClient{
    //...
    @GET("wms/PickingTaskJobs")
    @Headers("Accept: application/json")
    suspend fun pickingDetails(
        @Query("task.id") id: String,
        @Query("_select") select: String?,
    ): Response<List<PickingJobDetail>>
}
```

Grazie a Retrofit, si è in grado di accedere ai servizi REST semplicemente decorando con opportune annotazioni i metodi delle interfacce che rappresentano i servizi da utilizzare: *pickingDetails()* ne è un esempio. Passando a quest'ultimo l'id della missione (*task.id*) e i parametri a cui si è interessati (*select*), si riceverà una lista di oggetti di tipo *PickingJobDetail* (POJO, molto simile all'entità *presa* descritta precedentemente), racchiusa in un'altra classe involucro *Response*.

Quando finalmente l'API ritorna e dunque i dati saranno disponibili, verrà eseguito il codice nello scope del *let*, chiamato "continuazione": qui il check che tutto sia andato ok, e, se la richiesta al server è terminata con successo, ci sarà il ritorno all'invocatore delle informazioni ricevute sulla missione di prelievo con ID richiesto. Prima che accada però, c'è un ultimo passaggio di elaborazione effettuato da *mapToJobList()*, metodo responsabile della trasformazione del tipo *PickingJobDetail* a *JobDetail*, generalizzazione necessaria per compatibilità (in fasi successive) con il codice sottostante il modulo applicativo di prelievo.

Lista API

Di seguito, in tabella 6.1 la lista delle API usate per l'implementazione del prelievo vocale.

URL	Metodo	Descrizione
wms/PickingTask/{taskId}	GET	Ritorna un oggetto che modella una missione di prelievo manuale
wms/PickingTaskJobs/{taskId}	GET	Ritorna una lista di oggetti che modellano le prese (e relativi dettagli) associate a una specifica missione di prelievo manuale
wms/PickingServiceUdc/{udcBarcode}	GET	Ritorna la modellizzazione dell'entità che rappresenta un udc. Può essere usata per sapere se uno specifico barcode è già stato associato in precedenza a un altro udc.
wms/PhysicalStock/{itemId}	GET	Ritorna l'entità articolo avente l'id specificato.
wms/MapLocationInfo/{barcode}	GET	Ritorna l'entità posizione di magazzino e le sue caratteristiche. Contiene anche le cifre di controllo assegnate a quest'ultima.
wms/PickingTask/{taskId}/createHu	POST	Riceve nel corpo della richiesta HTTP un oggetto contentente le caratteristiche di un nuovo udc e associa quest'ultimo (come udc di servizio) al prelievo manuale avente taskId dato.

wms/PickingTaskjob/{jobId}/pickingConfirm	POST	Riceve un oggetto con i dettagli della presa con id specificato. La risposta HTTP del server confermerà la buona riuscita o meno dell'operazione di prelievo.
wms/PickingTask/{taskId}/confirmHu	POST	Chiamata al momento del raggiungimento della destinazione finale del prelievo, posta un oggetto contenente dettagli generali della missione. Da questo momento in poi, la missione è considerata dal server terminata.

Tabella 6.1: Lista API applicazione prelievo vocale

6.4.2 View

Come nei modelli Model–View–Controller (MVC) e Model–View– Presenter (MVP), la *Vista* (in inglese "View") è la struttura, il layout e l'aspetto di ciò che un utente vede sullo schermo: rappresenta il modello e riceve l'interazione dell'utente (gesti di tocco dello schermo). Successivamente, questi eventi saranno gestiti dal View-Model attraverso il *data binding* (proprietà, callback di eventi , ecc.) definito in modo opportuno per collegare View e View-Model.

In Android, le classi che costituiscono la View sono quelle che estendono *Activity* o *Fragment*, in linea di principio le uniche che dovrebbero aver a che fare con la *UI* o le *interazioni* con il sistema operativo: sono dunque definite come *classi colla* e possono essere distrutte in qualsiasi momento dall'OS (lifecycle).

Fragments

Per la costruzione dell'interfaccia grafica, si è scelto di usare i Fragment per la loro *flessibilità*: non essendo specificate le *caratteristiche* del dispositivo Android target, è indispensabile adottare la soluzione che meglio si adatta alla variabilità delle caratteristiche dello schermo (dimensione, orientamento, risoluzione, ecc...).

Il concetto di *frammento* è stato introdotto da Android per dare una soluzione proprio a questo problema e dunque consentire al programmatore di progettare un'interfaccia utente *univoca* per tutti gli schermi, invece di N pilotate da caratteristiche variabili. Le proprietà dei frammenti sono:

Modularità - un compito complesso, svolto da un'attività, può essere suddiviso in tanti compiti indipendenti più piccoli che possono funzionare da soli (ad esempio qualcosa che gestisce una mappa o un time tracker). Gli elementi

divisi possono essere implementati da singoli frammenti. Questo approccio si traduce in una migliore organizzazione e manutenibilità (divide et impera).

Riutilizzabilità - i frammenti sono indipendenti, riutilizzabili molte volte non solo nella stessa attività ma anche in altre attività, riducendo a zero la necessità di riscrivere le stesse cose.

Adattabilità - dividere un compito in compiti più piccoli significa flessibilità perché possono essere combinati in modi diversi, a seconda delle caratteristiche del dispositivo, ottenendo lo stesso risultato finale (palmare vs tablet). In questo caso l'attività fungerà da a *controllore gerarchico* delle funzionalità offerte da ciascun frammento (blocco).

I vantaggi dell'utilizzo dei frammenti in questo scenario sono: una GUI più adattiva, viste a schermate più facili da creare, supporto alla creazione di finestre di dialogo personalizzate e grafi di navigazione.

Inoltre, un frammento è un punto intermedio tra *Activity* e *View*, perché come una Activity ha un ciclo di vita complesso da gestire e prende parte all'interazione con l'utente, e dall'altre parte può avere una gerarchia di componenti grafici che possono far parte di quella dell'attività che la ospita, come appunto una View. In ogni caso, a differenza delle attività, i frammenti vengono manipolati direttamente dal programmatore.

Navigazione

Come descritto nella progettazione 5.3.3, l'applicazione sarà composta da diversi step, a cui corrisponderanno altrettante schermate: è pertanto necessario supportare la *navigazione*, in questo caso tra fragments.

La navigazione è abilitata dalla risorsa *nav_graph.xml*, un file contenente il *grafo* (visibile in figura 6.10) delle schermate dell'applicazione: ogni schermata è un *nodo*, mentre le transizioni possibili sono rappresentate dagli *archi* che collegano i nodi.

L'oggetto navController è incaricato di leggere questa struttura e fornire il metodo *navigate*, a cui è possibile passare l'id univoco dell'arco di navigazione desiderato (eventualmente arricchito da un bundle) per ottenere una transizione. Nell'applicazione di prelievo vocale, le transizioni saranno effettuate a seguito di specifici input e scatenate nelle *callback* di reazione. Il tutto avrà un'aspetto molto simile al seguente pezzo di codice:

```
1 voiceEngine.listenTo(numbersRgx).observe(viewLifecycleOwner) { strListened ->
2     if (strListened.containsMatchIn("...")) //se c'è un match
3         navController.navigate(R.id.action_schermata1_to_schermata2, bundle)
4 }
```

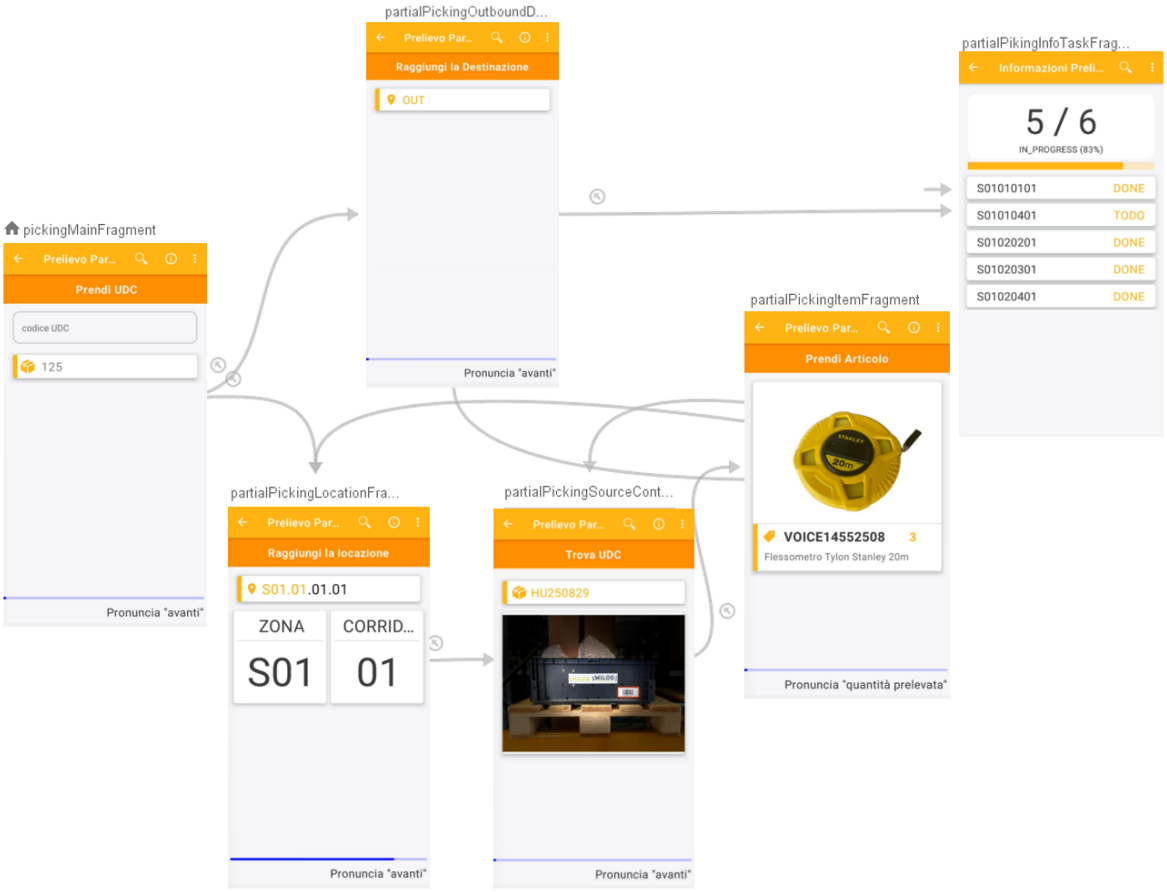

Figura 6.10: Grafo delle schermate dell'applicazione di prelievo vocale

Jetpack Compose

I singoli componenti grafici della vista, e più in generale l'intera gerarchia ("albero") di elementi GUI, sono stati definiti grazie alla moderna libreria Jetpack Compose, alternativa all'approccio più classico dei file layout .xml + inflater. Questa scelta è stata motivata dalla volontà di superare i limiti di quest'ultimo metodo, quali la frammentazione dell'albero di views in più file, una gestione dello stato attuale del sistema a stretta dipendenza dagli handlers, il passaggio di dati e informazioni tra le views lasciato alla responsabilità del programmatore.

Per ovviare questi problemi, Jetpack ha creato la *Compose library*, un moderno toolkit che mira a semplificare lo sviluppo dell'interfaccia utente, attraverso un approccio *dichiarativo* e *data-driven*: i blocchi costitutivi (uniti per creare l'albero delle views) sono emessi da funzioni *componibili* (etichettate con l'annotazione @Composable), responsabili di descriverne in modo dichiarativo l'aspetto e il comportamento. In questo modo, il programmatore *non* è più responsabile di

propagare i cambiamenti di stato nella UI, in quanto sarà il framework stesso responsabile dell'aggiornamento dell'intero albero di visualizzazione.

Le funzioni componibili ("composable") sembrano delle funzioni standard di Kotlin ma non lo sono: possono ricevere parametri di ogni tipo, ma *non restituiscono* un risultato, bensì *emettono* un blocco dell'interfaccia utente che dovrà essere visualizzato dal *compose runtime* invocando altri componibili più elementari forniti nativamente dalla libreria Compose. Inoltre, i composable possono essere *stateless* o *stateful*: stateless se guidate solo dai parametri che ricevono, stateful se incapsulano valori mutevoli che *influiscono* sia su loro stessi che sui composables figlio invocati; quando il valore cambia, il processo di *ricomposizione* verrà attivato.

Nell'applicazione di prelievo, a ogni Fragment corrisponde una grande funzione composable per la realizzazione della schermata, a sua volta composta da molte altre, responsabili di pezzi minori, secondo una strategia "divide et impera". Ad esempio, la schermata di "posizione magazzino" ha l'aspetto del seguente codice:

```
@Composable fun PickingLocationComposable(
    viewModel: PickingViewModel,
    topDescription: String,
    step: MutableState<Int>,
    progress: State<Float?>,
    keyword: MutableState<String>, ...){ ...
    TopDescription(topDescription),
    PickingLocationPartialHighlighted(...)
    Row {
        LocationPortionCard(
            label = if (step.value == 0) "ZONA" else "SCAFFALE",
            value = if (step.value == 0) ... else ..., ...)
        if (firstIndexDot != location.length) {
            LocationPortionCard(
                label = if (step.value == 0) "CORRIDOIO" else "RIPIANO",
                value = if ... else ..., ... ) }}
    BoxVoiceCommand(progress = progress, keyword = keyword.value)}
@Composable fun LocationPortionCard(
    label: String,
    value: String){
    Box{
        Text( text = label, textAlign = TextAlign.Center, ... )
        Divider(...)
        Text( text = value, fontSize = 70.sp, ... )}
}
```

Figura 6.11: Schermate posizione prelievo

In figura 6.11 si può infine apprezzare il risultato, e con esso la *flessibilità* e *modularità* offerta dalle funzioni composables: quando lo stato del sistema cambia (nel viewModel), con esso cambia anche *automaticamente* la visualizzazione della vista, la cui logica è definitivamente separata dalla gestione programmatica degli handlers e, invece, annegata in funzioni specifiche come PickingLocationPartialHighlighted(), LocationPortionCard(), BoxVoiceCommand().

6.4.3 View-Model

Il View-Model agisce da *intermediario* tra la vista e il modello: è responsabile di memorizzare e gestire i dati necessari alla vista, ma anche propagare al Model i cambiamenti effettuati dall'utente agli stessi. Inoltre, il VM è responsabile di elaborare i dati in arrivo dal modello e trasformarli in una forma facilmente usufruibile alla vista: in questo modo, sarà possibile un completo *disaccoppiamento* tra i due. Un ViewModel è dunque il luogo naturale in cui è possibile *bufferizzare* lo stato di visualizzazione: sono in genere forniti alla vista LiveData immutabili per osservarlo, mentre variabili private MutableLiveData consentono al VM di aggiornare il proprio stato dal modello.

Uno dei problemi a cui si va incontro nella definizione di un VM è il tempo: i metodi che hanno infatti a che fare con il modello (e quindi richieste HTTP),

potrebbero richiedere una certa quantità di secondi, un limite tecnico in Android, in quanto è vietato eseguire operazioni di *lunga durata* nel thread principale. La soluzione trovata a questo problema è ancora una volta le *coroutines*, che permetteranno di svolgere lunghe operazioni in thread secondari, in abbinamento a MutableLiveData per salvarne il risultato.

Il VM è stato creato tramite *delegated properties* grazie alla keyword "by" e activityViewModels(): così facendo, sarà possibile non solo *condividere* la stessa istanza di VM tra tutti i Fragment, ma anche *ricollegarla* (senza ricrearla) al fragment/activity distrutto e ricreato, per qualche motivo, durante il suo ciclo di vita (rotazione del dispositivo, navigazione, ecc...). Per evitare di referenziare qualcosa di non più esistente, il VM *non salva* mai qualcosa che possa contenere un riferimento al contesto dell'attività/fragment; unica eccezione è il contesto dell'applicazione che viene iniettato automaticamente nell'oggetto WM, perché è sicuro: il ciclo di vita dell'applicazione copre l'intera durata del processo.

Interazione tra View-Model e View

Prendendo a esempio il caricamento dell'immagine del prodotto da prelevare (l'operazione più lunga del modello), vedremo ora l'interazione tra View e View-Model. Il risultato è visibile all'immagine 6.12. Inizialmente nel Fragment si chiamano le funzioni sul WM per il setup dei dati, come l'articolo da prelevare e la relativa immagine: essendo operazioni asincrone, il loro termine è segnalato dal flag *imageLoadingTerminated* a true.

```
class PartialPickingViewModel @Inject constructor(
    private val pickingRepository: PickingRepository,
) : ViewModel{
    var currentStockImage: MutableState<Bitmap> //immagine
    lateinit var currentStock //articolo da prelevare
    fun getItemImage(path: String){
        viewModelScope.launch { //coroutine
            val leaResult = pickingRepository.getImage(path)
            if (leaResult.isSuccessful()) {
                if (leaResult.result != null)
                    currentStockImage = mutableStateOf(leaResult.result!!)
            } else showNotification(leaResult.error)
            imageLoadingTerminated.value = true //set del flag
        }}
```

Dal punto di vista della UI, la funzione Composable non riceverà l'oggetto Immagine, di cui non c'è traccia nei parametri, bensì sarà presa *direttamente* dal

VM quando sarà pronta: nel momento in cui *imageLoadingTerminated* è a true. Nel frattempo, durante il caricamento, sarà mostrato all'utente uno spinner per fargli capire di dover attendere che tutte le informazioni siano pronte per essere visualizzate.

```
@Composable fun PartialPickingItemComposable(
    viewModel: PartialPickingViewModel, ...){ ...
        if (viewModel.imageLoadingTerminated.value){
            PickingItemCard(...)
            BoxVoiceCommand(progress = progress, keyword = keyword)}
        else spinner()}
@Composable fun PickingItemCard(
    viewModel: PartialPickingViewModel, ...){
    Card(...){
        Image(bitmap = myImage.value!!.asImageBitmap(), ...)
        Divider(...)
        Text( text = viewModel.currentStock.stockLabel, ...)
        Text( text = viewModel.currentStock.quantity, ...)}
}
```

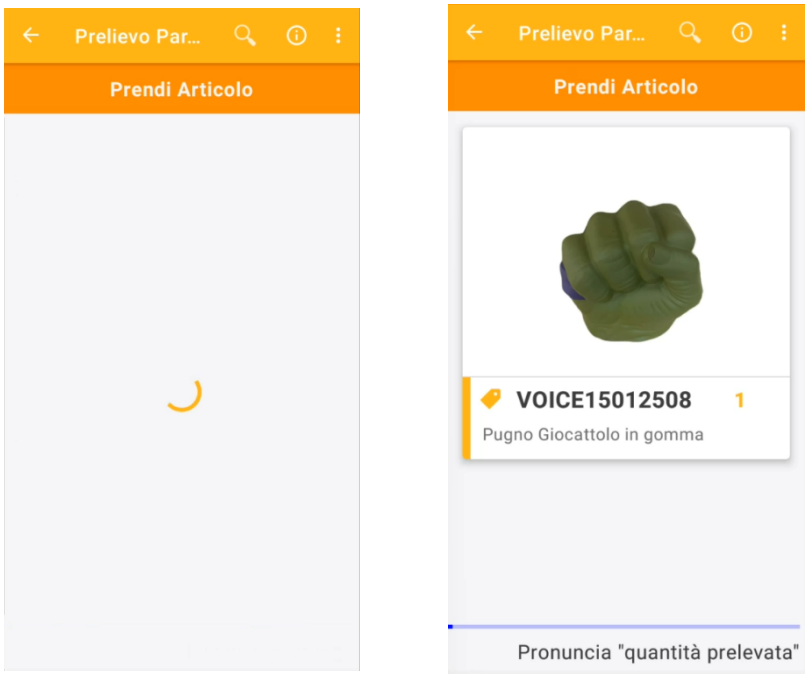

Figura 6.12: Visualizzazione prima e dopo il caricamento dell'immagine

Capitolo 7

Test preliminari con gli esperti

Dopo la *realizzazione* del sistema, il passo successivo è stata la fase di *sperimentazione*. Non essendo stato possibile interagire con gli utenti *target* del progetto (i pickeristi), i test condotti sono chiamati "preliminari", poichè hanno visto la partecipazione degli *esperti* di innovazione del settore logistico, con focus nell'ambito del task di prelievo. Questi ultimi sono stati selezionati sulla base del criterio dell'esperienza, il più ampia possibile, e con una ottima conoscenza delle esigenze dei pickeristi e delle loro mansioni.

Lo scopo dei test preliminari è in primis la *verifica* della stabilità e funzionalità del sistema realizzato, e in secundis la *valutazione*, *insieme* agli esperti, della sua usabilità e efficacia, oltre che la *rilevazione* di situazioni di inefficienza, errori e possibili miglioramenti.

7.1 Laboratorio "Area 42"

La sperimentazione del sistema di prelievo vocale si è svolta presso il laboratorio "Area42" di Reply, sito in Torino in via Nizza 250. Area42 è un laboratorio di grandi dimensioni, in cui si sperimentano e concretizzano le idee più creative e in svariati campi, sfruttando il potenziale delle tecnologie più innovative (robotica, mobilità avanzata e virtual reality). Qui è stata predisposta un'area specifica (in figura 7.1) a emulazione di un magazzino reale con corridoi, scaffali, ripiani, udc (scatoloni), il tutto etichettato da codici a barre che li individuano univocamente. Sono state così ricreate le condizioni di lavoro reali degli utenti finali del sistema, i pickeristi. I test sono stati condotti durante l'orario lavorativo normale, con luce diurna.

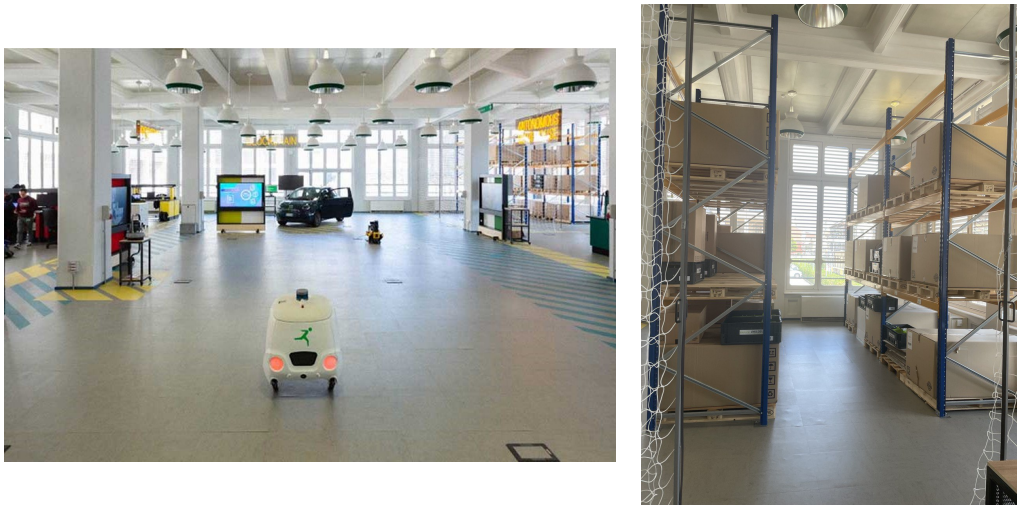

Figura 7.1: Foto del laboratorio Area42

7.2 Protocollo adottato per le prove del sistema

In questa sezione è riportata la descrizione della sperimentazione svolta, guidata da un particolare procedimento: si è infatti definito uno specifico protocollo per l'esecuzione dei test (visibile in Appendice C.2). Si sono dunque stabiliti *a priori* le modalità e gli aspetti più importanti su cui incentrare la sperimentazione: in tal modo sarà possibile replicare gli stessi test in futuro con altre realtà e avere un confronto attendibile tra i dati raccolti. Questo protocollo è la base per i test di usabilità da sottoporre a pickeristi *reali*, essendo basato sulle *stesse metodologie*, ma caratterizzato da domande più esplicite sui fattori da valutare, dal momento che diretto a esperti.

7.2.1 Gli aspetti principali

La definizione di un protocollo da utilizzare per la sperimentazione è stata incentrata sulla selezione degli aspetti del sistema più importanti da valutare: le operazioni svolte dai partecipanti sono state dunque pensate appositamente per la raccolta di tali informazioni, necessarie a stabilire un risultato complessivo sulla *riuscita* del progetto. Le stesse, possono essere catalogate nelle seguenti *proprietà* del sistema: *usabilità* (utilità, utilizzabilità, intuitività) e *funzionalità* (efficienza [%errori utente], prestazioni [%errori sistema]).

La raccolta delle informazioni è stata bilaterale: da una parte c'è stata una valutazione *qualitativa*, grazie a metriche come: inerenza commenti/domande, "positività" del linguaggio del corpo e espressioni, dall'altra una valutazione *quantitativa*,

che nasce da metriche più oggettive come: risposte questionario, tempi e medie errore di completamento.

7.2.2 Criteri

Il protocollo è stato disegnato sulla base dei due grossi limiti di questa sperimentazione: la mancanza di utenti reali (i pickeristi) e l'impossibilità di parlare liberamente durante i test, a causa del riconoscimento vocale attivo (disattivarlo/attivarlo continuamente sarebbe andato a discapito della fluidità dell'esperienza). Il primo limite (risolto con la partecipazione degli esperti) è stato trasformato in vantaggio, in quanto ha permesso domande dirette e esplicite sulle proprietà da valutare; la soluzione al secondo è stata invece l'adozione (quando necessario) della metodologia "Cooperative evaluation"[1], piuttosto che un "Think Aloud"[2], dal momento che il primo permette momenti di discussione più sparsi per eventuali chiarimenti e, in aggiunta, incoraggia il partecipamente a una critica "attiva" del sistema (e in questo caso le critiche "esperte" possono essere illuminanti).

Nel porre le domande, altri criteri usati sono:

- nelle domande a scelta multipla si è predisposto un numero di *opzioni pari*, nel caso in cui sia preferita una risposta "sbilanciata", piuttosto che una "neutrale" o "nel mezzo", da parte del partecipante;
- nelle domande aperte, come quelle di debriefing, si è posta grande attenzione a porle in modo *imparziale* e senza bias, in modo da non privilegiare un certo tipo di risposta, seppur auspicata;
- nel questionario finale c'è *alternanza* di domande poste in modo "positivo" e "negativo", per costringere il partecipante a leggere e rispondere sempre con attenzione.

7.2.3 Descrizione del test

Durante il test si sono istruiti i partecipanti all'uso del sistema, al fine di valutarne soprattutto l'usabilità, intuitività e funzionalità. Grazie al *facilitatore*, colui che ha avuto il compito di guidare i partecipati a compiere i vari task previsti, il test si è svolto velocemente e senza particolari intoppi. Oltre alla raccolta dati in forma "cartacea" tramite questionari, tutta la durata della sperimentazione è stata

[1] Cooperative evaluation è una variazione del Think Aloud, in cui partecipante e facilitatore collaborano durante la valutazione, ponendosi domande a vicenda.

[2] Think Aloud è una tecnica di valutazione che consiste nel chiedere all'utente di eseguire una serie di attività e farlo pensando ad alta voce, per spiegare cosa sta facendo in ogni fase e perché.

registrata, affinchè fosse possibile valutare, in un secondo momento e in tutta calma, sia le metriche quantitative, che qualitative.

Preparazione al test

Le operazioni si sono svolte all'interno di un laboratorio, il già citato Area42. La sperimentazione è stata effettuata con la collaborazione di tre "esperti" di innovazione nel settore logistico, con molta esperienza funzionale e/o tecnica nel task di picking, e altresì conoscenza diretta dei pickeristi.

Per poter mostrare tutte le funzionalità del progetto, il "magazzino" dell'Area42 è stato allestito da diversi udc, e al loro interno, vari tipi di articolo da prelevare in diversa quantità: il tutto etichettato con: codici a barre, coordinate e cifre di controllo delle posizioni. In particolare si sono disposti più articoli del necessario, per incentivare una "ricerca attiva" degli articoli, che non ci sarebbe stata se gli stessi fossero stati facilmente individuabili. Il risultato è visibile in figura 7.2.

Figura 7.2: Visuale di un udc (test)

Dopo questa preparazione, è stato necessario mappare i scaffali e il loro contenuto sul WMS, per poter schedulare prelievi reali da fare, e in aggiunta creare dei profili fittizi da pickerista. Prima dello svolgimento del test, a ogni partecipante è stato affidato un dispositivo Android (un comune smartphone consumer) e rispettive credenziali. Successivamente è stato spiegato loro il motivo della sperimentazione, le modalità, ed è stato fatto compilare (ad ognuno dei partecipanti) un breve questionario iniziale per meglio classificarne i profili: sono stati richiesti i nominativi, età, la "specialità" svolta in campo logistico, la regione di provenienza per il tracciamento degli accenti, e una quantificazione dell'esperienza maturata, sia dal punto di vista professionale che nell'uso di dispositivi Android.

Svolgimento del test

Prima dell'inizio della missione di prelievo, il facilitatore ha spiegato al partecipante i comandi vocali "sempre attivi", da poter usare per il setup del text-to-speech (volume, velocità) o all'occorrenza durante i movimenti in magazzino (pausa, ripeti, aiuto, vocabolario). Dopodichè si è chiesto di aprire l'app, fare il login con le credenziali consegnate, scegliere un task di prelievo e partire; il facilitatore ha quindi sottolineato che, dopo di questo, non bisognerà più prestare particolare attenzione al telefono, ma basterà aver cura di portarselo sempre dietro, ad esempio mettendolo in tasca.

Da qui in poi il partecipante è stato guidato, dalle istruzioni vocali del sistema, nella simulazione di casi d'uso comuni, e gli sono stati proposti nel mentre dei piccoli quesiti da risolvere, come ad esempio trovare il modo per regolare il text-to-speech o mettere in pausa il speech-to-text per poi riattivarlo, capire quindi i relativi feedback. In questi casi è stata prevista una "cooperazione" tra utente e facilitatore, per favorire la buona riuscita e una critica a "caldo". Si è inoltre agevolata l'*esplorazione* del sistema proponendo una situazione di errore molto comune (una quantità di prelievo sbagliata) e chiedendo di ottenere più informazioni dal sistema, se ad esempio non si conosce una posizione di magazzino da raggiungere. Per concludere, il partecipante è stato volutamente lasciato solo nel proseguimento di tutte le successive prese della missione, per verificare il grado di facilità e di intuitività nell'uso del sistema, ma anche valutare la sua robustessa e efficienza di fronte a eventuali casi anomali (errore del pickerista).

Figura 7.3: Foto dei partecipanti durante i test

Capitolo 8

Risultati dei test e obiettivi raggiunti

Il penultimo capitolo di questo lavoro è uno dei più importanti, dal momento che viene qui definito il *grado di successo* raggiunto dal sistema realizzato, attraverso la riflessione sugli obiettivi conseguiti (sia nella realizzazione, sia durante la sperimentazione in laboratorio) e il paragone di questi ultimi con i requisiti inizialmente definiti. E' importante sottolineare che aver potuto provare il sistema con utenti reali in un "vero" magazzino, ha aumentato l'attendibilità dei risultati ed è un primo passo verso nuovi sviluppi futuri. Di seguito, ci sarà dunque l'esposizione dei dati raccolti durante i test (in forma riassuntiva) per far capire come si è giunti ai risultati del sistema; questi ultimi saranno successivamente confrontati con i requisiti delineati prima della progettazione nel capitolo 3, e il tutto riportato in tabella per una consultazione più agevole.

8.1 La raccolta dei risultati

8.1.1 Questionario iniziale: profili dei partecipanti

Analizzando i questionari iniziali è possibile ricavare il profilo dei partecipanti al test. Si tratta di due esperti tecnici e un funzionale, tutti e tre di sesso maschile tra i 26-35 anni, e con un'esperienza professionale quantificabile tra i 2-5 o 5-10 anni: due di loro affermano di conoscere "molto" le esigenze dei pickeristi, e solo uno di loro "abbastanza". Per quanto riguarda invece l'accento, due sono di origine italiana e rispettivamente siciliana, piemontese, il terzo croato; l'esperienza con Android è eccellente per tutti e tre.

8.1.2 Risultati del test

Il settaggio della volume/velocità del parlato è stato effettuato correttamente da tutti i partecipanti e in particolare due sono riusciti a farlo in breve tempo, capendo e reagendo in modo ottimale a tutti i feedback del sistema. Solo uno su tre, però, è riuscito a trovare velocemente sia il modo per stoppare il riconoscimento vocale, che riattivarlo, mentre gli altri due hanno avuto necessità di indizi/collaborazione del facilitatore. Il comando "ripeti" si è invece dimostrato molto intuitivo, dal momento che tutti e tre gli utenti sono stati capaci di far ripetere al sistema le istruzioni, senza ricevere suggerimenti.

Per quanto riguarda invece la dichiarazione del codice a barre dell'udc, due su tre lo hanno pronunciato, mentre il terzo la ha digitato; dei primi due, uno è stato in grado di annullare velocemente le ultime cifre, mentre il secondo ci ha impiegato un po più di tempo, ma alla fine ha pronunciato comunque il comando giusto, senza particolari indizi. Nel proseguimento del prelievo, al momento di raggiungere una particolare posizione, tutti e tre hanno pronunciato correttamente le cifre di controllo, ma due di questi non hanno ricordato il comando "aiuto" in autonomia.

Nell'ascolto dei barcode, sia dell'udc che dell'articolo da prelevare, non ci sono stati particolari problemi, e tutti hanno individuato correttamente quanto richiesto. Anche il comando "descrizione" è stato facilmente usato in modo proficuo per accertarsi di aver prelevato la merce esatta, in un solo caso su consiglio del moderatore.

Infine, la gestione errore degli articoli è stata usata in modo ottimale e intuitivo da tutti e tre i partecipanti, ma probabilmente il risultato migliore di tutti è stato il conseguimento del 100% di accuratezza in tutte e tre le simulazioni di prelievo (6 prese nella missione).

8.1.3 Questionario finale: l'opinione degli esperti

In questa sezione è riportato un quadro complessivo dei dati raccolti dai questionari finali, arricchite talvolta da ulteriori considerazioni che sono state fatte nella fase finale di debriefing. E' di seguito riportato il parere dei partecipanti al test, ai quali è stata data la possibilità di rispondere alle domande secondo una scala Linkert da 1 a 4, dove 1 indicava il risultato peggiore e 4 quello migliore, in numero pari per favorire un "sbilanciamento" della risposta in senso negativo o positivo.

Ho trovato il sistema facile da usare
Dall'analisi delle risposte date a questa domanda, si evince che tutti e tre gli esperti hanno valutato il sistema come facile da usare, e, tra questi, due hanno mostrato particolare entusiasmo (hanno selezionato "fortemente d'accordo").

Il dispositivo mi è stato di intralcio durante i prelievi

La risposta dei tre partecipanti è in questo caso la stessa: "fortemente disaccordo", a conferma che la comodità di poter mettere il dispositivo in tasca durante i movimenti in magazzino, porta l'operatore a non preoccuparsene.

L'interfaccia è intuitiva
In questo caso abbiamo due risposte più contenute con "d'accordo" e una più entusiasta con "fortemente d'accordo". Relative motivazioni saranno spiegate nella sezione di debriefing.

Ho trovato le varie funzioni del sistema poco integrate
Secondo i partecipanti al test, questa affermazione è falsa, e sostengono che, al contrario, lo sono. Nessuno ha però selezionata la risposta più positiva, ossia "fortemente disaccordo". I motivi di tale scelta, come poi specificato, risiedono in una interfaccia multimodale che potrebbe essere ulteriormente migliorata per essere più efficiente.

Avevo paura di sbagliare durante il prelievo
In questo caso le risposte sono diverse, in quanto due partecipanti hanno detto di essere disaccordo, mentre uno di questi si è detto d'accordo. L'ultimo esperto ha poi affermato di aver selezionato questa risposta negativa, perchè ha trovato i feedback delle cifre di controllo errate, poco indicative.

I pickeristi riceverebbero un addestramento iniziale più rapido
Due esperti ritengono questa affermazione falsa, dichiarandosi disaccordo, mentre il terzo si è detto d'accordo. Secondo i primi due, infatti, l'addestramento con dispositivo ma senza vocale, la soluzione più comunemente usata, è altrettanto veloce da imparare. Se la si vede però su un discorso di "prenderci la mano", hanno sostenuto che pensano che il vocale possa effettivamente ridurre i tempi, soprattuto se il pickerista non è molto avezzo alla tecnologia.

Il sistema di prelievo non risponde alle esigenze dei pickeristi
Tutti e tre i partecipanti hanno espresso un parere negativo, selezionando l'opzione "disaccordo", e uno tra questi ha mostrato più fiducia con "fortemente disaccordo".

Sono favorevole all'introduzione in magazzino di soluzioni simili
In questa ultima domanda, sorprendemente, ho ricevuto solo pareri negativi con la risposta "disaccordo". Nel proseguio, sono state fornite le motivazioni e tutti e tre hanno espresso lo stesso dubbio: seppur sono favorevoli a un'introduzione in magazzino, in non tutti i casi è la scelta ideale, in quanto variano condizioni di velocità, spazio, dimensioni articoli: se ci sono tanti articoli di piccola dimensione (di cui molti simili tra loro) posti in un singolo udc, o ancora se lo spazio non è

quello enorme di un classico magazzino, ma piuttosto una postazione, allora questo sistema vocale non andrebbe bene.

8.1.4 Confronto con gli operatori

Il dibattito informale con gli esperti, creatosi durante e dopo il test, è stato molto interessante e soprattutto molto utile. Le impressioni, espresse dai partecipanti, sono state positive e di entusiasmo nei confronti di un sistema flessibile e innovativo come quello vocale. L'unico dubbio emerso a tal proposito è il proporre effettivamente questo prodotto a quei pickeristi che da molti anni fanno questo mestiere, in genere poco avezzi al cambiamento, che, al contrario, è respinto con forza. Sicuramente con un pickerista novello non si avrebbero gli stessi problemi, in quanto aperto a imparare qualsiasi genere di sistema, e quello realizzato potrebbe senza dubbio agevolarlo a ridurre, per quanto possibile, i tempi di formazione.

In aggiunta tutti gli esperti hanno concordato che le modalità permettono un prelievo hands-free, dal momento che il dispositivo può essere messo in disparte dove non è di intralcio all'operatore (in questo caso in tasca), ma comunque nel raggio d'azione del Bluetooth. Ciò permette non solo zero intralci manuali, e quindi maggiore sicurezza in magazzino, ma anche zero pensieri sul ricordare dove si è messo il "dispositivo", che non potrà più essere dimenticato.

Per quanto riguarda invece la domanda sull'utilità e la velocità di prelievo, la risposta è stata univoca e positiva, dal momento che tutte le funzioni necessarie al prelievo sono state implementate e le informazioni in cuffia permettono di inviduare precocemente il prodotto; molto apprezzata in tal senso il comando vocale "descrizione" per ascoltare le generalità dell'articolo da prelevare in cuffia. Secondo alcuni, però, la reattività del riconoscimento vocale (necessario per la conferma) è un limite alla velocità di prelievo, che potrebbe essere migliorata da un picking "spara barcode", qualora fatto senza errori.

Anche nella domanda riguardante l'usabilità della soluzione, sono stati riscontrati pareri molto positivi, grazie all'uso di un'interazione vocale semplice e naturale. Si sono però riscontrati anche dubbi e difficoltà durante il processo. Alcuni esperti hanno infatti lamentato la mancanza di un doppio check durante il prelievo, visivo oltre che vocale: nonostante le informazioni in cuffia, avrebbero voluto anche vederle oltre che sentirle; un suggerimento alternativo che è stato dato, è la pronuncia delle ultime cifre del barcode (oltre che la quantità) alla fine della presa, come ulteriore conferma vocale. A conferma di questa "mancanza visiva" lamentata, più volte i partecipanti si sono fermati per riprendere in mano il cellulare e leggere i dati sullo schermo.

Proprio per quest'ultimo motivo, la risposta alla domanda sull'integrazione interfaccia vocale/grafica ha trovato pareri piuttosto negativi. Probabilmente

questo giudizio può essere interpretato con la "deficienza" di abitudine nell'affidarsi, in modo primario, all'udito: le moderne interfacce sono perlopiù grafiche.

Per concludere, i partecipanti sono entusiasti nel dire che questo sistema vocale potrebbe migliorare le condizioni di lavoro dei pickeristi, poichè ritenuto più "divertente" e meno "alienante" di tutti gli altri, benchè ci abbiano tenuto a sottolineare che è adottabile *solo* in alcune configurazioni di magazzino. La soluzione è infatti adatta in presenza di celle frigorifere, in cui è previsto l'uso di guanti molto pesanti (contro il gelo delle mani) che non permettono il touch su schermo, o ancora magazzini con grandi superfici e molte aree diverse, dove c'è la necessità di qualcosa che aiuti nell'orientamento, ma, allo stesso tempo, è disponibile lo spazio per lavorare abbastanza "lontani", dal momento che la voce del pickerista x potrebbe disturbare il sistema del pickerista y.

8.1.5 Requisiti: confronto

A conclusione del lavoro di tesi, è possibile ora confrontare i requisiti (delineati inizialmente nel paragrafo 3.3) con i risultati estratti dalla sperimentazione. Sarà così possibile valutare il "grado di soddisfazione" dei singoli requisiti, e capire in quali ambiti sarà invece necessario introdurre dei miglioramenti per aggiustare il "tiro".

Prendendo in considerazione i requisiti di *usabilità*, possiamo valutare R1.1 e R1.2 totalmente soddisfatti, dal momento che il sistema permette un prelievo a mani libere, senza intralci o dimenticanze, grazie al binomio "dispositivo in tasca più cuffie". Discorso a parte invece R1.3 e R1.4, dove invece il raggiungimento è solo parziale, dal momento che sono stati trovati problemi di usabilità dai partecipanti e che l'interazione visiva è solo limitata ma non completamente esclusa.

Per quanto riguarda invece la parte sul *supporto ai lavoratori*, possiamo considerare R2.1 soddisfatta totalmente dal 100% di accuratezza raggiunto dai partecipanti, e allo stesso modo R2.2, grazie alle informazioni ricevute dal pickerista sul numero totale di colli da prelevare durante la missione. Ancora, i requisiti R2.4, R2.5 e R2.7 sono raggiunti totalmente da un appropriato design del sistema, mentre R2.3 e R2.6 sono solo parziali: nel primo caso la motivazione è una velocità di prelievo sì rapida, ma comunque più lenta della modalità scanner barcode (se effettuata senza errori), invece nel secondo è una formazione solo teoricamente più breve.

Infine, tutti i requisiti riguardanti le *prestazioni e efficienza* sono stati raggiunti da una progettazione mirata della soluzione; solo il R3.2 ha avuto una soddisfazione parziale, causata da errori di riconoscimento vocale durante la sperimentazione: in alcuni casi il sistema non ha riconosciuto correttamente il parlato. I risultati del confronto sono riportati in tabella 8.1 per semplicità di consultazione.

Categoria	#	Descrizione	Prior.	Soddisf.	Motivo
	R 1.1	Prelievo hands-free	1	Totale	
	R 1.2	Il sistema non deve intralciare	2	Totale	
	R 1.3	Il sistema deve essere facile da usare per tutti	2	Parziale	Problemi individuati durante il test
	R 1.4	Il sistema non deve affaticare gli occhi	3	Parziale	L'interazione visiva è molto minore ma continua a esistere
	R 2.1	Elevata accuratezza di prelievo	1	Totale	
Supporto per i lavoratori	R 2.2	Pickerista autonomo di decidere la tipologia di udc	2	Totale	
	R 2.3	Il sistema deve permettere tempi di completamento prelievo rapidi	1	Parziale	L'uso di scanner barcode senza errori è più rapido
	R 2.4	Procedure di gestione errore automatiche	3	Totale	
	R 2.5	Prevenzione errori	2	Totale	
	R 2.6	Il sistema deve permettere tempi veloci di formazione	3	Parziale	Non è previsto un sensibile miglioramento
	R 2.7	Installazione "plug&play"	2	Totale	
	R 3.1	Android e nessun hardware specifico richiesto per il funzionamento	2	Totale	
Prestazioni e Efficienza	R 3.2	TTS facilmente comprensibile e a STT reattivo e accurato	1	Parziale	Ci sono stati errori di riconoscimento vocale
	R 3.3	Impiego risorse del device in modo efficiente	1	Totale	
	R 3.4	Il sistema deve poter funzionare offline	1	Totale	

Tabella 8.1: Confronto diretto tra requisiti e risultati raggiunti

8.2 Costi del sistema

Nel complesso, i pezzi hardware usati per la costruzione di un singolo prototipo hanno avuto un costo totale leggermente superiore a 250 €, che è un prezzo molto minore di quello di soluzioni alternative, che si aggirano invece sui 2000-2500 €. Inoltre, questo costo è da sostenere nel caso di ingresso sul mercato, mentre in caso di cambio tecnologia da prelievo a prelievo vocale, i costi sono completamente azzerati dal riuso di una dotazione preesistente. L'obiettivo secondario del progetto si può dunque ritenere soddisfatto.

Capitolo 9

Conclusioni

L'obiettivo di questa tesi è stato quello di esplorare nuove possibilità di interazione tra pickerista e mobile, che ponesse al centro una comunicazione vocale e hands-free. A tale scopo ho sviluppato un sistema con un'interfaccia multimodale e focalizzata sul canale percettivo sonoro: quest'ultimo permette, in un primo momento, l'ascolto delle informazioni necessarie al prelievo di un determinato articolo, e successivamente, la restituzione di un feedback attraverso il parlato.

Se da una parte, *funzionalità* e *usabilità*, sono state poste al centro del progetto, dall'altra ci si è posti anche il traguardo di un progetto *modulare* ed *economico*: questa soluzione vuole infatti essere *innovativa*, sia dal punto di vista tecnico che commerciale, grazie a un sistema all'avanguardia e in grado di abbassare drasticamente i costi di transizione tecnologica richiesti dai competitor.

L'effetto sperato è in primis una ricaduta positiva sul lavoro quotidiano del pickerista, maggiormente soddisfatto dall'uso di un sistema semplice e intuitivo, ma secondariamente anche una larga adozione in tutti quei magazzini medio-piccoli, che finora non hanno avuto accesso a queste costose tecnologie, convinti da un sistema a basso prezzo, ma che riuscirà comunque a garantire un sensibile incremento di produttività, oltre che maggiore sicurezza, in magazzino.

Al fine di raggiungere il risultato atteso, la creazione del sistema vocale è stata approcciata per gradi:

- estrapolazione dei requisiti dalle personas;
- analisi delle soluzioni vocali in commercio;
- progettazione del sistema;
- implementazione;
- testing funzionale e usabilità (preliminare) in un reale caso di prelievo.

Il lavoro è iniziato con l'analisi di testimonianze dirette e indirette sul "chi è" un pickerista e le sue mansioni; da qui la ricostruzione di profili fittizi (personas) e successiva estrapolazione dei requisiti del sistema. Questa fase che si è rivelata di notevole importanza, in quanto ha sia posto un termine di paragone ai risultati di fine lavoro, ma ha anche spianato la strada ai passi successivi, dando delle sorta di "linee guida" da rispettare, per far si che la soluzione finale fosse conforme alle aspettative iniziali. Il progetto è proseguito con la ricerca delle soluzioni vocali attualmente sul mercato e quindi il loro diretto confronto (a "parità di condizioni"), per scegliere il motore vocale di riferimento. Sulla base di quest'ultimo, è stata progettata un'architettura software modulare (framework + applicazione): ciò ha implicato il focalizzarsi sulle funzionalità da supportare, definite in precedenza, e la conseguente definizione dei componenti da usare nel sistema e relative caratteristiche primarie da adottare. Oltre a questo, partendo dal processo *as is*, si è affrontata la prototipazione dell'interfaccia e, parallelamente, design del processo *to be*; anche in questo caso c'è stato un grande aiuto dagli esperti, che si sono resi partecipi a una valutazione euristica. La fase di implementazione è venuta di conseguenza: la difficoltà principale è stata integrare le diverse tecnologie previste, affinchè tutti i moduli del progetto potessero comunicare tra di loro in modo ottimale. Per ultima cosa, ma non meno importante, c'è stata la possibilità di sperimentazione con utenti reali (non pickeristi, ma sempre esperti), che, come osservato nel capitolo precedente, ha fatto rilevare la buona riuscita del progetto con la soddisfazione totale di molti requisiti prefissati.

9.1 Riflessioni

Questo tesi si è rivelata una sfida motivante, che mi ha regalato molte soddisfazioni. Le motivazioni sono nate soprattutto dalla possibilità di avere un riscontro pratico sull'utilità del lavoro svolto: aiutare lo svolgersi del lavoro quotidiano dei magazzinieri, migliorerà di conseguenza anche la produttività del centro logistico di impiego e, a catena, gli affari delle aziende, invogliate peraltro all'innovazione da costi di transizione tecnologica bassissimi, se non nulli. Inoltre, l'aver potuto provare il sistema fisicamente, una volta completato, ha portato maggiore maturità e spunti per migliorare: sia dal punto di vista funzionale, che dell'usabilità. In particolare, è stata riscontrata l'utilità di quanto fatto: la qualità del lavoro di un pickerista viene sicuramente migliorata rispetto alle classiche modalità carta stampata e/o dispositivo solo con tocco, grazie a un prelievo senza mani, che ha dimostrato maggiore accuratezza, sicurezza, praticità: questi, dall'inizio, i principali punti critici su cui si è insistito. Adesso, un pickerista, non avrà più problemi a spostarsi in magazzino e, nel frattempo, continuare a interagire con il proprio dispositivo e essere supportato da efficaci metodi di prevenzione e gestione errore.

9.2 Sviluppi futuri

Il sistema implementato si presta molto bene ad essere ampliato e migliorato, sia dal punto di vista prestazionale, che dell'usabilità.

9.2.1 Sviluppo: prestazioni e efficenza

Al fine di realizzare il cuore dell'applicazione di prelievo, ossia il framework vocale, durante la fase di scouting si è privilegiata la ricerca di soluzioni vocali aderenti a certi parametri, ma in primis open-source: l'intento, in questo caso, era di non rallentare il lavoro di tesi con la richiesta di licenze d'uso a terzi, ma procedere senza intoppi nella progettazione e sviluppo. Questa limitazione ha pilotato la ricerca, di fatto *escludendo* opzioni praticabili, poichè legate all'acquisto della relativa licenza.

Un possibile sviluppo futuro potrebbe, dunque, prendere in considerazione **soluzioni vocali a pagamento** che, da quanto ho potuto osservare a suo tempo, sono mediamente più reattive e accurate delle sorelle licenziate open-source. E' importante sottolineare che la modularità dell'architettura software del sistema e l'iniezione di dipendenze (Hilt) *favoriscono* volontariamente il cambio implementazione del motore vocale (la sua evoluzione) e *minimizzano* il codice sorgente da modificare nell'applicativo di prelievo.

9.2.2 Sviluppo: usabilità

Per eventuali sviluppi a miglioramento della intuitività, utilità, semplicità d'uso percepita dall'utente, sono due, in particolare, i fronti da valutare:

Utente - non è stato eseguito un reale studio sull'utente finale. Il sistema realizzato è stato progettato e testato, evitando accuratamente una vera interazione con il pickerista, dapprima immaginato nelle "personas", successivamente sostituito da esperti nelle fasi di prototipazione e test, in virtù di una loro approfondita conoscenza di processi e esigenze. Si ritiene che un'eventuale studio, a stretto contatto con l'utente, possa sicuramente tornare utile per mettere a punto nuove migliorie da apportare all'interazione uomo-macchina, che rendino il sistema maggiormente usabile.

Visualizzazione - durante i test è emersa l'esigenza di poter vedere le informazioni (oltre che sentirle), il tutto continuando ad avere le mani libere. Una possibile soluzione che vada in tal senso è l'aggiunta nel sistema di un nuovo componente *indossabile* sulla mano: il "ring-scanner", un dispositivo integrante schermo e scanner barcode (in figura 9.1). Questa nuova introduzione, seppur a discapito dei costi complessivi del sistema e della sua interoperabilità, potrebbe essere un'arma a doppio taglio da sfruttare: da una parte permetterebbe la

ridondanza delle informazioni ascoltate in cuffia tramite una visualizzazione a video comoda da osservare anche nel caso in cui le mani siano occupate, dall'altra di *scannerizzare* i codici a barre degli articoli prelevati come ulteriore doppio check, per una più robusta prevenzione errore. Dall' impiego di un ring-scanner avremmo dunque un sistema più usabile e flessibile, ma di contro maggiore complessità e costi.

Figura 9.1: Ring-scanner indossabile, con schermo integrato

Appendice A

Scouting

A.1 Testimonianze indirette raccolte

Durante la fase di raccolta requisiti, sono state consultate diverse fonti di genere articolo, o filmato, per conoscere meglio gli addetti al prelievo in magazzino, il loro lavoro. In particolare, si definiscono queste testimonianze come "indirette" in quanto non sono state redatte allo scopo di divulgazione, ma per motivi diversi come ad esempio pubblicità, annunci di lavoro, template per aiutare i responsabili di magazzino a descrivere il lavoro svolto da un pickerista. Sono stati quindi consultati:

- Volunteer Training: Order Picking, Daily Bread Food Bank, `https://www.youtube.com/watch?v=SZoKRrtSmUM&t=786s`
- Prelievo vocale con dispositivo multifunzione, Zetes Group, `https://www.youtube.com/watch?v=jwZGW2No878`
- I numeri uno come NUMBER 1 scelgono Vocollect Voice, kfitrading, `https://www.youtube.com/watch?v=DO-rXScx7WU`
- CaseHistoryMotorolaMC75Uhaul, kfitrading, `https://www.youtube.com/watch?v=Kuf9BcU6pfM`
- Vocollect Number 1 Abbreviated Case Study, kfitrading, `https://www.youtube.com/watch?v=gZm13PpqOhA`
- Cosa fa l'addetto al picking, LogisticaIT, `https://logisticait.com/cosa-fa-laddetto-al-picking/`
- Offerta lavoro, `https://it.indeed.com/offerte-lavoro?q=Pickeristi&redirected=1&vjk=b787b7f61db75fb4`

- Picker Job Description Template,https://www.betterteam.com/picker-job-description#:~:text=Picker%20Responsibilities%3A,orders%20as%20requested%20by%20management.

A.2 Motori Vocali considerati

Durante la fase di scouting, sono state consultate le seguenti librerie SDK Android:

- Lydia voice, Lydia, https://www.lydia-voice.com/it/
- IT works Voce, IT works, http://www.it-works.it/vo-CE/
- Vocalize, KFI, https://www.vocalize.eu/
- Dematic iQ Workflow, Dematic, https://www.dematic.com/it-it/
- Vocollect, Honeywell, https://sps.honeywell.com/it/it
- German Autolab, https://www.germanautolabs.com/sdk
- Kaldi, Volsk API, https://alphacephei.com/vosk/
- PicoVoice, https://picovoice.ai/platform/cat/
- KeenASR , KeenResearch, https://keenresearch.com/
- Mozilla DeepSpeech, https://github.com/mozilla/DeepSpeech
- Algolia, https://www.algolia.com/industries-and-solutions/voice-search/
- Ispeech, http://www.ispeech.org/api/
- Google Cloud's Speech-to-Text, Google, https://cloud.google.com/speech-to-text?hl=it
- Native Android Speech Synthesis, Google, https://developer.android.com/reference/android/speech/SpeechRecognizer
- Acapela, https://www.acapela-group.com/solutions/acapela-tts-for-android/
- VoiceRSS, https://www.voicerss.org/sdk/android.aspx
- eSpeak NG , https://github.com/espeak-ng/espeak-ng#license-information
- AndroidMaryTTS, https://github.com/AndroidMaryTTS/AndroidMaryTTS
- ReadSpeaker, https://www.readspeaker.com/it/
- Voicepods, https://www.voicepods.com/
- Mimic, MycroftAI, https://mimic.mycroft.ai/

Appendice B

Valutazione euristica

B.1 Euristiche usate

Le euristiche usate durante la valutazione euristica coincidono con quelle create da Jacob Nielsen. Ecco qui la lista completa:

H1 - Visibilità dello stato del sistema;

H2 - Match tra il sistema e il mondo reale;

H3 - Controllo e libertà dell'utente;

H4 - Consistenza e standard;

H5 - Prevenzione dell'errore;

H6 - Riconoscimento piuttosto che ricordo;

H7 - Flessibilità e efficienza d'uso;

H8 - Design estetico e minimalista;

H9 - Aiutare gli utenti a riconoscere, diagnosticare e recuperare dagli errori;

H10 - Aiuto e documentazione;

NH - Problemi non-euristici;

B.2 Lista di task

Durante la valutazione euristica dei due prototipi, è stata data ai valutatori una lista di task da eseguire, con lo scopo di aiutarli a sperimentare le interfacce e valutarne l'efficacia e intuitività. La lista usata nei test è di seguito riportata:

1. Hai appena iniziato il tuo turno in magazzino da pickerista, e hai già scelto una missione da fare. Iniziala e inserisci un codice udc casuale: non è importante ai fini della ricerca;
2. Mentre lo inserivi, ti sei accorto di aver sbagliato il quarto carattere. Cerca di rimediare;
3. Dopo aver messo il cellulare in tasca, ti accorgi di non sapere come raggiungere la posizione richiesta. Cerca di reperire informazioni;
4. Dopo aver raggiunto la posizione, qui trovi le cifre di controllo 67;
5. Vedendo meglio le etichetti sugli scaffali, hai sbagliato. Le cifre di controllo sono 45. Usale per passare al prossimo step, e trova quindi l'articolo da prelevare;
6. Preleva l'articolo indicato con una quantità minore;
7. Finisci i prelievi rimanenti e termina la missione;

B.3 Risultati

#Problema	#Prototipo	#Euristica	#Gravità	Descrizione
1	P1	H3	5	Nella prima schermata, in cui va inserito il codice dell'udc, potrebbe non essere agevole interagire con schermi touch se non si è dotati di guanti touch.
2	P1	H7	4	Spezzare la posizione in singole coordinate (da schermata 3 a 6) rallenta le operazioni, soprattutto quando si conosce perfettamente la struttura del magazzino e il processo di prelievo.
3	P1	H7	5	Alla schermata 8, non è intuitivo/comodo riprendere in mano il dispositivo quando si hanno problemi con la quantità da prelevare.
4	P2	H1	3	Nella prima schermata, in cui va inserito il codice dell'udc, non è chiaro se il sistema è da subito in ascolto oppure se si debba attendere.
5	P2	H5	2	Durante la pronuncia dei codici a barre, l'ultima parte non è seguita da nessuna pausa e anzi, parte subito la parte successiva della descrizione tts. E' poco chiaro se l'utente abbia individuato l'articolo in così breve tempo.

6	P2	H6	3	Indicare solo il codice a barre di un articolo da prelevare non è agevole quando ci sono numerevoli oggetti, di piccole dimensioni, in una sola scatola.
7	P2	NH	5	Nelle operazioni di gestione errore durante il prelievo, non è previsto il caso in cui non ci sia del tutto giacenza in magazzino.
8	P1/P2	H6	5	Se la descrizione tts termina e l'utente non ha prestato attenzione (o ha dimenticato qualche dettaglio), non c'è modo di risentirla.
9	P1/P2	H8	5	Ripetere ogni volta *"pronuncia avanti per continuare"* può risultare noioso, oltre che far perdere tempo ogni volta all'operatore.

Tabella B.1: Risultati valutazione euristica

Appendice C

Protocollo per la sperimentazione del sistema

C.1 Obiettivi dei test preliminari

Lo scopo dei test preliminari è in primis la *verifica* della stabilità e funzionalità del sistema realizzato, e in secundis la *valutazione* della sua usabilità e efficacia, oltre che la *rilevazione* di situazioni di inefficienza, errori e possibili miglioramenti, *insieme* agli esperti.

C.2 Usability Testing Plan

C.2.1 Partecipanti

La popolazione target dei test preliminari sono gli *esperti* nell'innovazione del settore logistico, caratterizzati da ruolo tecnico e/o funzionale e con focus nell'ambito del task di prelievo. Questi dovranno inoltre avere un'esperienza più ampia possibile e una ottima conoscenza delle esigenze dei pickeristi e delle loro mansioni.

C.2.2 Attrezzatura necessaria

- Stanza fisica;
- Attrezzatura di magazzino reale (corridoi, scaffali, ripiani, udc, articoli), il tutto etichettato con barcode e informazioni utili (cifre di controllo);
- Dispositivo Android con connessione Wi-Fi e sensore di prossimità;
- Cuffie Wireless Bluetooth con microfono;

- Profilo fittizio abilitato al prelievo;
- Backend WMS attivo;
- Videocamera;
- Pc per prendere nota;
- Cronometro;

C.2.3 Artefatti

- Informativa legale;
- Questionario iniziale;
- Questionario finale;

C.2.4 Informativa legale

Mi sono liberamente offerto volontario per partecipare a questo esperimento. Sono stato informato in anticipo quali saranno i miei compiti e quali procedure saranno seguite. Mi è stata data l'opportunità di porre domande e le risposte sono state di mia soddisfazione. Sono consapevole di avere il diritto di revocare il consenso e di interrompere la partecipazione in qualsiasi momento, fermo restando il mio futuro trattamento. Do il permesso alla **registrazione video** durante lo svolgimento dei test. Do il permesso di scattare foto/girare video di me che uso il sistema, durante la partecipazione ai test. La mia firma qui sotto può essere considerata un'affermazione di tutte le dichiarazioni; mi è stato dato prima della mia partecipazione a questo studio.

Firma: _____

C.2.5 Questionario iniziale

Nome e Cognome: _____

1. Sesso:
   - ○ Maschio
   - ○ Femmina

2. Età:
   - ○ 16-25
   - ○ 26-35
   - ○ 36-45
   - ○ 55+

3. Nazionalità di Provenienza: _____

4. Regione di Provenienza: _____

5. Specialità professionale: _____

6. Esperienza nell'uso di dispotivi Android:
   - ○ Nessuna
   - ○ Poca
   - ○ Sufficiente
   - ○ Buona
   - ○ Eccellente

7. Esperienza nel settore della logistica:
   - ○ Meno di un anno
   - ○ 1-2 anni
   - ○ 2-5 anni
   - ○ 5-10 anni
   - ○ 10+ anni

8. Esperienza nel processo di picking:
   - ○ Meno di un anno
   - ○ 1-2 anni
   - ○ 2-5 anni
   - ○ 5-10 anni
   - ○ 10+ anni

9. Quanto conosci bene le esigenze dei pickeristi?
   - ○ Per niente
   - ○ Poco
   - ○ Abbastanza
   - ○ Molto

C.2.6 Questionario Finale

1. Ho trovato il sistema facile da usare.
   - ○ Fortemente disaccordo
   - ○ Disaccordo
   - ○ Sono d'accordo
   - ○ Fortemente d'accordo

2. Il dispositivo mi è stato di intralcio durante i prelievi.
   - ○ Fortemente disaccordo
   - ○ Disaccordo
   - ○ Sono d'accordo
   - ○ Fortemente d'accordo

3. L'interfaccia è intuitiva.
   - ○ Fortemente disaccordo
   - ○ Disaccordo
   - ○ Sono d'accordo
   - ○ Fortemente d'accordo

4. Ho trovato le varie funzioni del sistema poco integrate.
   - ○ Fortemente disaccordo
   - ○ Disaccordo
   - ○ Sono d'accordo
   - ○ Fortemente d'accordo

5. Avevo paura di sbagliare durante il prelievo.
   - ○ Fortemente disaccordo
   - ○ Disaccordo
   - ○ Sono d'accordo
   - ○ Fortemente d'accordo

6. Usando questa applicazione, i pickeristi riceverebbero un addestramento iniziale più rapido.
   - ○ Fortemente disaccordo
   - ○ Disaccordo
   - ○ Sono d'accordo

○ Fortemente d'accordo

7. Il sistema di prelievo non risponde alle esigenze dei pickeristi.
   ○ Fortemente disaccordo
   ○ Disaccordo
   ○ Sono d'accordo
   ○ Fortemente d'accordo

8. Sono favorevole all'introduzione in magazzino di soluzioni simili.
   ○ Fortemente disaccordo
   ○ Disaccordo
   ○ Sono d'accordo
   ○ Fortemente d'accordo

C.2.7 Domande debriefing

1. Quali sono le impressioni che ti vengono in mente alla luce del test svolto?
2. Durante il test, è stato un peso/ingombro il dover portarsi dietro il dispositivo?
3. Il sistema ti sembra utile? Secondo te aumentare la velocità di prelievo e può garantire più sicurezza grazie a mani libere?
4. Qual è il tuo giudizio sulla usabilità del sistema? Il sistema è facile e intuitivo da utilizzare oppure hai incontrato difficoltà?
5. Come giudichi l' interfaccia utente, sia grafica che vocale? Sono ben integrate?
6. Il sistema può migliorare le condizioni di lavoro di un pickerista?

C.2.8 Tasks

#	Testo del Task	Criteri di Successo	Metodologia
T1	Avvia la missione di prelievo vocale e metti il dispositivo in tasca. Ascolta quindi le istruzioni in cuffia e prova a regolare il volume/velocità del parlato. Controlla che sia ok.	Il partecipante pronuncia i comandi relativi correttamente e si regola in base al feedback del sistema.	None
T2	Ti si avvicina un collega che ti saluta ma non puoi parlare. Metti in pausa il sistema vocale e poi riprendi.	Il partecipante pronuncia "in pausa" e "pronto" correttamente.	Cooperative evaluation

T3	Hai dimenticato quanti colli hai da prendere e vuoi capire quanto grande deve essere l'udc. Fai in modo che il sistema ripeti le informazioni. Scegli quindi un udc.	Il partecipante pronuncia "ripeti" correttamente.	Cooperative evaluation
T4	Dichiara il codice a barre dell'udc al sistema. Qualora tu senta il bisogno di riprendere in mano lo smartphone, potrai farlo.	Il partecipante digita/pronuncia il codice correttamente.	None
T5	(Se ha pronunciato il codice) Hai pronunciato le ultime cifre del codice male. Cerca di rimediare e reinseriscile.	Il partecipante pronuncia "annulla" correttamente.	Cooperative evaluation
T6	Recati presso la posizione di prelievo. Se non ricordi dov'è usa l'aiuto del sistema per individuarla.	Il partecipante pronuncia "aiuto" correttamente.	Cooperative evaluation
T7	Dopo aver raggiunto la posizione, trova e pronuncia le cifre di controllo della posizione.	Il partecipante pronuncia le cifre di controllo correttamente.	None
T8	Ascolta il barcode dell'udc contenente l'articolo da prelevare, pronunciando "avanti" per passare al prossimo gruppo di simboli. Mentre ascolti individualo.	Il partecipante ha individuato l'udc correttamente.	None
T9	Ascolta il barcode dell'articolo da prelevare e, mentre ascolti, individualo. Come conferma, trova un modo per ascoltare la descrizione della merce.	Il partecipante ha individuato l'articolo correttamente e pronuncia "descrizione".	Cooperative evaluation
T10	Preleva l'articolo e dichiara una quantità sbagliata. Cerca di rimediare, e infine dichiara quindi la giusta quantità.	Il partecipante ha rimediato correttamente.	Cooperative evaluation
T11	Continua a prelevare i prossimi articoli e porta a termine la missione raggiungendo la destinazione.	Il partecipante ha terminato correttamente.	Cooperative evaluation

Tabella C.1: Lista di task da eseguire durante i test

C.2.9 Metriche

Metriche qualititative

- Commenti e domande dei partecipanti
- Linguaggio del corpo e espressioni facciali dei partecipanti

Metriche quantitative

- Tempo di completamento task

- Media di completamento (Numero di task completate/ Numero totale di task)
- Media di completamento senza errore
- Questionario finale

C.2.10 Script

Grazie <nome partecipante> per aver accettato di prendere parte a questo test. Io sono <nome facilitatore> e sono qui con te oggi per testare questa nuova soluzione di prelievo. Tutto bene finora? <discussione per mettere a proprio agio il partecipante, dandogli un attimo per parlare>.

Lo scopo di oggi è di provare le funzionalità e l'usabilità di un sistema di supporto per il prelievo vocale in magazzino, verificarne le potenzialità e capirne i problemi. Vi spiego ora come si svolgerà il test.

Proveremo il sistema simulando un intero giro di prelievo, dall'inizio della missione fino alla sua fine, quando la destinazione sarà raggiunta dall'udc con gli articoli prelevati. Io sarò il facilitatore del test e ti spiegherò passo dopo passo gli obiettivi da raggiungere, insieme a alcune istruzioni utili per usare il sistema. Fai attenzione che in alcuni casi dovrai provare a cavartela da solo, senza le mie spiegazioni. In questo modo potremo valutare la facilità (o difficoltà) di utilizzo del sistema. Non avere paura di sbagliare, qualsiasi errore è colpa del sistema e non tua!

Le spiegazioni che ti darò sono scritte in modo che io possa dare le stesse informazioni a tutti i partecipanti anche in test futuri. Se non avete domande, possiamo iniziare con il leggere questo documento [Informativa legale] e, se sei d'accordo, firmarlo, per permetterci di registrare la sessione. Con questo ci darai il diritto di fare foto e video durante i test. Ora possiamo procedere con la compilazione del questionario iniziale. [Compilazione questionario iniziale]

Ecco, possiamo iniziare il test. Prima però, è importante sapere alcune cose: ci sono dei comandi vocali che è sempre possibile dire:

- **in pausa** per sospendere il riconoscimento vocale, e **pronto** per riattivarlo;
- **volume sù/giù** per controllare il volume;
- **accellera/rallenta** per controllare la velocità del parlato;
- **ripeti** per far ripetere al sistema l'ultima cosa detta;
- **aiuto** per avere indicazioni maggiori;
- **vocabolario** per risentire tutti i comandi vocali che è possibile dire sempre;

Ora, se sei pronto iniziamo! [Inizio riprese]

T1 - Apri l'applicazione sul dispositivo e inserisci le credenziali che ti sono state fornite. Dopodichè scegli una missione di prelievo vocale tra quelle disponibili e metti il dispositivo in tasca. Ascolta quindi le istruzioni in cuffia e prova a regolare il volume/velocità del parlato. Controlla che sia ok.

T2 - Ti si avvicina un collega che ti saluta ma non puoi parlare. Metti in pausa il sistema vocale e quando sei pronto riprendi.

T3 - Hai dimenticato quanti colli hai da prendere e vuoi capire quanto grande deve essere l'udc. Fai in modo che il sistema ripeta le informazioni per aiutarti a decidere. Scegli quindi un udc.

T4 - Dopo averlo preso, troverai il codice a barre sull'etichetta dell'udc. Dichiara il codice a barre al sistema. Qualora tu senta il bisogno di riprendere in mano lo smartphone, potrai farlo.

T5 - (Solo se il partecipante ha pronunciato il codice) Hai pronunciato le ultime cifre del codice male. Cerca di rimediare e, poi, reinseriscile.

T6 - Dovrai ora recati presso la posizione di prelievo in magazzino. Se non ricordi dov'è, usa l'aiuto del sistema per individuarla. Porta con te l'udc, qui inserirai gli oggetti prelevati.

T7 - Dopo aver raggiunto la posizione specificata dal sistema, trova e pronuncia le relative cifre di controllo della posizione, per controllare che sia quella giusta.

T8 - Devi ora trovare l'udc contenente l'articolo da prelevare. Ascolta il barcode cercato, pronunciando "avanti" per passare al prossimo gruppo di simboli. Mentre ascolti in cuffia il codice, individualo.

T9 - Tra gli oggetti nell'udc, ne dovrai prelevare uno con una certa quantità. Ascolta quindi il barcode dell'articolo da prelevare e, mentre ascolti, individualo.

T10 - Preleva l'articolo e dichiara una quantità errata, ossia diversa da quella richiesta. Cerca di rimediare, e infine dichiara quindi la quantità giusta.

T11 - Continua a prelevare i prossimi articoli e porta a termine la missione raggiungendo la destinazione.

[Fine riprese]

Grazie mille di aver permesso questo test approfondito, il vostro aiuto sarà preziosissimo per migliorare il sistema. Ti chiediamo, gentilmente, ancora alcuni minuti di tempo per il questionario finale. [Compilazione questionario finale e poi domande debriefing]

Bibliografia

[1] *ISO 13407: Human Centered Design*. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=21197 (cit. alle pp. 3, 47).

[2] Zetes Group. *Picking e packing: differenze tra i due sottoprocessi*. Mecalux. Nov. 2019. URL: https://www.mecalux.it/blog/picking-packing-differenze-prelievo-imballaggio#:~:text=I%5C%20termini%5C%20picking%5C%20e%5C%20packing,un%5C%20primo%5C%20spacchettamento%5C%20degli%5C%20articoli. (cit. a p. 8).

[3] Abby Jenkins. *What Is Voice Picking? How It Works, Benefits & FAQs*. ORACLE Netsuite. Apr. 2015. URL: https://www.netsuite.com/portal/resource/articles/inventory-management/voice-picking.shtml#:~:text=Voice%5C%20picking%5C%20solutions%5C%20are%5C%20paperless,pick%5C%20to%5C%20complete%5C%20customer%5C%20orders. (cit. alle pp. 9, 11).

[4] *WMS (Warehouse Management System)*. Logistica Efficiente. URL: https://www.logisticaefficiente.it/wiki-logistica/magazzino/wms-warehouse-management-system.html (cit. a p. 9).

[5] *Voice Recognition Expands Beyond Order Picking*. Food Logistics. Gen. 2015. URL: https://www.foodlogistics.com/software-technology/article/11598300/voice-recognition-expands-beyond-order-picking (cit. a p. 10).

[6] *What hardware do I need for a WMS deployment?* Corax. Ott. 2013. URL: https://www.coraxsaaswms.com/en/what-hardware-do-i-need-for-a-wms-deployment/ (cit. a p. 11).

[7] Ben Lutkevich e Karolina Kiwak. *What is speech recognition?* TechTarget. URL: https://www.techtarget.com/searchcustomerexperience/definition/speech-recognition (cit. a p. 13).

[8] Kan Li e Jose C Principe. «Biologically-Inspired Spike-Based Automatic Speech Recognition of Isolated Digits Over a Reproducing Kernel Hilbert Space». In: *The Effect of Carpal Tunnel Changes on Smartphone Users* (2012) (cit. a p. 14).

[9] *Speech synthesis*. Wikipedia. URL: `https://en.wikipedia.org/wiki/Speech_synthesis` (cit. a p. 14).

[10] Gaia Mellone. «In Cina ha debuttato il primo giornalista AI». In: *Giornalettismo* (2018). URL: `https://www.giornalettismo.com/cina-debuttato-primo-giornalista-ai/` (cit. a p. 16).

[11] *Useful, Usable, and Used: Why They Matter to Designers*. Interaction Design Foundation. 2021. URL: `https://www.interaction-design.org/literature/article/useful-usable-and-used-why-they-matter-to-designers#:~:text=Usable%5C%20refers%5C%20to%5C%20the%5C%20usability,as%5C%20possible)%5C%20and%5C%20effective%5C%20manner.` (cit. a p. 17).

[12] Yen-ning Chang, Youn-kyung Lim e Erik Stolterman. «Personas: From Theory to Practices». In: New York, NY, USA: Association for Computing Machinery, 2008. URL: `https://doi.org/10.1145/1463160.1463214` (cit. a p. 18).

[13] *M300XL smart glasses*. Vuzix. URL: `https://www.vuzix.eu/products/m300xl-smart-glasses-1` (cit. a p. 19).

[14] Steve Krug. *Don't Make Me Think: A Common Sense Approach To The Web Usability: A Common Sense Approach to Web Usability*. A cura di New Riders Pub. Second. New Riders Publishing 201 West 103rd Street Indianapolis, IN 46290 USA, 2005 (cit. a p. 25).

[15] *Kaldi*. Alpha Cephei. URL: `https://github.com/kaldi-asr/kaldi` (cit. a p. 33).

[16] *Kaldi Android Demo*. Alpha Cephei. URL: `https://github.com/alphacep/vosk-android-demo` (cit. a p. 34).

[17] *Android Native TextToSpeech*. Android. URL: `https://developer.android.com/reference/android/speech/tts/TextToSpeech` (cit. a p. 35).