POLITECNICO DI TORINO

Corso di laurea in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

Progetto di realizzazione di un serious game per l'insegnamento delle tecniche di uso consapevole delle fonti energetiche

Relatore

Prof. Giovanni MALNATI

Candidato

Andrea GARINI

Aprile 2023

Sommario

Edilclima è un'azienda di Borgomanero (NO) che si occupa della creazione di software per la valutazione della dispersione energetica nelle abitazioni; tuttavia, riconoscendo l'importanza della prevenzione nell' ambito, da anni si propone di effettuare intenti informativi nelle scuole medie inferiori e superiori del territorio. Dal desiderio di sfruttare le caratteristiche peculiari dei serious game nel corso di queste lezioni, l'azienda si è in passato cimentata nella creazione di un gioco da tavolo. L'esperienza del Covid ha, tuttavia, fatto sorgere la necessità di creare un nuovo gioco educativo sotto forma di mobile app. Questo progetto di tesi verteva quindi sullo sviluppo della suddetta applicazione, partendo dalla fase di design del gioco stesso per passare poi alla realizzazione vera e propria. Il processo di definizione delle dinamiche di gioco è stato influenzato dalle necessità peculiari di questo progetto; infatti, data la modalità gruppale in cui l'app sarebbe stata utilizzata, e tenuto conto del contesto scolastico in cui si sarebbe fruito del prodotto, si rendeva evidente la necessità di sviluppare un serious game dalle meccaniche semplici e di immediata comprensione, che stimolasse l'interazione fra gli studenti, e che riuscisse in un tempo limitato a restituire una panoramica della realtà operativa di un termotecnico. Per soddisfare il primo requisito si è deciso di sviluppare un gioco che facesse riferimento ad un mondo ludico noto, in modo tale da sfruttare la dimestichezza pregressa degli utenti per facilitare la comprensione del funzionamento del serious game. Per questo motivo si è deciso di sviluppare un card game, il cui regolamento può essere così definito: All'inizio di ogni partito i giocatori connessi vengono suddivisi in squadre, fino ad un numero massimo di quattro. La partita è suddivisa in differenti livelli, ognuno dei quali mira a proporre una tipologia di edificio sul quale effettuare un'opera di riqualificazione energetica. Le squadre dovranno progettare un piano di lavori a effettuare nel corso di un anno. Per poter decidere quali interventi svolgere in ognuno dei dodici mesi, i giocatori dovranno giocare le carte in loro possesso su di un tabellone rappresentante i vari mesi. Soltanto un giocatore alla volta per ogni squadra può effettuare una mossa. Ogni giocatore può effettuare una sola azione durante il suo turno, scelta fra:

- Giocare una carta in suo possesso in una posizione libera sul tabellone della sua squadra;

- Prendere una carta dal tabellone liberando la posizione corrispondente

Al termine di ogni livello si calcolano i punteggi ottenuti dalle squadre, che dipenderanno dall'adeguatezza degli interventi progettati.

Definito il design del serious game si è passati alla definizione dei mezzi tecnici da utilizzare per la sua realizzazione. Oggigiorno, esistono numerosi strumenti pensati per lo sviluppo di applicazioni mobile, tuttavia, questi possono essere suddivisi in due macro categorie, che a loro danno origine ad altrettante tipologie di applicazioni: native e platform-independent. Le prime vengono sviluppate specificatamente per un unico sistema operativo, il che garantisce che le performance ottenibili dal software siano massimizzate. Le app platform-indipendent invece nascono per poter essere utilizzate attraverso device con sistemi operativi differenti. Queste ultime garantiscono l'interoperabilità del prodotto, sacrificando per questo performance ed ottimizzazione. Dovendo sviluppare un'applicazione che sarebbe stata utilizzata dagli studenti sugli smartphone di loro proprietà, era evidente la necessità di ricorrere all'utilizzo di strumenti per lo sviluppo indipendente dal sistema operativo. Il mercato dei framework platform-independent oggi può vantare numerosi prodotti largamente utilizzati come, ad esempio, React Native, Xamarin e Ionic. Per questo progetto di tesi si è scelto di utilizzare Flutter, framework prodotto da Google nel 2018 in grado di generare app per Android, iOS, web, Windows e MacOS. La scelta è ricaduta su questo prodotto poiché, rilasciato nel Novembre 2018, nella sua breve vita è riuscito ad attrarre gli sviluppatori, al punto da divenire il framework platform-indipendent più utilizzato nel 2021, superando il competitor React Native. Il serious game oggetto di questa tesi si configura come un prodotto multigiocatore, il che implica che, durante il suo utilizzo, lo stato dell'applicazioni muti sulla base delle azioni degli altri giocatori. Questa necessità ha portato all'integrazione nell'app della piattaforma Google Firebase, ossia un aggregatore di servizi cloud backend che possono essere implementati nelle applicazioni, in modo tale che gli sviluppatori possano concentrarsi sullo sviluppo front-end. In particolare, per il progetto di tesi si è deciso di utilizzare Firebase RealTime Database, che permette di avere accesso ad un database NoSQL costituito da un unico documento JSON. Il database è stato organizzato per contenere le informazioni relative alle partite, dividendo i dati in tre rami principali, ossia le informazioni relative al livello corrente, quelle riguardanti i singoli giocatori e quelle concernenti le squadre in gioco. I giocatori erano identificati attraverso un meccanismo di autenticazione anonima messo a disposizione da Firebase, il quale si occupa di andare a generare un UID univoco per ogni device collegato. L'applicazione è stata sviluppata seguendo i dettami del pattern architetturale MVVM, ossia un'architettura software front-end che vuole il progetto diviso in tre sezioni logiche: Model, View e ViewModel. La rappresenta la parte dell'applicazione che si occupa di gestire i dati, nonché implementare la logica necessaria per elaborarli, ed infine salvarli e conservarli. Il Model deve essere indipendente da ViewModel e View, in modo tale che possa garantire sempre

la consistenza dello stato del software. Il ViewModel fornisce alla View i dati necessari per visualizzare l'interfaccia utente e gestisce la logica dell'applicazione in risposta alle azioni dell'utente sulla View. Inoltre, il ViewModel fornisce una rappresentazione delle informazioni contenute nel Model in un formato che può essere facilmente visualizzato e gestito dalla View. L'ultima sezione, che prende il nome di View, rappresenta la parte dell'applicazione che si occupa di andare a costituire l'interfaccia utente. In altre parole, la View definisce la struttura, il layout e il comportamento della UI. Questo modello è stato implementato definendo la classe GameLogic, la quale implementava una serie di data class che andavano a rappresentare tutti gli asset di gioco, inoltre all'interno dei suoi metodi trova spazio tutta la logica di gestione del gioco. Al contempo la GameLogic si occupa di ricevere i dati estratti dal database dal ViewModel al fine elaborarli, costruendo nuove ed aggiornate strutture dati da ritornare al ViewModel, in modo che quest'ultimo si occupi unicamente di scriverle sul database. Il ViewModel è costituito dalla classe GameModel, che si occupa di trasferire i dati della GameLogic alle varie View. Per ottenere questo, si è sfruttato il package "provider" di Flutter, il quale mette a disposizione degli sviluppatori i componenti "ChangeNotifier" e "Consumer". Il primo permette di informare una serie di listeners dell'aggiornamento del suo stato, mentre il secondo permette di ai componenti della UI di essere informato del cambiamento dello stato dell'app, dando, inoltre, la possibilità di implementare la logica reattiva a questo evento, per forzare poi l'aggiornamento automatico dell'interfaccia. Il GameModel racchiudeva in sé anche tutte le chiamate a Firebase RealTimeDatabase, per cui si conformava come il centro di smistamento dati unico fra database, interfaccia grafica e Model. La UI dell'app è costituita da differenti schermate, visitabili attraverso una sistema di navigazione interna definito grazie all'utilizzo del package "go-router", che permette di ottenere due differenti pattern di navigazione: una semplice navigazione che sostituisce una schermata alla successiva, ed un altro archetipo di navigazione che inserisce le pagine all'interno di un guscio grafico, consentendo di mantenere alcuni componenti dell'interfaccia grafica fissi ed altri modificabili. L'applicazione presenta un percorso costituito da schermate successive, che ha lo scopo di permettere l'accesso del giocatore alla partita. Una volta iniziata la partita gli utenti vengono portati all'interfaccia di gioco, costituita da una statusBar superiore, una barra di navigazione inferiore, ed una sezione centrale che contiene le pagine navigabili. Questo è stato possibile grazie alla definizione di un componente MainScreenContent, che si occupa di andare a distribuire lo spazio a schermo fra questi componenti, ma anche di generare componenti grafici in overlay oppure di aprire finestra di dialog all'occorrenza. La sezione di gioco prevede tre pagine interne, una per far sì che i giocatori possano giocare le carte in loro possesso in una posizione sul tabellone di gioco, una per effettuare l'operazione opposta, ossia prelevare le carte dal tabellone, ed un'ultima pagina per visualizzare la classifica delle squadre in tempo reale. Infine,

per permettere l'accesso alla partita è necessario che un utente si ponga come gameMaster, generando un QR code che, quando inquadrato dagli altri utenti, permette di registrare il device nella giusta locazione nel database. il gameMaster, dopo aver fatto iniziare la partita, verrà reindirizzato in una sezione dell'app a lui preposta. Questa pagina speciale è stata progettata per essere proiettata sulle lavagne interattive delle classi, e rappresenta il "tabellone di gioco". Al suo interno trovano spazio delle Card contenenti uno Stack di png. Quest'ultimo si modificherà sulla base delle azioni effettuate dagli utenti, in modo tale da rappresentare visivamente lo stato attuale dell'edificio su cui i giocatori stanno intervenendo. Lo sviluppo dell'applicazione è stato quindi concluso, ora dovrebbe seguire la fase di testing ed analisi degli worst case, per poi passare all'ottimizzazione delle performance e del consumo energetico.

Indice

Elenco delle figure

Capitolo 1

Serious game

L'apprendimento è il processo umano attraverso il quale noi comprendiamo e conosciamo la realtà che ci circonda, al fine di sviluppare la capacità di carpirne informazioni utili, oppure essere in grado di padroneggiare tecniche che reputiamo vantaggiose per noi stessi. Tradizionalmente, il processo di apprendimento viene fatto coincidere con l'attività dello studio, intesa come il processo di lettura e memorizzazione di concetti, ma anche la successiva applicazione di quanto compreso. Tuttavia, esiste un processo di apprendimento che potremmo definire "per esperienza", poiché non con lo studio si viene a conoscenza del sapere di interesse, bensì attraverso un'interazione diretta con la realtà che ci circonda. In questa seconda modalità, le nozioni vengono interiorizzate attraverso la ripetizione di azioni o l'utilizzo di tecniche ed analizzando i risultati di volta in volta ottenuti. La nostra memoria si occuperà dunque di fissare una data sequenza di azioni ed il risultato ottenibile attraverso la loro esecuzione, in modo tale che la conoscenza possa considerarsi definitivamente appresa e dunque fruibile quando necessario.

1.1 Apprendimento per esperienza

Questa seconda modalità di apprendimento è quella prediletta dai bambini, i quali si cimentano in una costante scoperta ella realtà e delle leggi che la regolano interagendo con gli oggetti che la compongono. Tipicamente i bambini possono sfruttare due processi per comprendere il mondo che li circonda, benché questi portino a risultati esperienziali molto differenti (Deplano V. 2010)[1]:

1. • La prima modalità è quella che comprende l'interazione diretta con gli oggetti al fine di addomesticarli al proprio volere. Il bambino in questo caso figura nella sua mente un risultato sperato, e successivamente cerca di ottenerlo manipolando per tentativi la materia che lo circonda. Questa modalità per tentativi ed errori sottopone il bambino allo stress del fallimento, finché, infine,

riuscirà a comprendere la corretta sequenza di azioni tramite le quali superare con successo la sfida autoimposta. L'esempio citato nell'articolo per descrivere questa modalità è quello del bambino che tenta di impilare dei mattoncini giocattolo al fine di costruirne un'alta torre ed evitarne il crollo;

2. • La seconda modalità è quella che prevede che il bambino estragga alcuni elementi selezionati dalla realtà, per poi portarli nel regno ella propria fantasia in modo da potervi interagire come preferisce. In questa seconda soluzione il soggetto crea delle astrazioni interiori degli oggetti reali, depurandoli di tutte quelle caratteristiche che non si confanno al successo della creazione della sua realtà immaginaria. Questo mondo fantastico risponde in tutto e per tutto alle aspettative del suo creatore, e si modifica secondo i suoi desideri, per cui, benché anche quello per mezzo della fantasia sia un processo di conoscenza (nello specifico per astrazione) esso non presenta alcun insuccesso possibile, ma anzi esiste solo la sensazione appagante di un meccanismo perfettamente funzionante. L'esempio citato da Deplano per descrivere questa modalità ci racconta di un bambino intento a trasportare della sabbia con un camion giocattolo.

In entrambi i processi descritti il soggetto è intento a giocare, ma nel farlo sta contemporaneamente apprendendo concetti relativi al funzionamento della realtà circostante. Tuttavia, Deplano ci racconta di come la prima modalità sia da considerarsi più formativa, poiché l'apprendimento passa attraverso lo scontro con le leggi fisiche, impossibili da piegare al proprio volere, per cui l'ottenimento del risultato sperato può avvenire solo se il soggetto accetta di comprendere le suddette leggi e di integrarle nella sua ricerca della soluzione dell'enigma. Il processo che conduce alla conoscenza in questo caso passa attraverso tre fasi :

1. Il soggetto tenta di piegare la realtà ai propri modelli, come il bambino che impila disordinatamente i mattoncini convinto possano stare l'uno sull'altro, incurante delle leggi fisiche;

2. In seguito alla scoperta della non corrispondenza fra le proprie previsioni e la realtà, il soggetto sviluppa una vera e propria crisi cognitiva, e con essa la necessità di risolverla;

3. Infine, i modelli mentali vengono modificati, arricchiti dall'analisi degli esiti fin qui ottenuti, in modo da avvicinarsi all'esecuzione corretta del processo.

Questi processi d'apprendimento, tipici dell'età infantile, vengono poi sostituiti in età scolare da quelli basati sulla lettura e sull'utilizzo di varie tecniche mnemoniche. Resta tuttavia evidente il potenziale formativo derivante dalle esperienze dirette, in particolare in quegli ambiti in cui risulta difficile acquisire conoscenza basandosi

unicamente sullo studio, come ad esempio quello relazionale o interpersonale, nel quale è la pratica ad insegnarci come agire correttamente. Entra dunque in gioco il concetto di "learning by doing", coniato dal filosofo e pedagogista John Dewey intorno alla metà del secolo scorso. Questa branca pedagogica crede fermamente nel potere educativo delle esperienze dirette, e punta dunque sul coinvolgimento attivo dello studente, rendendolo protagonista dei processi di apprendimento. Il soggetto quindi non è più da intendersi come fruitore di una sapienza statica che gli viene impartita, costringendolo ad un'educazione unicamente passiva, ma anzi egli viene posto davanti ad un dilemma da risolvere, in modo da costringerlo a formulare ipotesi che dovranno poi essere tradotte in azioni concrete, delle quali poter misurare l'esito; e proprio dal successo o meno del percorso delineato lo studente sarà in grado di correggere le propria ipotesi, per poter tentare nuovamente avendo presumibilmente alzato le proprie possibilità di buona riuscita. Il principio del "learning by doing" si basa su tre pilastri fondamentali (Ord, 2012)[2]:

1. L'apprendimento è più rapido e duraturo se il soggetto è coinvolto attivamente nell' esperienza educativa;

2. La conoscenza non è intesa come una verità imposta alla quale aderire, ma è invece scoperta direttamente dal soggetto attraverso un percorso per tentativi;

3. L'impegno profuso dal soggetto nell' attività sarà direttamente proporzionale alla libertà della quale lo stesso può godere nel prendere le decisioni che ritiene più adeguate per raggiungere il risultato sperato.

Al di sotto del grande cappello del "learning by doing" possiamo inserire numerose esperienze formative di differente natura; infatti qui potremmo citare forme di apprendimento sul posto di lavoro, come laboratori, workshop, stage e l'alternanza scuola-lavoro presente nei nostri istituti superiori, ma anche simulazioni di vario genere dell'ambiente lavorativo o professionale.

1.2 Simulazioni

A questo punto è bene descrivere quali siano le caratteristiche di una simulazione, poiché alcune di queste saranno mutuate dai serious game. Una simulazione è il risultato di un processo di mimesi della realtà attraverso uno specifico medium (ad es. quello digitale), benché sia chiaro che il reale è eccessivamente complesso perché possa essere ricreato per intero. Una simulazione, dunque, tenta di riprodurre alcuni elementi del mondo che ci circonda, ma nel farlo opera una semplificazione dello stesso, depurandolo di tutte quelle informazioni non utili alla comprensione dell'oggetto della simulazione stessa. Questo processo di selezione degli elementi del reale da riproporre deve comunque garantire che il prodotto finale mantenga

una validità in termini di coerenza con il reale come lo conosciamo. Inoltre, le simulazioni presentano l'enorme potenziale incarnato dalla possibilità di entrare in contatto con eventi, processi o esecuzioni tecniche che per limiti di tempo, spazio oppure costo difficilmente si potrebbero ottenere nel mondo reale. In questo modo le simulazioni possono preparare i professionisti ad operare in contesti rischiosi, come ad esempio la guida di un aereo per i piloti in formazione, oppure l'intervento in caso di incendio per i vigili del fuoco. La natura semplificata che dà vita alla simulazione permette di accorciare i tempi intercorsi fra causa ed effetto, in modo tale che sia possibile osservare l'evoluzione di un modello senza dover attendere i tempi che normalmente questo processo richiederebbe. Allo stesso modo si può operare dilatando i tempi per poter descrivere al meglio i passaggi che definiscono il modificarsi di un processo che nella realtà avverrebbe con eccessiva rapidità, impendendo così di poterne cogliere i dettagli. Infine, il maggiore punto a favore dell'utilizzo delle simulazioni è la possibilità dell'errore a costo zero: infatti le simulazioni permettono ai loro fruitori di prendere decisioni sbagliate senza dover fare i conti con le conseguenze che queste scelte avrebbero nel mondo reale. Quest'ultima peculiarità favorisce quell'apprendimento per tentativi già descritto precedente, poiché il diritto di sbagliare senza che questo implichi un immediato impatto negativo, unitamente al fatto che la simulazione può essere ripetuta più volte in un tempo ristretto, si traduce in un rapido apprendimento per esperienza. Le modalità con cui si può realizzare una simulazione sono molteplici, poiché esse possono essere puramente astratte, basate su giochi da tavolo, digitali oppure supportate da strumenti specifici, come monitor avvolgenti uniti a sistemi meccanici per riprodurre le sollecitazioni tipiche dell'esperienza che si vuole andare a riprodurre (caso tipico di quest'ultima tipologia sono i simulatori di guida oppure di volo). L'evoluzione della simulazione può essere garantita tramite l'utilizzo di un tipico albero decisionale a bivi in cui le decisioni dell' utente definiscono un percorso di progressione dell'ambiente, ma al contempo è possibile utilizzare sistemi dinamici costituiti da un grande numero di variabili che devono essere controllate dall'utente. Esistono, infine, simulazioni puramente probabilistiche in cui una serie di elementi atomici con caratteristiche peculiari ed un comportamento definito da variabili statistiche guidano con il loro agire l'evoluzione del mondo virtuale.
Definite caratteristiche e potenziale formativo delle simulazioni, dobbiamo ora introdurre un ulteriore definizione al fine di sopraggiungere alla più ampia descrizione dei "serious game", ossia quella di gioco.

1.3 Gioco

Una possibile definizione di "gioco" potrebbe essere: un'attività che coinvolge uno o più partecipanti in una situazione di svago e divertimento, che può essere

caratterizzata da regole e obiettivi definiti, e che può essere svolta per motivi educativi, sociali o competitivi. In un gioco, i partecipanti spesso cercano di raggiungere un certo risultato o di superare gli altri giocatori, ma l'elemento principale è il piacere che deriva dal giocare stesso. Il gioco può assumere molte forme diverse, come i giochi da tavolo, i giochi di carte, i videogiochi, i giochi all'aperto e così via. Un gioco è dunque un'attività ludica che io decido di svolgere poiché intrinsecamente motivato, infatti io gioco per trarre piacere dall' atto stesso di giocare, e non per ottenerne necessariamente un vantaggio. In italiano, noi abbiamo un 'unica parola per descrivere due concetti differenti che possono però essere meglio espressi rifacendosi alla lingua inglese, dove troviamo infatti il fun, attività svolta per se stessa senza la necessità di ottenerne nulla, ed il play, ossia un tipo di gioco sottoposto ad una serie di regole che definiscono le modalità d'interazione fra giocatori oppure fra giocatore ed ambiente di gioco.

Volendo dare una definizione più completa, potremmo rifarci a quanto affermato da Roger Caillois, sociologo e critico letterario francese, che ha scritto molto sulla natura del gioco. Nel suo libro del 1958 "Les jeux et les hommes" (tradotto in inglese come "Man, Play and Games")[3], Caillois ha offerto una definizione di gioco che si basa su quattro categorie principali:

1. Agon: la competizione, intesa come il desiderio di confrontarsi con gli altri e di vincere.

2. Alea: il caso, inteso come l'incertezza e la casualità, che aggiunge suspense e imprevedibilità al gioco.

3. Mimicry: la simulazione, intesa come l'imitazione di un'attività, un personaggio o una situazione, che consente ai giocatori di sperimentare la vita attraverso una rappresentazione.

4. Ilinx: la vertigine, intesa come la sensazione di smarrimento o di perdita di controllo, che si sperimenta in alcuni giochi di movimento o di equilibrio.

Secondo Caillois, questi quattro elementi sono presenti in varie proporzioni in tutti i giochi, e possono essere combinati in modi diversi per creare esperienze di gioco diverse. La sua definizione di gioco si basa quindi su un'analisi sociologica delle funzioni e dei significati del gioco nella cultura umana. All'interno dell'ambito dei serious game troviamo principalmente le componenti agon e mimicry, infatti generalmente questi pongono i giocatori in competizione fra di loro oppure contro il gioco stesso (nel caso di giochi digitali potremmo dire che lo sfidante è il computer), benché siano presenti serious game in cui è prevista anche la cooperazione fra i partecipanti; al contempo queste esperienze ludico-formative, tipicamente, fanno riferimento ad una porzione di realtà che è quella che si vuole simulare al fine di farne comprendere le varie sfaccettature.

1.4 Elementi atomici di un gioco

Passiamo ora ad elencare quelli che potremmo definire gli elementi atomici di un gioco (qui inteso come play) ossia:

- Giocatori: i giocatori possono essere uno o molteplici, ed in questo secondo caso essi sono in relazione attraverso alcune regole di interazione, in modo che i partecipanti entrino a far parte di un mondo definito da leggi proprie che il giocatore implicitamente accetta ed alle quali si attiene volontariamente;

- Obbiettivo: l'obbiettivo definisce lo scopo ultimo del gioco, ma contemporaneamente già ne fa intravedere il genere oppure la natura. Se, ad esempio, scopo del gioco è battere i propri avversari, il gioco sarà necessariamente di tipo competitivo;

- Procedure: le singole azioni che è lecito compiere all'interno del mondo del gioco. Ogni procedura permette di ottenere un determinato risultato sulla base delle regole del gioco;

- Regole: le regole sono quei limiti invalicabili che vengono applicati alle procedure, al fine di dare a queste ultime un significato preciso e non equivoco. Ponendo, ad esempio, di star giocando a dama, una procedura si realizza nella possibilità di muovere una pedina, ma la regola impone che se ne possa muovere soltanto una per turno.

- Risorse: le risorse sono oggetti fruibili nel corso della partita al fine di ottenere i propri obbiettivi. Il valore di queste ultime è definito dalla loro scarsità o unicità. In questo caso, esemplificativo è il denaro che si possiede nel corso di una partita a Monopoly;

- Conflitto: il conflitto rappresenta il livello di sfida fra due giocatori o, nel caso ei videogiochi, fra giocatore e computer. Esso è generato dalle relazioni esistenti tra giocatori, regole e procedure. Si possono identificare tre tipologie fondamentali di conflitto:

    1. Gli ostacoli: gli ostacoli sono dispositivi creati dal gioco al fine di bloccare il progresso del giocatore. Essi possono essere intesi come strumenti per guidare la partita verso un'evoluzione definita, oppure come mezzi per rallentare o complicare il raggiungimento del successo da parte dei partecipanti;
    2. Gli avversari: gli avversari, se presenti, sono coloro che, con le loro azioni, cercano di impedire al giocatore di raggiungere il proprio scopo;

3. Il Dilemma: si identifica come dilemma una qualsiasi situazione in cui il giocatore è portato a compiere una scelta sulla base delle informazioni in suo possesso, oppure delle risorse di cui dispone.

- Limiti: i limiti sono un insieme di concetti che nella loro totalità vanno a definire i confini del cosiddetto "cerchio magico", ossia il sistema di assunzioni che i giocatori devono fare per poter entrare nel mondo di gioco. Il superamento di un limite tipicamente trasforma un gioco in un altro, rompendone la natura originaria. Se, ad esempio, si sta giocando a "guardie e ladri" non è possibile inserire una terza figura senza snaturare il gioco originario;
- Risultato: il risultato finale del mio percorso. Quest'ultimo deve essere sempre presente, ma al contempo ha la necessità di essere verosimile, poiché tutte le parti in gioco devono essere convinte che l'esito dipenda dalle azioni che i singoli hanno compiuto nel corso della partita, e non da un'intrinseca natura del gioco che tende a favorire un soggetto piuttosto che un altro.

La sinergia fra questi singoli elementi atomici determina la costruzione del fenomeno che prende il nome di "gioco", ma è il corretto bilanciamento di queste stesse componenti a determinare quella che potremmo definire la "qualità" del gioco, qui intesa, in ambito di play, unicamente come la capacità di generare divertimento nei partecipanti.

1.5 Videogiochi ed edutainment

I giochi possono presentarsi sotto forme estremamente differenti, a partire da quella puramente fantastica tipica dell'età infantile, a tutte le forme di gioco delle quali si continua ad usufruire anche in età adulta. In particolare, il gioco si presta ad essere veicolato attraverso media differenti, e fra questi possiamo ritrovare proprio le simulazioni già analizzate precedentemente. L'unione del mondo ludico con quello delle simulazioni, ed in particolare con il sottoinsieme delle simulazioni digitali, ha portato alla nascita dei cosiddetti videogiochi. Benché la fusione del mondo digitale con l'universo ludico fosse oggetto di studio già a partire dagli anni '60, è solo con l'avvento dei primi anni '80 che il fenomeno entra a far parte della cultura pop, in particolare con l'avvento delle sale giochi nelle quali era possibile intrattenersi con i cosiddetti "cabinati", ossia una struttura esterna di supporto completa di dispositivi di input e schermo, attraverso la quale era possibile fruire di un singolo gioco. L'evoluzione tecnologica, il crescente interesse nei confronti del fenomeno da parte di un pubblico giovane ed i consistenti investimenti nel settore portano, entro gli inizi degli anni novanta, allo sviluppo di videogiochi di successo internazionale, nonché alla nascita delle prime console dedicate in grado di ottenere ottimi risultati di vendita (e.g. Playstation di Sony ed Xbox di Microsoft). In questo

periodo, i generi videoludici di maggiore successo sono rappresentati dai cosiddetti "sparatutto", dai giochi di strategia in tempo reale ed infine dai primi giochi di ruolo online multi-giocatore di massa. L'articolazione della struttura del gioco, l'impatto degli aspetti percettivi, il ritmo e il coinvolgimento emotivo, il ricordo ad ambienti sempre più precisi e ricchi di immersione sono stati resi possibili dalla crescita esponenziale di tecnologie avanzate. L'industria dei videogiochi, abbandonate le sale giochi, ha puntato fortemente alla creazione di esperienze ludiche fruibili in ambito domestico, ottenendo così una penetrazione molto elevata specialmente nelle fasce d'età più giovani. Per coloro che sono nati dopo gli anni Ottanta, i videogiochi hanno rappresentato un'esperienza fondamentale ed abituale, tanto da far parlare di Game Generation. Oggi i videogiochi costituiscono un settore rilevante del mercato dell'intrattenimento, con un fatturato annuo pari a due volte quelli ottenuti dal mondo del cinema e della musica combinati. Il fenomeno videoludico è divenuto tanto abituale da portare con sé disturbi specifici legati ad un' eccessiva immersione all'interno di questo mondo; infatti noti e numerosi sono i casi, per la maggior parte riconducibili a soggetti in età adolescenziale, del fenomeno patologico della "dipendenza da videogioco", che consiste in una reiterazione ossessiva dell'attività ludica per un lungo periodo al giorno, in una ricerca spasmodica delle ultime novità nell'ambito, in un grado elevato di incontentabilità, nel dispendio eccessivo di risorse dedicato ai videogiochi, nei disturbi del sonno, nell'isolamento sociale, in una facile distraibilità ed incapacità di concentrazione dell'attenzione. La capacità dei videogiochi di generare in coloro che li utilizzano un forte trasporto fa sì che la fruizione di questi prodotti sia stato fin da subito identificato come un'esperienza ad alto livello di immersione. Come precedentemente esposto, queste sono ottime premesse per un più proficuo apprendimento per esperienza. La fascinazione di insegnare divertendo, anche attraverso l'utilizzo di prodotti digitali, ha riscosso grande interesse dapprima da parte del mondo accademico e, successivamente, sulla base dei risultati positivi emersi da studi condotti in ambito universitario, anche da parte del mondo divulgativo e mass-mediale. Il tentativo di ibridazione fra il mondo dell'intrattenimento e quello educativo ha portato alla coniazione del termine "edutainment". Un esempio noto di edutainment nel nostro Paese è la celebre serie tv animata "Esplorando il corpo umano", la quale si poneva come obbiettivo quello di generare prodotti educativi che grazie alla loro forma divertente e ad una narrazione giocosa permettono di intrattenere i bambini insegnando loro la biologia umana di base. Se l'unione dei concetti di intrattenimento e apprendimento ha portato alla nascita del fenomeno dell'edutainment, è solo dall'unione dei tre mondi finora descritti che nascono i serious game, ossia dalla sinergia creata dal mescolamento di gioco, simulazione e potenzialità dell'apprendimento per esperienza.

Figura 1.1: Intersezione fra apprendimento, simulazione e gioco

1.6 Storia e caratteristiche dei Serious game

I serious game, o giochi seri, sono un tipo di gioco che unisce l'intrattenimento all'apprendimento. Questi giochi sono stati sviluppati con l'obiettivo di educare e formare i giocatori, piuttosto che solo divertirli. In questo capitolo, esamineremo il concetto di serious game, i loro usi e benefici, e alcune delle sfide associate alla loro creazione. Iniziamo definendo meglio cosa siano i serious game. Ad esempio, un serious game potrebbe essere utilizzato per addestrare il personale medico a riconoscere i sintomi di una malattia, o per aiutare gli studenti a imparare un nuovo linguaggio. L'idea alla base dei serious game è quella di combinare il gioco con l'apprendimento, in modo da rendere l'esperienza di apprendimento più coinvolgente e memorabile. Ci sono molti benefici nell'utilizzo dei serious game come strumento di apprendimento. In primo luogo, i giochi sono intrinsecamente motivanti e coinvolgenti. I giocatori si impegnano attivamente nella risoluzione dei problemi e nella ricerca di soluzioni, il che rende l'esperienza di apprendimento più interessante e stimolante. Inoltre, i giochi offrono un ambiente sicuro in cui i giocatori possono sperimentare e imparare senza il rischio di conseguenze negative nel mondo reale. Ciò significa che i giocatori

possono fare errori e imparare dalle loro esperienze senza causare danni reali. Ci sono anche alcune sfide associate alla creazione di serious game. In primo luogo, è importante garantire che il gioco sia efficace nell'insegnamento di ciò che si propone di insegnare. Ciò significa che il gioco debba essere accurato dal punto di vista scientifico e debba utilizzare metodi di insegnamento comprovati per massimizzare l'apprendimento del giocatore. Inoltre, il gioco deve essere coinvolgente e divertente, altrimenti i giocatori potrebbero non impegnarsi nel medesimo. Il fenomeno dei "serious games" ha solo recentemente raggiunto una certa notorietà, per quanto riguarda l'ambito digitale. La vera nascita del settore e dell'espressione nella sua accezione attuale è convenzionalmente legata alla creazione della Serious Games Initiative nel 2002 secondo quando espresso da Anolli e Mantovani nel loro libro "Come funziona la nostra mente. Apprendimento, simulazione e Serious Games" (Il Mulino, 2012)[4]. In una prima fase il termine "Serious" indicava l'impiego di giochi digitali di strategia in ambito formativo, tuttavia, nel corso dei primi anni duemila i serious games hanno incluso diverse linee tematiche nei domini più diversi dell'esperienza, con precise finalità di apprendimento. Il 2002 è considerato un momento cardine nella storia dei Serious Games, poiché nel corso di quell'anno avviene la pubblicazione del videogioco "America's Army", progetto interamente finanziato dall'Esercito statunitense, pensato e sviluppato come mezzo per promuovere l'immagine dell'Esercito e favorire il reclutamento di nuovi volontari. Questo prodotto fu un vero punto di svolta nella riflessione sulla funzione dei videogiochi in ambiti che vanno oltre il divertimento e sulle loro potenzialità di promuovere varie forme di apprendimento. Il titolo era giocabile gratuitamente previa registrazione online al sito dal quale era possibile effettuare il download del gioco stesso. Entro il 2006, a 4 anni dal lancio ufficiale, avvenuto, infatti, il 4 giugno del 2002, America's Army poteva contare dieci milioni di utenti registrati sul proprio sito, con oltre 2200 server attivi nel mondo per sostenere il carico delle richieste. Dal 29 Agosto 2013 è stato rilasciato un nuovo capitolo della saga: America's Army Proving Grounds, disponibile per il download attraverso la piattaforma di distribuzione Steam di proprietà di Valve Corporation.
Come già citato precedentemente, i serious games possono essere scissi in tre differenti componenti fondamentali:

- La componente simulativa, infatti, i serious games sono giochi simulativi che puntano a riprodurre alcune caratteristiche dell'esperienza di riferimento, ponendo il focus sugli aspetti considerati salienti dagli ideatori, permettendo, inoltre, di anticipare le evoluzioni future del sistema sulla base delle scelte del giocatore. Una caratteristica sulla quale è possibile dividere i giochi simulativi è il grado di fedeltà al reale; infatti, si possono avere simulazioni dette "ad alta fedeltà" in cui il progresso del mondo virtuale è delineato sulla base di un modello statistico, il quale deve essere in grado di prevedere l'andamento di

Figura 1.2: Scena di gameplay tratta da America's Army Proving Grounds

tutte, o quasi, le variabili che influenzano l'evoluzione effettiva della porzione di realtà che si vuole rappresentare. Poiché la definizione del modello statistico retrostante richiede l'impegno di tempo e risorse adeguate, spesso si ricorre alla creazione di simulazioni "a bassa fedeltà", ossia un tipo di riproduzione in cui si limita il numero di variabili che definiscono il comportamento del modello a quelle reputate di maggiore impatto. In questo secondo caso, soltanto alcuni aspetti del fenomeno verranno rappresentati, a discapito di altri;

- La componente ludica, la quale si configura come motore motivazionale dell'esperienza, poiché è generatrice di sensazioni positive e gratificanti per l'utente. Le motivazioni intrinseche più comuni che spingono le persone a giocare sono la curiosità, il desiderio di esplorazione di nuovi mondi o scoperta di nuove soluzioni ai problemi posti dal gioco stesso, il senso di competizione, sia essa da intendersi contro altri giocatori oppure contro la macchina stessa e, infine, il senso di autoefficacia che ci deriva dalla constatazione di possedere le capacità per superare le sfide che il gioco ci pone;

- La componente di apprendimento. Quest'ultima nasce dal desiderio di trasformare il tempo di utilizzo del serious game in competenze, nozioni o consapevolezza acquisite da parte degli utenti. Per questo motivo spesso l'utilizzo di questi applicativi si inserisce all'interno di una sezione guidata, come, ad esempio, un corso di formazione oppure una lezione scolastica. Questo tipo di situazioni garantisce la presenza di una figura di riferimento, come può essere l'insegnante, che valuti la risposta dei partecipanti all'incontro con questa

inusuale tipologia di strumenti di formazione, ma, contemporaneamente, il contesto gruppale in cui ci si ritrova permette la condivisione di opinioni, aumentando ulteriormente il coinvolgimento dei giocatori.

Quest'ultima componente è quella che maggiormente va a delineare la differenza fra serious games e videogiochi veri e propri, i quali, tuttavia, presentano numerosi punti di contatto. Entrambi gli strumenti condividono il medium digitale, ed attraverso esso sono in grado di generare un elevato coinvolgimento emotivo basato sul gioco che si traduce in potere attrattivo del prodotto, in grado di attivare nello stesso tempo diverse funzioni mentali e quindi favorire un'elevata concentrazione e un impiego prolungato delle risorse cognitive. Nonostante questo, i videogiochi si configurano come forme di intrattenimento adatte ad essere utilizzate nel tempo libero, anche in virtù della capacità di generare emozioni forti, ma non necessariamente fanno riferimento ad un qualche aspetto della realtà. Lo scopo principale dei serious games è l'apprendimento nelle sue varie forme: a tal fine essi fanno ricorso sistematico a diversi dispositivi di apprendimento e si prefiggono di avere un'efficacia osservabile nel miglioramento delle attività professionali e scolastiche dei partecipanti, in termini di estensione del sapere e di gestione delle difficoltà. I serious games si propongono, quindi, di diventare percorsi alternativi e innovativi di apprendimento nei quali la componente di gioco assume valore strumentale, servendo principalmente per mantenere viva la motivazione.

1.7 Feedback in un serious game

In virtù di tali differenti finalità, i due prodotti necessitano dunque di un sistema di feedback interni completamente differente, dove con feedback andiamo ad intendere la reazione del sistema simulativo alle azioni dell'utente. I videogame dedicati al puro intrattenimento utilizzano infatti i feedback come strumento per alimentare la dinamica del gioco e favorire il coinvolgimento dei giocatori. Anche i serious game devono tenere presenti tali dimensioni, tuttavia, nel loro caso, i feedback devono necessariamente possedere una valenza formativa, ossia essere funzionali al raggiungimento di obiettivi di apprendimento. Non esistono modelli capaci di indicare univocamente in che modo si progetta o si utilizza un serious game a fronte di specifici obiettivi didattici e tali limiti toccano quindi direttamente la dimensione del feedback. Esistono tuttavia delle linee guida che definiscono l'efficacia di un feedback formativo che vanno tenute presenti nella progettazione e nell'utilizzo di tali esperienze. Vardisio (2020), psicologo del lavoro e CEO di Entropy KN, società che si occupa di formazione e sviluppo per grandi aziende sia private sia pubbliche, indica tre differenti livelli di analisi del sistema di feedback nei serious game: quello di forma, di tempo e di contenuto. In letteratura esistono diverse riflessioni su queste tre variabili e sul modo in cui interagiscono influenzando

l'efficacia del feedback. Riporterò di seguito una serie di analisi su questi tre aspetti che sembrano trovare riscontro nell'esperienza formativa sulla base degli studi riportati nel paper di Vardisio[5]. La forma di un feedback viene solitamente concepita in termini di specificità. Un feedback può essere definito specifico se tende a suggerire il percorso corretto per migliorare le proprie performance, anziché indicare meramente se tale risposta sia corretta o meno. A tal proposito, esistono ricerche che mostrano come i feedback specifici siano più efficaci rispetto a quelli che mancano di tale caratteristica (Bangert-Drowns, Kulik, C.C., Kulik, J.A. et al. 1991; Pridemore, Klein, 1991). La mancanza di specificità nel feedback potrebbe, d'altra parte, generare frustrazione nel giocatore, il quale sarebbe informato circa l'erroneità delle proprie azioni, senza essere comunque in grado di comprenderne le motivazioni. Trattando di serious game, una dimensione attribuibile alla forma che riteniamo particolarmente interessante è quella che scaturisce dalla distinzione tra conoscenza tacita e conoscenza esplicita. Come afferma Vardisio:

"La conoscenza tacita è un tipo di sapere che gli individui acquisiscono attraverso lo svolgimento di una certa esperienza. Agire all'interno di un certo "mondo", di una certa comunità, per esempio, insegna agli individui a riconoscere le caratteristiche di quell'ambiente e le regole che lo governano senza aver bisogno di consultare codici o norme. Al contrario, dunque, la conoscenza esplicita è quella dichiarata, ossia formalizzata in qualche modo per essere resa fruibile".

Benché i feedback che mirano a fornire conoscenza esplicita debbano essere indubbiamente presenti, particolarmente interessante sembra la possibilità dei serious game di agire a livello di conoscenza tacita, e ciò per due ordini di ragioni: in primo luogo per l'importanza che è stata attribuita a questo genere di conoscenza rispetto alla capacità di comprendere e di agire all'interno di un certo ambiente. Il serious game in questo senso amplia le possibilità dell'utente di individuare i percorsi corretti con maggiore rapidità, riuscendo dunque a diminuire i tempi d'azione, ed al contempo ottimizzare gli esiti delle azioni, degli operatori all'interno del suddetto mondo. In secondo luogo, per il fatto che questi strumenti danno la possibilità di creare liberamente ambienti all'interno dei quali realizzare esperienze ricche, personalizzate e al contempo controllabili da chi è interessato a orientarle verso obiettivi specifici. Per queste ragioni, dunque, le potenzialità in termini di conoscenza tacita dei serious game appaiono come uno degli orizzonti più interessanti in termini di sviluppi futuri di questi strumenti. Si tratta di potenzialità che hanno bisogno di tempo per essere esplorate e comprese meglio.
Una seconda dimensione che concorre a determinare l'efficacia del feedback è quella del tempo. La riflessione su questa variabile potrebbe tradursi in una domanda, ovvero: secondo quale tempistica un serious game dovrebbe fornire feedback all'individuo in apprendimento? la frequenza ideale di intervento del sistema in aiuto del giocatore deve tener conto di aspetti differenti. Una quantità eccessiva di aiuti ed informazioni porterebbe rapidamente l'utente a considerare il proprio libero

arbitrio inutile, facendo crollare la motivazione, restituendo l'idea di un sistema che si auto-esplica, e dunque incapace di stimolare curiosità, desiderio di esplorazione e quindi motivazione. D'altro canto, l'assenza totale di feedback di aiuto, o la loro scarsità, potrebbe implicare la costituzione di una sfida eccessivamente complessa per l'utente, foriera di frustrazione. Urge dunque comprendere a priori quale frequenza possa mantenersi equidistante da questi due estremi, mantenendo il giocatore in quello stato che in ambito videoludico prende il nome di "flow".
Un modello che sembra più adeguato a descrivere i feedback dal punto di vista del contenuto è quello che distingue tra due categorie: valutazione ed elaborazione. Per valutazione intendiamo un giudizio circa la correttezza di una risposta in base a un criterio predefinito. Per elaborazione intendiamo invece la descrizione, la rappresentazione della risposta in base a un certo modello. All'interno di un serious game la valutazione può assumere forme diverse, dalla semplice indicazione circa il raggiungimento (o meno) del risultato a forme più complesse che evidenziano quanto il risultato ottenuto risulti in linea con gli obiettivi. Anche l'elaborazione può ovviamente assumere forme diverse: commenti, grafici, ma anche analisi delle strategie di gioco oppure "suggerimenti" su strategie alternative da adottare.

1.8 Vantaggi dei serious game

Avendo dato una definizione di serious game ed espresso in quali componenti esso si articola, nonché compreso con quali modalità tarare gli interventi formativi del sistema, passiamo ora a descrivere i principali vantaggi di tali strumenti, rifacendoci sempre a quanto affermato da Vardisio. Secondo lo studioso, l'efficacia dell'apprendimento attraverso questa modalità può essere analizzata sotto differenti punti di vista:

- Conoscenze/competenze: dagli studi psicologici condotti sul mondo dei videogame apprendiamo che tale medium è in grado di sviluppare nei suoi fruitori grandi livelli di attenzione, che a loro volta portano ad un migliore fissaggio di quanto esperito in memoria, ma al contempo anche ad una maggiore capacità di orientazione spaziale;
- Motivazione: la componente giocosa fa sì che queste simulazioni godano delle motivazioni intrinseche derivante dal play;
- Autoefficacia: come descritto da Yee e Bailenson nel 2007 le esperienze fatte attraverso un avatar possono andare ad influire anche sui comportamenti futuri del soggetto, anche quando questo, lasciato il mondo virtuale, ritornerà ad interagire con quello reale. Questo fenomeno sarebbe determinato dall'aumento di autostima derivante dal successo riportato all' interno della simulazione, che persisterebbe nel tempo, di fatto generando sicurezza nell'individuo;

- Gestione della complessità: tali simulazioni, come già descritto, sono in grado di semplificare modelli estremamente complessi. In questo modo è possibile inserire il soggetto all'interno di un mondo che nella realtà risulterebbe troppo pregno di variabili perché un singolo individuo, oppure un piccolo gruppo, possano sperare di controllarlo, ma al contempo porre in mano agli utilizzatori degli strumenti per modificare quel mondo nella sua, pur se ridotta, complessità. La simulazione si propone come laboratorio del nuovo millennio;

- Oggettività dei dati: il medium digitale non è da considerarsi unicamente come supporto realizzativo, ma anche come dispositivo di tracciamento delle performance dei singoli. La quantificazione della bontà delle scelte prese dagli utenti è di grande utilità per gli organizzatori dell'esperienza formativa, in modo che essi possano comprendere come migliorare la stessa; ma al contempo questo tracciamento giova anche ai giocatori, che possono confrontare i propri risultati con quelli degli altri, generando così un proficuo scambio di opinioni. Non a caso in ambito videoludico numerose sono le comunità di utenti che si confrontano sulla base di simili sistemi (classicamente i punteggi oppure le classifiche);

- Socialità: i giochi digitali hanno negli anni dimostrato di essere in grado di conglomerare intorno a se stessi grandi comunità di utenti. Sulla base di questa caratteristica è bene immaginare esperienze che mirino alla socialità, che essa sia perseguita attraverso sessioni di utilizzo del serious game in modalità gruppale, oppure che si tratti di interazioni virtuali;

- Trasversalità ed innovatività: il mezzo ludico si adatta perfettamente alla modernità, poiché il bisogno di sviluppare continuamente nuove abilità, o di imparare a conoscere concetti e ambienti sempre nuovi, ha fatto sorgere la necessità di uno strumento di facile utilizzo, poco stressante ed al contempo efficace nelle sue finalità educative. Il meccanismo di apprendimento basato sulla correzione dei propri errori, inoltre, impone agli utilizzatori la necessità di rivedere i propri schemi decisionali in grande rapidità, sapendo adattarli correttamente al contesto.

Sulla base dei vantaggi appena esposti possiamo affermare che i serious game si configurano come ottimi strumenti di apprendimento. Dovendo progettare correttamente uno strumento di questo tipo ci ritroviamo a scontrarci con la realizzazione della sua veste giocosa, la quale, mutando i propri principi dal mondo videoludico, deve necessariamente rifarsi al più ampio concetto di game design. Al fine di giustificare le scelte prese in corso di realizzazione del progetto di tesi descriverò alcune linee guida di questa disciplina.

1.9 Note di game design

Parlando del concetto di "gioco" ho già introdotto i cosiddetti elementi "atomici" o " formali", ora li rielaborerò nell'ottica non di darne di una definizione, bensì di comprendere come sia possibile utilizzarli per disegnare l'architettura di un gioco. Riprendiamo dunque il discorso partendo dagli obbiettivi. Essi definiscono quali compiti deve eseguire il giocatore all'interno del mondo che si ha intenzione di creare. Esistono differenti tipologie di obbiettivo, ed ognuna di queste definisce a sua volta il tipo di esperienza che abbiamo intenzione di offrire all'utente. Al contempo, un obbiettivo indica il ritmo proprio di gioco. Quest'ultimo concetto può essere espresso come la frequenza con cui il giocatore è chiamato ad intervenire per permettere l'evoluzione del gioco, ma al contempo anche la complessità delle scelte poste allo stesso. Un alto ritmo do gioco implicherà naturalmente un maggiore ammontare di stress a cui si sottopone l'utente. Da qui comprendiamo come sia importante valutare il target di utenti per i quali sto progettando il gioco anche in sede di definizione dell'obbiettivo. Infatti, comunità di giocatori assidui saranno disposti, se non incentivati, a fruire del nostro prodotto nel caso in cui questo abbia un ritmo molto serrato. I diversi tipi di obbiettivo che si possono identificare sono:

- Cattura: in questo caso l'utente dovrà entrare in possesso di un asset interno al gioco al quale viene dato un valore implicito;
- Corsa: si deve eseguire un certo percorso sotto date condizioni, come, ad esempio, un tempo limite. Classicamente questo obbiettivo viene arricchito ponendo più giocatori in competizione fra loro, confrontandone le performance;
- Allineamento: si devono creare configurazioni muovendo oggetti per creare combinazioni oppure risolvere un enigma. A questa categoria fanno capo tutti i puzzle logici;
- Azione proibita: il giocatore deve evitare una determinata azione, oppure spingere i suoi avversari a compiere quella determinata azione;
- Creazione o costruzione libera: in questo caso si potrebbe dire che l'obbiettivo è assente, poiché il divertimento deriva dalla libertà di eseguire ciò che il giocatore desidera;
- Esplorazione: si spinge il giocatore a muoversi in un ambiente ricco di informazioni da fare proprie;
- Conoscenza e astuzia: un gioco o un insieme di giochi che prevedono che per raggiungere l'obbiettivo primario io debba utilizzare la mia intelligenza sociale per prevalere sul gruppo. Questa tipologia di obbiettivo implica l'interazione

fra utenti, e per questo stesso motivo, nelle versioni digitali di giochi che utilizzano questa tipologia di obbiettivi si inseriscono canali di comunicazione vocale o testuale.

Ognuna di queste tipologie di obbiettivo può configurarsi come principale all'interno di un gioco; tuttavia, è comune, benché non necessario, avere obbiettivi secondari di diversa natura. Internamente il gioco di dispiega attraverso quelle che già sono state definite come "procedure", ossia l'insieme di azioni che l'utente può compiere per interagire con il gioco stesso. La procedura più comune in assoluto è quella di start, essa è presente in ogni gioco digitale ed è implicita in quelli analogici. Questa procedura delinea l'istante in cui i giocatori decidono di dedicare del tempo all'interazione con l'ambiente di gioco. Le procedure interne al gioco vengono generalmente raggruppate in quello che prende il nome di "core loop" del gioco. Le singole procedure racchiuse in esso prendono il nome di "progress action" e fanno riferimento a tutti i compiti che il giocatore è chiamato a svolgere perché l'ambiente di gioco evolva. Il core loop si presenta appunto come un ciclo poiché esso andrà eseguito a ripetizione perché la progressione nel gioco possa effettuarsi. Nonostante questo, possono esistere delle azioni che esulano dal ciclo principale, ma che si configurano piuttosto come azioni speciali che modificano l'interazione standard dell'utente. Generalmente esse vengono inserite al fine di evitare un'eccessiva ripetitività dell'esperienza, che sarebbe demotivante per i partecipanti. A fare da contraltare alle procedure di start o apertura vi son le "resolving actions", ossia quelle procedure che indentificano la fine di una porzione intermedia del gioco, oppure della partita nella sua totalità. Le scelte dell'utente si manifestano dunque entro i percorsi definiti da questi processi, i quali a loro volta implicano l'esistenza dell'ultima famiglia di procedure: le procedure di sistema. All'interno di questo raggruppamento andremo a porre tutti quei processi non direttamente eseguibili per volere dell'utente. Essi sono necessari perché il gioco calcoli correttamente le conseguenze delle decisioni prese dall'utente, potendo di conseguenza modificare il proprio stato con coerenza. Per fare un esempio, è una procedura di sistema il calcolo dei punti accumulati da un giocatore. Un'ulteriore scelta da prendere in fase di design è quella legata alla tipologia di risorse che si vogliono fornire all'utente. Storicamente i giochi presentano due risorse fondamentali: una prima risorsa che se esaurita pone termine alla partita (ad es. le vite nei videogiochi cabinati), in genere decretando una sconfitta; ed una seconda che invece viene utilizzata, o più comunemente spesa, per poter progredire lungo il percorso delineato dall'esperienza (ad es. i soldi nel Monopoly). Esiste poi una terza tipologia di risorsa, ossia quella che non viene perduta o acquisita ma alla quale si può fare ricorso per interagire con l'ambiente. Teniamo conto che i giochi fanno largo uso del tempo come risorsa, definendo una deadline ultima entro la quale poter raggiungere i propri obbiettivi. Non riprenderò qui ognuno degli elementi formali precedentemente descritti per

declinarlo in ambito di game design; mi fermerò, dunque, a quanto espresso fin qui, poiché i concetti espressi come possibilità si tradurranno nelle scelte prese per l'ideazione del serious game oggetto di questa tesi. La somma di tutti gli elementi formali crea quindi il gioco vero e proprio. A questi si aggiungono poi gli elementi drammatici. Per via delle necessità espresse dal serious game il cui sviluppo presenterò più avanti, indagherò soltanto la sfida come elemento drammatico, ed in particolare per la sua correlazione con lo stato di "flow". Abbiamo già detto che il flow, che potremo identificare come l'obbiettivo principale dei giochi, si raggiunge se si fa ingaggiare all'utente una sfida proporzionata alle sue capacità. In qualunque esperienza della vita noi siamo posti davanti a degli ostacoli da superare, e il tipo di interazione con essi può far capo a tre strategie di base. Noi potremmo cimentarci, infatti, con il superamento della sfida più e più volte, ottenendo però sempre un esito negativo. In questo caso non ci resterebbe che aggirare l'ostacolo anziché affrontarlo. Questa decisione in ambito ludico si traduce con l'abbandono dell'esperienza di gioco. Un'altra possibile opzione è quella per cui la sfida è facilmente superabile sulla base delle nostre capacità, e dunque, se costretti a ripetere un gran numero di volte l'attività, finiremmo di incappare nell'esecuzione automatica. Quest'ultima modalità d'azione è chiaramente demotivante e dunque entra in conflitto con le aspettative del giocatore. La terza via è rappresentata da una sfida inizialmente troppo complessa da essere superata, benché dopo un numero ragionevole di tentativi si riesca ad affrontarla con esito positivo. In quest'ultimo caso il giocatore esperisce la gratificazione data dal premio dei propri sforzi. Il mantenimento dell'utente in quest'ultima fase costituisce il citato "stato di flow".

Mihaly Csikszentmihalyiha, psicologo ungherese emigrato negli Stati uniti, nel suo libro "Flow. Psicologia dell'esperienza ottimale" del 2021[6] ha indagato il tema del raggiungimento dello stato di flow, individuando otto elementi fondamentali per l'ottenimento di un'esperienza appagante:

- Il confronto con compiti reputati sfidanti, ma, al contempo, pur sempre risolvibili secondo le nostre capacità;
- Il mantenimento di un tale livello di concentrazione da riuscire a far risultare semplici complesse sequenze di azioni. I giochi, specie quelli che coinvolgono la coordinazione motoria, possono portarci a memorizzare processi articolati con grande facilità, per portarci poi a rieseguirli con familiarità;
- L'obbiettivo posto come meta finale deve essere chiaro ed il successo nel suo raggiungimento deve essere ricompensato immediatamente;
- Quando la concentrazione è tanto alta da generare il fenomeno dell'isolamento spaziale dell'individuo, per cui l'immersione all'interno del medium ludico è tale da portare ad ignorare parzialmente, ed in modo unicamente temporaneo, l'ambiente esterno;

- La qualità dell'esperienza sale se riesce a generare nel giocatore il cosiddetto "paradosso del controllo". All'interno di un gioco il sistema comunica al giocatore che qualunque modifica lui voglia applicare allo stesso è plausibile. Per questo motivo l'utente ha la percezione di poter controllare perfettamente il mondo in cui è inserito. Questa illusione è in grado di reggere solo se, quando il giocatore commette un errore il gioco, è capace di reagire fornendo all'utente un indizio sulle ragioni di quello sbaglio. Se così fosse, il partecipante accetterebbe l'errore, convinto di poterlo evitare in futuro. Questo "paradosso del controllo" sta nel fatto che gli utenti accettano gli sbagli e li comprendono, mantenendo l'illusione di poter controllare l'ambiente pur commettendo errori;
- L'esperienza ottimale può portare ad un'inibizione dell'autocoscienza, per cui in fase di gioco siamo disposti a compiere azioni che normalmente potrebbero generare in noi disagio (ad es. ballare);
- Un alto livello di concentrazione porta ad esperire una trasformazione del tempo percepito. Il giocatore in media sottovaluta il tempo trascorso dall'inizio dell'esperienza ludica, e, al contempo, è possibile sottostimare la propria stanchezza fisica o psicologica;
- L'esperienza deve essere fine a se stessa. Il giocatore deve quindi desiderare personalmente di entrare nell'universo ludico, senza che questo implichi alcun tipo di ricompensa materiale, o che sia sottoposto ad un obbligo di qualche tipo.

Avendo ora esposto i benefici derivanti dalla generazione dello stato di flow, ed avendo già esposto come questo sia direttamente dipendente dal tipo di sfida che il gioco propone, nonché dalla complessità di quest'ultima, si comprende l'importanza che in fase di design si deve dare alla sua definizione. La sfida non è però da intendersi unicamente come la prova che il prodotto ludico sottopone al giocatore, ma anche come l'accessibilità che questa è in grado di garantire. Un serious game ben progettato dovrebbe, ad esempio, tener conto della possibilità di essere completamente fruibile anche da giocatori con disabilità. La regola aurea del mondo videoludico vuole che la sfida si costituisca per gradi: la cosa migliore da fare per mantenere i giocatori motivati è introdurre una serie di strumenti di base che il nostro utente deve imparare ad utilizzare, mentre solo quando avrà imparato ad utilizzarli in modo perfetto io gli proporrò nuovi mezzi con cui risolvere anche task più complessi.

1.10 Game design del caso di studio

Il primo passo per lo sviluppo del serious game oggetto di questa tesi è stato la definizione delle meccaniche di gioco, le quali, a loro volta, dovevano soddisfare una

serie di requisiti espressi dall'azienda committente. Edilclima si impegna, come già detto, in attività di formazione presso istituti medi inferiori e superiori, ed è proprio in questa sede che si progettava l'inserimento del serious game. Dalle richieste esposte inizialmente dall' azienda sono emersi i seguenti requisiti:

- Gli interventi dei tecnici dell'azienda miravano a sviluppare negli studenti una consapevolezza circa le mansioni principali di un termotecnico, nonché le importanti implicazioni degli interventi operati dall'azienda al fine di riqualificare energeticamente gli edifici del territorio. Sulla base di questo si è deciso che il gioco avrebbe dovuto immergere ulteriormente gli utenti all'interno del mondo operativo di queste figure professionali, illustrandone il punto di vista e l'approccio operativo;
- Il gioco doveva essere di facile comprensione, poiché gli interventi formativi negli istituti hanno una durata usale di due ore, per cui non più di mezz'ora poteva essere concessa all'utilizzo dell'applicazione. I tempi stringenti implicavano la creazione di un gioco che ruotasse intorno ad un numero limitato di meccaniche, le quali avrebbero dovuto recuperare modelli mentali già conosciuti dagli studenti perché comunemente presenti in altri giochi;
- La modalità gruppale dell'esperienza di utilizzo del serious game imponeva che esso si configurasse come un multi-giocatore; tuttavia, considerate le dimensioni del gruppo (si è deciso di considerare una trentina di utenti contemporanei, essendo questa mediamente l'entità di una classe) non si poteva ricorrere ad un meccanismo a turni singoli, poiché questo avrebbe implicato tempi d'attesa eccesivi per i giocatori. Il serious game doveva quindi ammettere la possibilità che più utenti agissero contemporaneamente;
- Il progetto, benché unitario, sarebbe stato diviso in due differenti sezioni: la prima sarebbe stata quella deputata all' utilizzo da parte dei giocatori attraverso i propri smartphone, mentre la seconda sarebbe stata invece d'utilizzo esclusivo per il "game master", ossia colui che si sarebbe occupato di dare inizio alla partita. All' interno di questa seconda sezione si è deciso di inserire un ambiente comune che si sarebbe modificato sulla base delle scelte degli utenti, di fatto costituendo il mondo di gioco vero e proprio. l'ambiente comune sarebbe stato infine proiettato sulle lavagne smart in dotazione alle aule presso gli istituti del territorio;
- Dovendo garantire la possibilità d'utilizzo del serious game tramite gli smartphone di proprietà degli studenti sarebbe stato necessario sviluppare un applicativo cross-piattaforma;
- Non si poteva dare per certo che fosse possibile proiettare il contenuto dello smartphone del game master sulla lavagna interattiva. Questa possibile

problematica ha fatto sorgere la necessità di sviluppare un applicativo web raggiungibile attraverso i computer in dotazione alle classi sul territorio, poiché questi ultimi sono direttamente collegati alle suddette lavagne interattive.

L' insieme di queste necessità hanno, nel complesso, portato alla definizione del game design del prodotto, ma, contemporaneamente, hanno anche guidato la definizione degli strumenti tecnici con i quali si sarebbe effettuato lo sviluppo. Dovendo riassumere la complessità del mondo operativo dell'azienda, nasceva la necessità di strutturare il gioco intorno ad un numero finito di elementi, ognuno dei quali avrebbe esposto una differente operazione possibile per la riqualificazione energetica. Considerando, inoltre, la necessità di mutuare schemi di gioco già conosciuti dagli utenti, il serious game doveva necessariamente appartenere ad un genere videoludico noto. Per queste due motivazioni si è deciso di sviluppare un card game: infatti questa scelta ci avrebbe permesso di sfruttare le singole carte per indicare i possibili interventi che comunemente un termotecnico progetta in ambito lavorativo, ma, allo stesso tempo, godendo il genere di grande successo in ambito videoludico mobile, è presumibile che gli utenti riconoscano rapidamente gli schemi introdotti nel gioco. Definito il genere videoludico di riferimento, si è passati a sviluppare il design del gioco vero e proprio. All'inizio della partita i giocatori sarebbero stati divisi in 4 differenti squadre. La modalità a squadre è stata preferita a quella a giocatori singoli per incentivare l'interazione fra i componenti del gruppo di gioco. Il gioco sarebbe stato composto da differenti livelli, ognuno dei quali avrebbe proposto un contesto lavorativo differente, sulla base di quelle che sono le tipologie di edifici che più comunemente subiscono interventi di riqualificazione energetica. Il primo livello avrebbe dunque proposto di operare su un'abitazione singola, il secondo su di un appartamento inserito in un condominio, ed, infine, il terzo avrebbe fatto intervenire i giocatori su di un condominio nella sua interezza. Nonostante gli edifici su cui lavorare in ogni livello rimangano sempre gli stessi, ad ogni partita sarebbe cambiato il contesto in cui esso sarebbe stato inserito. Poiché in ambito operativo reale un'abitazione può, ad esempio, trovarsi in un contesto in cui il geotermico è favorito, mentre il solare è scoraggiato. I possibili contesti sarebbero stati: mare, montagna e città. All'inizio di ogni livello il contesto sarebbe stato scelto casualmente. Questa soluzione è stata adottata al fine di aumentare la rigiocabilità del prodotto. Ogni livello avrebbe avuto un tempo predefinito per essere completato che è stato fissato a 7 minuti. In questo modo la partita sarebbe durata in toto circa venti minuti, ai quali si sarebbe dovuto aggiungere il tempo per completare un tutorial inziale che presenta l'applicativo ai giocatori. Queste tempistiche ci avrebbero garantito di restare entro il limite di trenta minuti posto come tempo di utilizzo massimo del serious game in sede scolastica. All'inizio di ogni livello ad ognuna delle squadre viene dato un obbiettivo differente. Gli obbiettivi possono fare riferimento a tre macroaree: diminuire l'impatto ambientale di un

edificio, diminuire il consumo energetico dell'edificio oppure aumentare il comfort degli abitanti degli abitanti dell'edificio stesso. All'inizio di ogni livello, alle squadre sarebbe stato fornito un budget iniziale che può essere utilizzato per progettare gli interventi. Il suddetto budget può essere differente in base al contesto in cui è inserito l'edificio. I giocatori dovranno sviluppare un progetto di interventi da effettuare sull'edificio nell'arco di 12 mesi. Per poter decidere quali lavori compiere in ogni mese i giocatori dovranno giocare delle carte rappresentanti l'intervento su di un tabellone rappresentante i vari mesi. La singola carta avrebbe riportato quattro dati differenti: l'impatto ambientale dell'intervento, il suo costo, l'impatto energetico dello stesso e, infine, l'aumento o diminuzione di comfort per gli abitanti che l'intervento avrebbe implicato. All'inizio di ogni livello ai giocatori di ogni squadra verranno distribuite alcune carte selezionate in modo casuale. Durante il livello le squadre giocano contemporaneamente, ma internamente alla squadra vige un sistema a turni, per cui solo un giocatore alla volta gioca per ogni squadra. Questa modalità avrebbe mischiato dunque una classica logica a turni con una a gioco simultaneo; in questo modo si è ritenuto di trovare il giusto compromesso per ottenere un ritmo di gioco di media intensità. Ogni giocatore può effettuare una sola azione durante il suo turno, scelta fra:

- Giocare una carta in suo possesso in una posizione libera sul tabellone della sua squadra;
- Prendere una carta dal tabellone liberando la posizione corrispondente.

Operata l'azione scelta, il turno del giocatore finisce ed il computer fa partire il turno del giocatore successivo nella squadra. Alla fine del turno di ogni giocatore, il computer sceglie casualmente una carta da scartare dalla sua mano, e poi fa pescare un'altra carta casuale allo stesso giocatore. Le performance di alcune carte possono dipendere dal fatto che altre carte siano state già giocate prima, oppure che l'intervento venga progettato per un mese invernale piuttosto che estivo, o ancora dal contesto casuale in cui l'edificio è stato inserito. Ogni edificio avrebbe avuto un impatto ambientale, energetico ed un comfort per gli abitanti quantificati attraverso una scala arbitraria. Posizionare o prendere una carta dal tabellone avrebbe implicato la modifica di ogni indicatore globale del valore corrispondente sulla carta (i valori sono da intendersi come delta). Al termine di ogni livello si valuta il punteggio di ogni squadra, il quale dipende da:

- Raggiungimento degli obbiettivi di squadra affidati ad inizio livello;
- Adeguatezza degli interventi scelti sulla base del contesto in cui è inserito l'edificio;
- Costo complessivo delle opere selezionate.

Andando a riprendere quelli che erano gli elementi formali evidenziati nel capitolo precedente, potremmo assumere di aver scelto una tipologia di obbiettivo principale che recupera le caratteristiche dell'"allineamento", infatti il compito che gli utenti dovranno svolgere è di fatto quello di comprendere quali tessere del puzzle (le carte) posizionare nei punti corretti (i mesi dell'anno) al fine di massimizzare il punteggio della squadra. Nonostante questo, il limite temporale imposto al livello è invece da ascriversi all'insieme degli obbiettivi "corsa", così come a tale gruppo fa capo la dinamica della costituzione di una classifica dettata dal punteggio raggiunto dalle squadre. La scelta di questi campi semantici fra quelli possibili per gli obbiettivi è stata derivata direttamente dal mondo dei card game, i quali generalmente possono essere inseriti nel contenitore dei giochi ad "allineamento", benché gli stessi giochi in ambito mobile spesso presentino la commistione con elementi "corsa", come le già citate classifiche. Per quanto concerne le procedure di start si è deciso di fare in modo che il game master possa generare un qr code, il quale avrebbe facilmente permesso agli utenti di collegarsi alla partita semplicemente inquadrando lo stesso. Una volta collegati tutti i giocatori, il game master può dare inizio alla partita vera e propria. Il core loop del gioco è costituito dalla procedura di decisione di selezione dell'intervento da eseguire sull'edificio e di decisione del mese d'esecuzione dei lavori. Giocata la carta avverrà la procedura di scarto casuale di una carta e conseguente pescata casuale. Quest'ultima operazione, benché inserita nel core loop, è da considerarsi come un'azione speciale, poiché mira specificatamente ad aggiungere imprevedibilità alla partita. Le resolving actions e le operazioni di sistema sono eseguite direttamente da una logica di gioco in-app presente sia lato utente che lato game master. Volendo infine identificare le risorse del gioco, potremmo individuarne tre:

- Il tempo è da intendersi come risorsa ad esaurimento, poiché terminata di fatto si pone termine al livello/alla partita;
- Il budget fornito alle squadre è la risorsa ad esaurimento utilizzata per poter eseguire le proprie mosse;
- Le carte sono risorse utilizzate direttamente nel core loop per poter modificare l'ambiente di gioco.

Sulla base delle caratteristiche fin qui descritte e del processo di game design effettuato, si è scelto di operare attraverso un framework di sviluppo mobile cross-piattaforma, il quale avrebbe garantito la libera fruibilità dell'applicativo da parte degli utenti indipendentemente dallo smartphone in loro possesso. La scelta è ricaduta infine sul framework Flutter, che presenterò nel prossimo capitolo. Lato back-end, richiedendo il semplice hosting di dati, senza dunque la necessità di implementare una vera e propria logica di business lato server, si è deciso di

utilizzare Firebase in virtù della sua semplicità di utilizzo ed implementazione in particolare per traffici modesti.

Capitolo 2

Applicazioni *mobile*

Con il termine di applicazione mobile andiamo ad intendere un software specificatamente disegnato e sviluppato per poter essere fruito attraverso l'utilizzo di dispositivi mobile, come potrebbeero essere smartphone, tablet e wearable. Il mondo delle app mobile è in costante crescita, trainato dall'enorme successo ottenuto dai dispositivi mobili, con questi ultimi divenuti di fatto strumenti imprescindibilmente entrati nella quotidianità di ognuno di noi. Le necessità realizzative del mondo software mobile hanno portato alla creazione di strumenti ad hoc per la loro realizzazione. Oggigiorno stanno prendendo piede strumenti di sviluppo cross-piattaforma, poiché, garantendo la possibilità di sviluppare un'unica code base, hanno reso possibile la riduzione di tempi di lavoro e costi annessi. Oggi disponiamo di un gran numero di strumenti di sviluppo per l'ambito mobile, benché questi possano tutti essere ricondotti a due filoni realizzativi principali, che a loro volta implicano l'implementazione di differenti applicazioni: native oppure cross-platform. Le applicazioni native sono app che si installano e si utilizzano interamente sul dispositivo mobile e sono create appositamente per uno specifico sistema operativo. Il fatto che tali app siano sviluppate per interfacciarsi con un unico sistema operativo permette un livello di ottimizzazione del software di alto livello; inoltre, per le stesse ragioni, i linguaggi utilizzati garantiscono il completo accesso alle potenzialità del sistema, nonché all' utilizzo dei sensori del device. Nel caso in cui un'app nativa voglia essere prodotta per un sistema operativo differente, l'unica soluzione sarebbe riscrivere l'applicazione interamente. Le applicazioni native si scrivono in Java o Kotlin per i device Android, mentre per gli smartphone iOS i linguaggi utilizzabili sono Objective-C e Swift, entrambi di proprietà di Apple.

2.1 Applicazioni cross-platform

La applicazioni cross-platform sono app installabili su tutti i device, esse funzionano indipendentemente dal sistema operativo, il che permette di sfruttare un'unica code base per poter effettuare il delivery del software su entrambi gli store proprietari di Google ed Apple. L'utilizzo di tali soluzioni abbassa notevolmente i tempi di produzione, ed al contempo permette di restringere il budget necessario per lo sviluppo, permettendo, oltretutto, di ottenere buone performance da parte dell'applicativo. Tali vantaggi giungono, tuttavia, ad un costo, poiché le performance si mantengono inferiori rispetto a quelle raggiungibili da un'app nativa. Inoltre, il fatto di non utilizzare strumenti sviluppati per uno specifico sistema operativo può rendere complesso l'accesso ad alcune funzionalità, come, ad esempio, quelle relative all'utilizzo dei sensori propri del device. Per queste ragioni le aziende produttrici di software mobile sono spesso ricorse a soluzioni ibride, in cui, dunque, alcune porzioni dell'app sono scritte in codice nativo, mentre altre saranno scritte in codice cross-piattaforma. Queste soluzioni implicano una certa complessità derivante dalle comunicazioni fra le sezioni del software scritte in linguaggi differenti, ma permettono di trovare il giusto compromesso fra i vantaggi di entrambe le soluzioni. In passato, le applicazioni più utilizzate erano scritte in codice nativo, ma la tendenza sembra essersi modificata recentemente, principalmente in virtù del rilascio di strumenti di sviluppo cross-platform molto efficienti, quali React Native, Ionic, Cordova e Flutter, con quest'ultimo che in base al survey di statista nel 2021 è divenuto il famework cross-platform più utilizzato al mondo [7]. I motivi per cui orientarsi verso lo sviluppo di applicazioni utilizzando un'implementazione platform-independent sono molteplici:

- Manutenibilità: l'utilizzo di un'unica code base garantisce di diminuire i tempi necessari per le ottimizzazioni e/o le correzioni da operare sul software. Contemporaneamente, per le stesse ragioni è possibile implementare team di app manteinance più compatti, aumentando l'efficienza dello stesso contenendone i costi;

- Rapidità di sviluppo: come già detto, i tempi di sviluppo vengono considerevolmente diminuiti.

- Buone performance: nonostante esse si mantengano al di sotto di quelle ottenute dal codice nativo, framework quali Flutter e React Native garantiscono ottime performance.

- Possibili sviluppi del mercato mobile: in ottica futura, per quanto riguarda gli sviluppi del mercato delle app, può rivelarsi un asset strategico la conoscenza di linguaggi e framework cross-piattaforma.

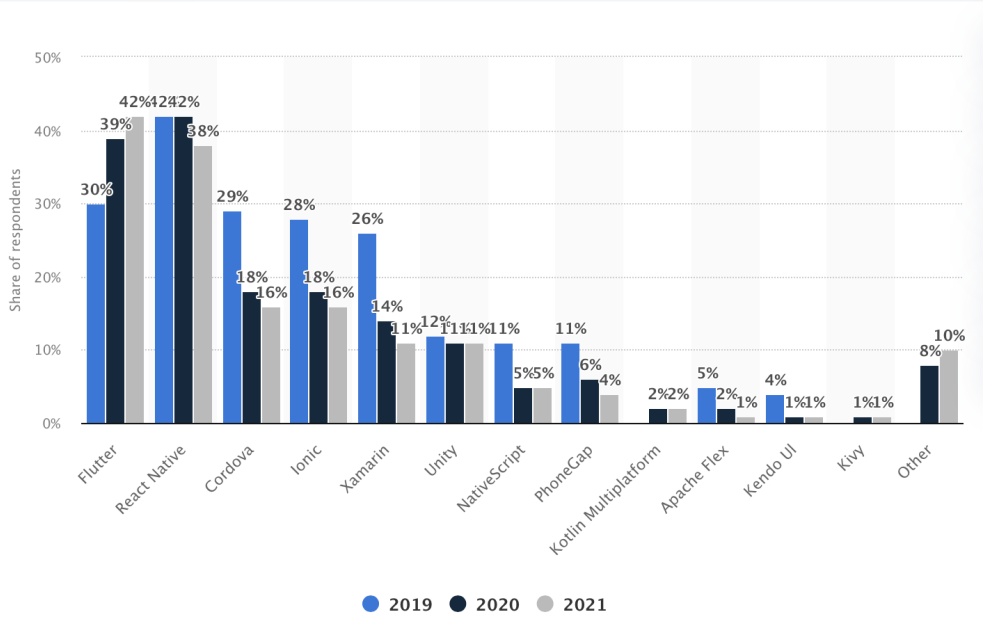

Figura 2.1: Utilizzo dei framework platform-independent

2.2 Esempi di framework platform-independent

Attualmente sono disponibili numerosi framework platform-independent. Ognuno di essi implementa soluzioni differenti, per questo motivo esporrò brevemente le caratteristiche dei framework più utilizzati secondo l'indagine di statista precedentemente esposta. In questa lista non apparirà Flutter, che sarà oggetto di un capitolo a parte essendo il framework utilizzato per il progetto oggetto di questa tesi.

2.2.1 React Native

React Native [8] è un framework open-source per lo sviluppo di applicazioni mobili cross-platform. Utilizzando il linguaggio di programmazione JavaScript e la libreria React, React Native consente ai programmatori di creare applicazioni per iOS e Android con una sola base di codice. React Native utilizza un'architettura a componenti, che consente ai programmatori di suddividere l'applicazione in parti più piccole, facili da gestire e riutilizzare. Questo rende il codice più organizzato e modulare, semplificando anche la manutenzione dell'applicazione. In React Native, i componenti sono scritti in JavaScript e poi compilati in codice nativo per iOS o Android. Ciò significa che le applicazioni sviluppate in React Native hanno prestazioni molto simili a quelle native e una UI reattiva, ma senza la necessità

di scrivere codice nativo per entrambe le piattaforme. React Native supporta anche l'hot reloading, che consente ai programmatori di vedere immediatamente i cambiamenti apportati all'applicazione senza dover ricompilare l'intero progetto. Ciò rende il processo di sviluppo più rapido e produttivo. Inoltre, React Native è supportato da una vasta comunità di sviluppatori che contribuiscono alla libreria e forniscono supporto tecnico. Inoltre, sia React che React Native sono prodotti da Meta, di conseguenza gli utilizzatori possono contare su un costante sviluppo ed ottimizzazione del framework, dato che Meta stessa la utilizza per sviluppare le proprie app mobile, come Facebook ed Instagram. L'insieme di queste caratteristiche ha portato React native ad avere un grande successo presso gli sviluppatori, tanto che esso risolta essere il framework cross-piattaforma più utilizzato secondo il survey di statista, eccezione fatta per il sorpasso effettuato da Flutter nel 2021. In sintesi, React Native è un framework potente e flessibile per lo sviluppo di applicazioni mobili cross-platform, che utilizza il linguaggio di programmazione JavaScript e la libreria React per creare applicazioni native per iOS e Android con una base di codice condivisa.

2.2.2 Xamarin

Xamarin [9] è una piattaforma di sviluppo cross-platform open-source che permette di creare applicazioni mobili per iOS, Android e Windows utilizzando il linguaggio di programmazione C. Xamarin offre una vasta gamma di strumenti e librerie di sviluppo che semplificano il processo di creazione di applicazioni mobili e forniscono una vasta gamma di funzionalità. Anche Xamarin, come React Native, propone un'architettura a componenti, dati gli evidenti vantaggi di questa soluzione. Inoltre, Xamarin fornisce l'accesso alle funzionalità native del dispositivo tramite il Xamarin.Forms API. Ciò significa che gli sviluppatori possono utilizzare funzionalità native come la fotocamera, la geolocalizzazione, i sensori e molte altre funzionalità all'interno delle proprie applicazioni. Inoltre, Microsoft ha acquisito Xamarin e lo ha integrato nella sua suite di strumenti di sviluppo, offrendo agli sviluppatori un'esperienza di sviluppo integrata e completa. Per questo motivo Xamarin utilizza la suite .NET, largamente utilizzata e conosciuta dagli sviluppatori, il che permette di diminuire i tempi di formazione per l'utilizzo del framework.

2.2.3 Ionic

Ionic [10] è un framework open-source per lo sviluppo di applicazioni mobili ibride utilizzando tecnologie web come HTML, CSS e JavaScript. Ionic fornisce un'ampia gamma di strumenti e librerie di sviluppo che semplificano il processo di creazione di applicazioni mobili e offrono una vasta gamma di funzionalità. Ionic utilizza il framework Angular di Google come base, offrendo una vasta gamma di componenti

predefiniti e personalizzabili, oltre alla possibilità di integrare librerie di terze parti. Ionic utilizza il framework CSS open-source Bootstrap per fornire un'ampia gamma di componenti UI personalizzabili e adattabili. Ionic è una soluzione ibrida multipiattaforma estremamente comoda perchè consente agli sviluppatori Web di continuare a utilizzare un framework front- end molto conosciuto: Angular. Incorporando Angular nel framework, Ionic facilita la transizione dallo sviluppo web a quello mobile. Il più grande punto debole di Ionic deriva dal fatto che, come per altri framework di sviluppo di applicazioni ibride, le prestazioni delle app realizzate tramite Ionic possono non essere ottimali.

Capitolo 3

Dart

Dart [11] è un linguaggio di programmazione general-purpose sviluppato da Google, che è stato annunciato nel 2011 e ha visto la sua prima release nel 2013. Dart è stato progettato per essere facile da imparare e da usare, e allo stesso tempo potente e flessibile. In questo senso, può essere utilizzato per sviluppare una vasta gamma di applicazioni, tra cui applicazioni web, mobile e desktop. Una delle caratteristiche principali di Dart è la sua capacità di essere eseguito su diverse piattaforme. Dart può essere eseguito su browser web tramite il compilatore dart2js, che genera codice JavaScript compatibile. In alternativa, Dart può essere eseguito su server tramite il runtime DartVM, che fornisce una maggiore velocità e una migliore gestione della memoria rispetto a JavaScript. Dart è un linguaggio fortemente tipizzato. Questo significa che Dart è in grado di fornire una maggiore sicurezza del tipo rispetto a linguaggi come JavaScript, senza la necessità di scrivere codice molto verboso. Il typing, infatti, permette di rilevare bug al momento stesso della compilazione, in modo tale che gli sviluppatori siano in grado di identificare e correggere rapidamente gli errori. Inoltre, la tipizzazione impone ai programmatori di scrivere codice più facilmente leggibile e meno ambiguo. I vantaggi dei linguaggi tipizzati hanno portato anche il JavaScript, che faceva della tipizzazione dinamica una sua caratteristica peculiare, ad evolversi nel progetto TypeScript. Dart supporta anche il paradigma di programmazione asincrona, che è utile per scrivere codice reattivo e per lavorare con eventi in tempo reale. Ciò è reso possibile da una caratteristica del linguaggio chiamata Future, che permette di eseguire codice in modo asincrono. Essendo Dart il linguaggio utilizzato in Flutter passerò ora a descriverne alcuni meccanismi caratteristici.

3.1 Principali caratteristiche di Dart

Di seguito presenterò alcune delle caratteristiche principali del linguaggio Dart: dynamic, Future, Async e Stream.

3.1.1 dynamic

Dart è un linguaggio fortemente tipizzato ed è dotato di inferenza. Resta comunque possibile non indicare alcun tipo per le proprie variabili, ed in questo caso Dart considererà implicitamente il dato di tipo “dynamic”. In Dart, il termine "dynamic" è utilizzato come tipo di dati speciale che rappresenta un valore senza alcuna informazione sul tipo a tempo di compilazione. Questo significa che il tipo di dati "dynamic" consente di scrivere codice che può accettare valori di qualsiasi tipo a tempo di esecuzione. Il tipo di dati "dynamic" è particolarmente utile quando si lavora con dati provenienti da fonti esterne, come ad esempio un database o un servizio web, dove non è sempre possibile conoscere in anticipo il tipo dei dati che si riceveranno. Questo tipo speciale è altrettanto indicato per situazioni in cui si prevede il cambio di tipo della variabile a runtime. In questi casi, il tipo "dynamic" consente di scrivere codice più flessibile e adattabile alle varie situazioni. L’utilizzo eccessivo del tipo "dynamic", tuttavia, può rendere il codice meno sicuro e meno efficiente, in quanto non consente al compilatore di effettuare controlli sul tipo di dati durante la compilazione. Per questo motivo, è importante utilizzare il tipo "dynamic" solo quando necessario e cercare di utilizzare tipi più specifici quando possibile. Inoltre, in Dart tutti i tipi ereditano dalla classe generica Object. La differenza fondamentale fra “dynamic” ed Object risiede nel fatto che se sul primo si chiama un qualsiasi metodo non ammesso il sistema non restituirà un errore, benché questo verrà rilevato a runtime, mentre il secondo impedisce la compilazione in questo caso.

3.1.2 Future

In Dart, un Future è un oggetto che rappresenta un valore che non è ancora stato calcolato o ottenuto. In altre parole, un Future rappresenta un’operazione asincrona che deve essere completata in futuro. Un Future può essere utilizzato per eseguire operazioni asincrone come, ad esempio, il recupero di dati da un database o la chiamata a un servizio web. Quando si esegue un’operazione asincrona, il valore non è immediatamente disponibile e quindi non si può attendere il risultato direttamente. Invece, si può creare un Future che rappresenta il valore in attesa e che sarà completato in futuro, quando l’operazione asincrona sarà stata eseguita. Il Future<T> restituisce un valore di tipo T, tuttavia, nel caso in cui il future debba eseguire alcune operazioni senza che sia previsto un valore di ritorno è possibile

utilizzare Future<void>. In Dart è possibile concatenare alcune operazioni ai Future alle quali viene passato il risultato di quest'ultimo. Tali operazioni possono essere eseguite concatenando il metodo then, oppure si può usare whenComplete se si vogliono eseguire operazioni senza sfruttare il valore di ritorno del Future. Questo paradigma che concatena una callback ad un'operazione asincrona è molto simile a quello implementato nel JavaScript.

3.1.3 async e await

La parola chiave async deve essere premessa nella signature di un Future o di una funzione che internamente faccia uso di una funzione asincrona. Await invece da la possibilità di attendere che il risultato di un Future sia disponile. Essa non implica un'operazione bloccante, ma semplicemente permette al software di proseguire nel running finché il risultato dell'operazione asincrona non è disponibile, e solo a quel punto di proseguire nelle operazioni concatenate al Future. Di fatto questo meccanismo permette di evitare l'utilizzo di ulteriori callback.

3.1.4 Stream

Gli Stream di Dart sono un meccanismo per la gestione di sequenze di dati asincroni. In particolare, un oggetto Stream rappresenta una sequenza di eventi, che possono essere di qualsiasi tipo, come ad esempio una stringa. Gli Stream sono utili per la gestione di flussi di dati asincroni in situazioni in cui non si conosce il momento in cui arriveranno i dati. Ad esempio, si possono utilizzare gli Stream per ricevere dati da una connessione di rete, oppure per elaborare eventi generati da un'interfaccia utente. Per utilizzare gli Stream in Dart, è possibile creare un oggetto StreamController, che consente di emettere eventi all'interno dello Stream. In particolare, un oggetto StreamController dispone di due metodi principali: add e addError, che permettono di emettere eventi rispettivamente positivi e negativi. Per ascoltare gli eventi all'interno di uno Stream, è possibile utilizzare il metodo listen(), il quale accetta una o più funzioni di callback che verranno chiamate ogni volta che un nuovo evento avviene. Inoltre, è possibile utilizzare operatori come map, where o reduce per manipolare gli eventi all'interno dello Stream e ottenere i dati desiderati. Inoltre, così come i Future await, gli Stream non sono bloccanti. Gli Stream possono essere single subscription, il che permette ad un solo listener di restare in ascolto degli eventi. Questo tipo di Stream è particolarmente indicato se si stanno attendendo dati in seguito ad una chiamata web. Alternativamente, si possono implementare degli Stream broadcast, i quali rendono possibile che più listener restino in ascolto degli eventi, e che ognuno di essi registri proprie callback. Questa seconda modalità è adatta a situazioni in cui si attendono interazioni dell'utente con una possibile interfaccia.

Capitolo 4

Flutter

Flutter è un framework open source sviluppato da Google in grado di creare applicazioni mobile, web e desktop. Esso è di recente sviluppo, essendo stato rilasciato nel Dicembre del 2018, e, come esposto nel già citato survey di statista, dal 2021 è il framework cross-platform più utilizzato avendo superato React Native. Utilizza il linguaggio di programmazione Dart, ed è stato progettato per semplificare il processo di sviluppo, migliorare le performance e fornire un'esperienza utente di alta qualità. Il framework fornisce anche una vasta gamma di widget personalizzabili e preconfezionati, che semplificano la creazione di interfacce utente attraenti e intuitive. Inoltre, Flutter utilizza un sistema di rendering personalizzato, chiamato "Skia", che fornisce prestazioni elevate e permette di creare animazioni fluide e reattive. Flutter fornisce anche un'ampia gamma di strumenti per il testing, il debugging e la profilazione delle applicazioni, tra cui il supporto integrato per i test di unità e di integrazione, il supporto per la rilevazione di bug e la visualizzazione della gerarchia dei widget. Flutter si basa su elementi visuali detti Widget, in grado di modificarsi sulla base della logica definita dallo sviluppatore. Questa soluzione è stata adottata poiché idealmente molto simile a quanto già sperimentato in ambito mobile nativo con Swift per iOS e, più recentemente, con Jetpack Compose, un toolkit per lo sviluppo Android. Flutter è costituito da una struttura a strati così composta [12]:

- Strato framework: il primo strato del framework scritto in Dart. In questo livello sono contenute tutte le librerie necessarie per lo sviluppo delle app mobile, nonché le astrazioni di cui fa uso il framework. In esso possiamo trovare le librerie che contengono i widgets fondamentali, quelle per il rilevamento delle interazioni dell'utente, siano queste tramite mouse o touchscreen, ma anche le librerie Material e Cupertino, che permettono di sviluppare applicazioni esteticamente simili a quelle rispettivamente Android ed iOS native;

- Strato engine: in questo strato si sfrutta il C++ al fine di rendere le applicazioni Flutter molto più reattive e performanti, senza dover ricorre a linguaggi interpretati;
- Strato embedded: nello strato più profondo si interviene sul sistema del device in linguaggio nativo. Questo strato rappresenta il "cuore" della compatibilità con i differenti sistemi operativi.

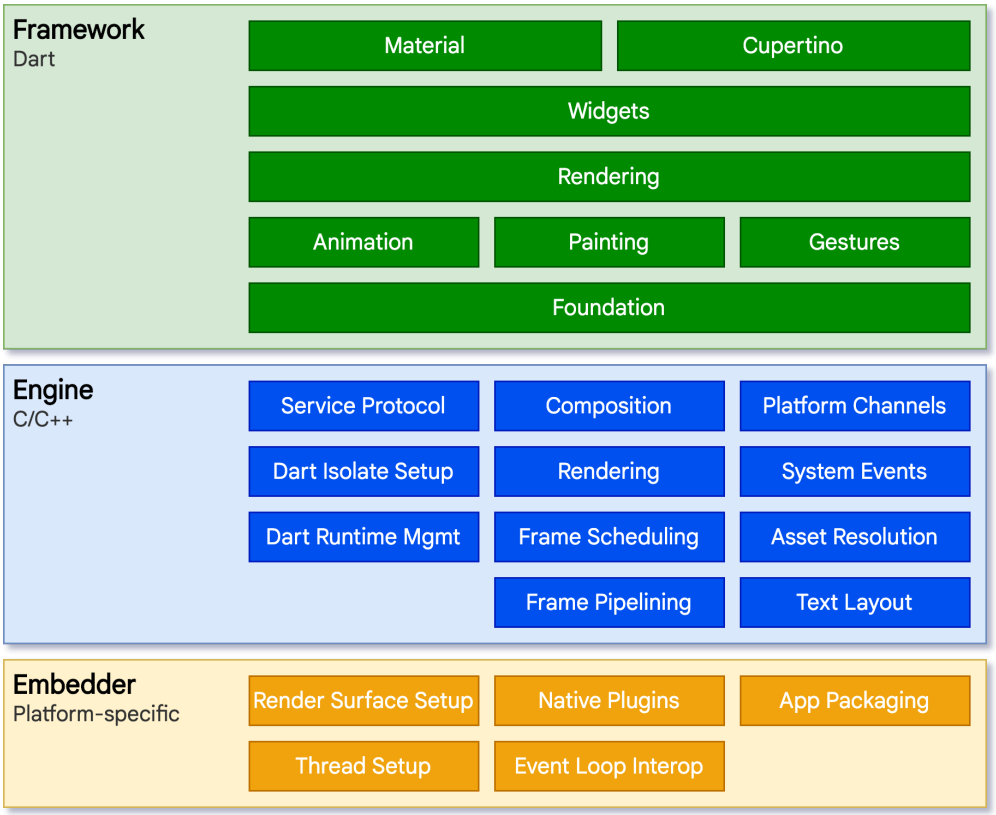

Figura 4.1: Struttura del framework Flutter

4.1 Widget

In Flutter si sfruttano gli Widget [13] per andare a generare qualunque elemento visuale. In Flutter, gli Widget sono i mattoni fondamentali per la creazione di interfacce utente. In altre parole, un Widget è un componente che definisce l'aspetto e il comportamento di una parte dell'interfaccia utente, come ad esempio un bottone, una casella di testo o un'icona. Gli Widget in Flutter possono essere

combinati tra loro, formando alberi di Widget, chiamati "Widget tree". In questo modo, gli sviluppatori possono creare complesse interfacce utente, organizzando i Widget in modo gerarchico e posizionandoli in modo da soddisfare le esigenze dell'applicazione. Inoltre, Flutter fornisce un sistema di layout, il quale permette di definire il posizionamento dei vari componenti. I Widget possono essere di due differenti tipologie:

- Stateful Widget: questa tipologia di Widget può essere modificata una volta istanziata, poiché il suo stato può essere salvato e aggiornato in base alla logica in essi implementata. Essi sono pensati specificatamente per andare a costituire quei punti dell'interfaccia che muteranno a runtime, sia questo dovuto ad un'interazione da parte dell'utente o al modificarsi di un dato in seguito ad una chiamata HTTP. Quando uno stateful Widget deve essere modificato, Flutter rigenera (di fatto renderizza nuovamente l'albero dei Widget), andando a modificare lo stato di quelli che lo necessitano. Gli stateful Widget vengono composti tramite l'utilizzo di due classi, una per istanziare lo Widget vero e proprio che attraverso il metodo create() genera la seconda, con quest'ultima che andrà a rappresentare lo stato. In quest'ultima classe si inserirà il codice per la resa visale del Widget, ma anche la sua logica interna;
- Stateless Widget: questa tipologia di widget viene utilizzata per andare a creare componenti il cui stato non deve essere salvato, poiché non è previsto che essi si modifichino a runtime. In casi di rielaborazione dell'albero dei Widget il framework non dovrà calcolare alcuna modifica su questi Widget, velocizzando il rendering. Essi vengono definiti tramite l'implementazione di un'unica classe.

In Flutter, come anticipato, il codice deputato alla definizione del layout risiede nei Widget, così come quello legato alla logica interna degli stessi. Per questo motivo è possibile inserire tutto il codice relativo ad un elemento visuale in un unico componente, rifiutando dunque il paradigma che vorrebbe il codice deputato ai due compiti in sedi differenti, come avviene in React Native tramite l'utilizzo dei CSS sheet. La maggior parte dei Widget prevede un parametro child nel suo costruttore; esso serve per istanziare un Widget al livello gerarchicamente inferiore di quello attuale. Una volta generato l'albero dei Widget il framework impedisce che un Widget figlio di un altro possa modificarsi, eccezion fatta per l'utilizzo di un operatore ternario che sfrutti un booleano così costituito:

In questo caso al variare del valore di flag otterremo un Widget piuttosto che un altro. SizedBox() rappresenta un'area di spazio vuota, esso dunque non implicherà alcun elemento visivo nell' interfaccia dell'applicazione. Questa soluzione deve essere adottata perché le librerie di Flutter implementano la null-safety, per cui utilizzare la parola chiave null in alternativa ad un Widget risulta in un errore.

```
child: flag ? const Text("Hello World!") : const SizedBox()
```

Figura 4.2: Utilizzo dell'operatore ternario in Flutter

4.1.1 Metodo initState()

Il metodo initState() viene chiamato un'unica volta quando si istanzia la classe che descrive lo stato di uno Stateful Widget. Esso ha lo scopo principale di fornire un valore inziale alle variabili dello stato. Bisogna notare che in Flutter, infatti, non è possibile inizializzare le variabili di stato sulla base del valore di un'altra variabile, se non all'interno di questo metodo. Per inizializzare una variabile all'interno di questo metodo è necessario dichiararla precedentemente aggiungendo la parola chiave "late".

4.1.2 Metodo setState()

Il metodo setState() in Flutter ordina al framework di ridisegnare l'albero delle viste; quest'operazione terrà conto della logica implementativa inserita nel metodo stesso, come ad esempio il cambio di valore di alcune variabili. Va segnalato che Flutter impedisce di chiamare il metodo setState() durante l'operazione di ricostruzione dell'albero delle viste, poiché, in questo caso, il framework interromperebbe il rendering dell'interfaccia restituendo un'eccezione. Esistono altre modalità tramite le quali imporre il rebuild della UI, tuttavia queste verranno trattate più avanti.

4.1.3 Metodo build()

Il metodo build si trova sia negli Stateless che negli Stateful Widgets. Al suo interno è contenuto il codice che definisce la visualizzazione del componente, ma anche la logica che ne definisce il comportamento. Negli Stateful Widget questo metodo viene chiamato ogni volta che si impone un rebuild, ad esempio chiamando il Metodo setState(). Nel caso dei Widget di tipo Stateless invece il metodo viene chiamato solo quando la classe è istanziata.

4.2 GoRoute

GoRoute [14] è un pacchetto che fornisce una soluzione per la navigazione interna all'app. il GoRouter è un widget che permette di gestire differenti Route, ognuna

delle quali rappresenta un'astrazione di una pagina dell'applicazione. Al GoRouter è possibile fornire una route inziale che sarà la pagina mostrata di default all'apertura dell'app. Successivamente è possibile popolare il campo "routes" con un array di oggetti del tipo GoRoute. Come si può notare da questa immagine ogni

```
GoRoute(
  path: '/initialScreen',
    parentNavigatorKey: rootNavigatorKey,
  builder: (context, state) { return WaitingScreen();},
  routes: [
    GoRoute(
      path: 'cameraScreen',
        parentNavigatorKey: rootNavigatorKey,
      builder: (context, state) => CameraScreen(),
      routes: [
        GoRoute(
          path: 'splashScreen',
          parentNavigatorKey: rootNavigatorKey,
          builder: (context, state) => SplashScreen(),
    ),  // GoRoute
  ]
),  // GoRoute
```

Figura 4.3: Esempio di implementazione di GoRoute

Widget GoRoute possiede un path specifico che la identifica. La pagina inziale deve necessariamente avere un path che comincia con "/". Quando una determinata route viene chiamata attraverso il suo path identificativo il Widget GoRoute chiamata la funzione contenuta nel suo campo "builder", la quale restituisce una nuova pagina dell'app. Dall'immagine si può notare come ogni oggetto GoRoute possiede un proprio campo "routes", il quale può essere utilizzato per andare a generare una navigazione gerarchica, così che alcune pagine possano essere raggiunte solo da altre. Per muoversi attraverso le pagine è possibile sfruttare i metodi go() e pop(). Il primo richiede che venga fornito un path valido, che sia dunque esplicitato fra quelli inseriti nelle routes del GoRouter, ma al contempo che sia raggiungibile della pagina corrente sulla base della gerarchia definita. Quando si vuole scendere in profondità nella gerarchia delle pagine è importante notare che il path di una route figlia dovrà comprendere tutte le routes genitrici. Tornando all' esempio in foto, il

path per raggiungere lo SpashScreen() sarà quindi "/initialScreen/cameraScreen/-splashScreen". Il metodo pop() permette di ritornare alla pagina precedente sulla base della gerarchia definita. Il Widget GoRouter permette inoltre di andare a definire una ShellRoute: attraverso questa soluzione è possibile andare ad inserire la navigazione fra pagine internamente ad un altro componente. Questa soluzione si rivela particolarmente utile quando si mira a creare una navigazione fra pagine contenute fra un appBar inferiore ed una StatusBar superiore, con queste due che non mutano durante il routing. Così facendo alcuni componenti della schermata restano fissi, diminuendo il peso della costruzione dell'interfaccia in seguito ad una richiesta di re-indirizzamento verso una pagina differente da quella corrente. In quest'immagine si può notare come il "builder" della ShellRoute possieda il

Figura 4.4: Esempio di implementazione di ShellRoute

parametro "child", che rappresenta proprio la pagina corrente. Quest'ultimo viene passato ad un componente custom "MainScreenContent" che internamente si occupa di inserire le pagine fra una barra di navigazione inferiore ed una statusBar superiore. Tratterò più avanti l'implementazione di tale componente. Da notare che la SheelRoute richiede una "navigatorKey", ossia un identificativo, dedicata e differente da quella usata nella navigazione normale.

4.3 Principali widget utilizzati nel caso di studio

In Flutter esiste una pletora di Widget precostruiti per svolgere i compiti più disparati. Questi possono poi essre combinati dagli sviluppatori al fine di creare nuovi Widget personalizzati. Di seguito esporrò i principali widget utilizzati per la realizzazione estetica del progetto in esame.

4.3.1 Scaffold

In Flutter uno Scaffold è un componente che si occupa di andare a creare lo scheletro dell'interfaccia utente. All'interno del suo costruttore esso accetta numerosi parametri, ognuno dei quali deve essere uno widget che costituirà un elemento tipicamente presente in una mobile app. Tra questi possiamo trovare:

- appBar: questo parametro andrà a costruire l'AppBar superiore dell'applicazione;

- body: il body è il cuore dello scaffold. Esso costituisce la parte centrale dello schermo, per cui ciò con cui gli utenti interagiranno per la maggior parte del tempo. Usualmente, così come è stato fatto per lo sviluppo dell'app oggetto di questa tesi, a questo parametro si fa corrispondere il "child" di una ShellRoute;

- bottomNavigationBar: a questo parametro si fa corrispondere il widget che si occuperà di andare a disegnare la barra di navigazione inferiore dell'applicazione;

- floatingActionButton: attraverso questo parametro è possibile inserire un botton in primo piano in una qualsiasi posizione dell'interfaccia.

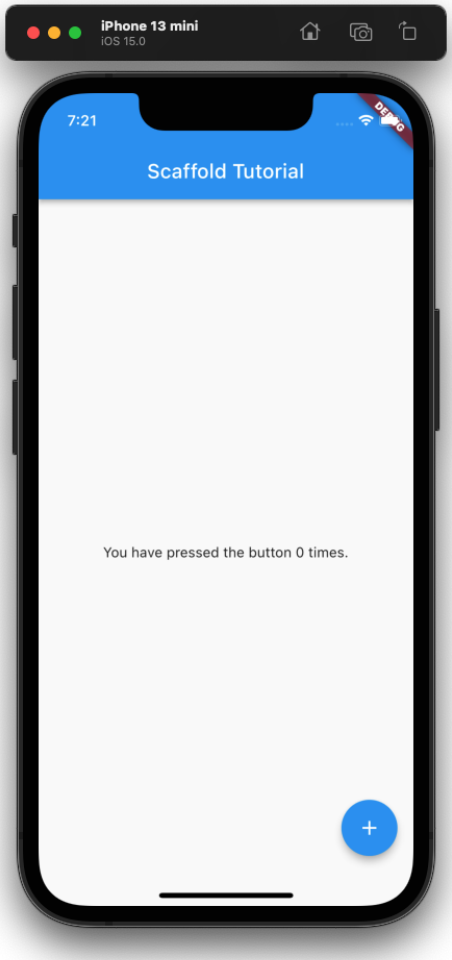

Figura 4.5: Resa visiva di uno Scaffold

4.3.2 BottomNavigationBar

Questo Widget permette di creare rapidamente una barra di navigazione inferiore. Esso accetta nel suo costruttore una serie di parametri tra i quali:

- items: questo parametro altro non è che un array di altri Widget, ossia dei BottomNavigationItem, ognuno dei quali rappresenta una singola icona della barra di navigazione;

- onTap: qui è possibile passare una funzione che possieda internamente la logica per andare a definire il routing. Questa funzione verrà chiamata quando uno qualsiasi degli items verrà cliccato;
- currentIndex: a questo parametro è possibile passare una funzione che verrà chiamata in seguito ad onTap; essa si occuperà di andare a definire qual è l'icona rappresentante la pagina corrente;
- unselected/selectedItemColor: qui si possono definire i differenti colori degli items, intesi come icone e/o scritte, se rappresentanti la pagina corrente o meno;
- backgroundColor: qui si può definire il colore dello sfondo della navBar.

4.3.3 Rows e columns

Una Row è una Widget che mostra i suoi figli in un array orizzontale. Non è un Widget in grado di scorrere lateralmente, di conseguenza si verificherà un errore nel caso gli si assegnino più Widget figli di quelli che può contenere. Nel caso in cui un Widget figlio debba espandersi per tutto lo spazio orizzontale disponibile è possibile inserirlo all'interno del Widget Expanded. Una Column è un' Widget che mostra i suoi figli in un array verticale. Esattamente come per le Row, anche le Column non scorrono e di conseguenza non è possibile aggiungere un numero troppo grande di Widget figli.

4.3.4 Stack e Positioned

Uno Stack è un Widget che occupa tutto lo spazio disponibile. La sua particolarità sta nel fatto che accetta un array di Widget come figli e li posiziona uno sopra l'altro a partire dal primo. Questo Widget è logicamente molto simile al FrameLayout presente nelle applicazioni native Android. Un Positioned è invece un Widget che si occupa di andare a definire la posizione di un Widget figlio. il Positioned accetta nel suo costruttore 4 distanze che andranno a definire l'offset (destra, sinistra, alto e basso) da applicare per il posizionamento a partire da un punto iniziale, che di default sarà l'angolo in alto a sinistra del Widget padre, a meno che quest'ultimo non imponga diversamente. Si può specificare anche solo un offset fra alto e basso e destra e sinistra, purché si specifichino altezza e larghezza del Widget.

4.3.5 Listener e GestureDetector

Flutter fornisce dei Widget specifici per andare a gestire le interazioni dell'utente con l'interfaccia. Il Listener è un Widget che si occupa di andare a monitorare

un'interazione generica con la UI, sia questa effettuta tramite touchscreen oppure tramite mouse. Esso possiede i seguenti metodi principali:

- onPointerDown: questo parametro accetta una callback che verrà eseguita quando il puntatore entra in contatto con l'interfaccia, di conseguenza questa callback è specificatamente pensata per un utilizzo tramite mobile device;
- onPointerUp: qui è possibile definire un'ulteriore callback per quando l'nitera-zione con touchscreen termina;
- onPointerMove: questa funzione verrà chiamata ogni qual volta il puntatore si muoverà, sia esso il dito che scorre sul touch screen oppure il mouse a schermo. Essa passa alla funzione un pointerMoveEvent che contiene una serie di parametri che descrivono il tipo di interazione, tra cui l'entità del movimento del puntatore nelle quattro direzioni dalla chiamata precedente a questa callback. Sulla base di questi dati è possibile comprendere, ad esempio, la direzione di scorrimento del dito sul touchscreen.

In alternativa è possibile utilizzare un GestureDetector, ossia un Widget specificatamente pensato per comprendere le gesture eseguite dall'utente, e fornire la possibilità di reagire con una callback dedicata ad ognuna. Mediante l'utilizzo di GestureDetector è possibile gestire molti eventi. Le principali categorie sono Tap, Double Tap, Long Pressed, Pan e Drag orizzontale e verticale. Gli eventi onTap identificano il tocco dell'utente sullo schermo e possono essere:

- onTapDown: identifica il punto di contatto fra il dito dell'utente e lo schermo;
- onTapUp: serve per registrare l'istante in cui il tocco termina;
- onTap: qui è possibile registrare una callback che verrà chiamata ad ogni interazione dell'utente con il touchscreen.

L'evento onDoubleTap identifica un doppio tocco in rapida successione sulla stessa zona dello schermo da parte dell'utente. L'evento onLongPressed identifica un tocco prolungato dell'utente. Gli eventi VerticalDrag e HorizontalDrag identificano un trascinamento dello schermo da parte dell'utente nelle differenti direzioni.

4.3.6 Transform e AnimatedBuilder

Essendo l'oggetto di questa tesi un serious game, le animazioni hanno giocato un ruolo importante nella creazione della UI. Dovendo per questo motivo sviluppare animazioni complesse si è fatto largo uso dei Widget a questo preposti. Flutter fornisce una serie di Widget deputati alla definizione di animazioni ed effetti visivi di ogni sorta. Il Widget Transform, fortemente usato nel serious game oggetto

di questa tesi, permette di andare ad applicare ad un Widget figlio una certa trasformazione, la quale deve essere definita come una matrice 4x4 comprendente i coefficienti relativi a rotazione, traslazione e scalamento lungo i 3 assi x,y e z. Il fatto che la matrice comprenda i coefficienti dell'asse z implica che questo Widget sia in grado di restituire trasformazioni tridimensionali degli oggetti. La Transform accetta, inoltre, un punto di origine, definito come offset bidimensionale a partire dal punto in alto a sinistra del Widget figlio, dal quale applicare la matrice di trasformazione. Questo Widget si rivela molto potente per la composizione di animazioni complesse se abbinato ad un AnimatedBuilder. In Flutter esistono Widget che si occupano specificatamente di una singola tipologia di animazione, come ad esempio la traslazione su un unico asse effettuabile tramite il Widget SlideTransition; tuttavia, ognuno di questi Widget implementa internamente il Widget AnimatedBuilder, poiché questo consente di generare animazioni personalizzate. Esso accetta una callback definita "builder", che restituisce un Widget, ed un'animazione. Al progredire dell'animazione il "builder" viene chiamato nuovamente permettendo di restituire un Widget il cui stato può essere modificato in base all'avanzamento dell'animazione. In Flutter un'animazione è sempre definita come un'interpolazione fra uno valore matematico inziale ed uno finale. Ogni animazione richiede un AnimationController, ossia un componente che si occupa di far progredire l'animazione dallo stato iniziale a quello finale in un dato tempo. L'AnimationController può determinare l'inizio dell'animazione attraverso il suo metodo forward(). Mescolando tutte le nozioni qui brevemente esposte possiamo comprendere le potenzialità espresse dalla combinazione di questi Widget. Data infatti un'animazione definita come interpolazione fra una matrice di trasformazione inziale ed una finale, è possibile passare quest'animazione all'AnimatedBuilder, facendo in modo che la callback "builder" si occupi di generare un Widget Transform con il valore corrente dell'interpolazione fra le matrici inizialmente definite. Con questa tattica è possibile generare animazioni complesse, che verranno applicate a qualsiasi Widget si ponga come figlio della Transform. Più avanti esporrò come questa soluzione sia stata sfruttata nel serious game.

4.4 pubspec.yaml

Il file pubspec.yaml è un file di configurazione richiesto per ogni progetto Flutter e viene utilizzato per specificare le dipendenze, le risorse utilizzate e altre informazioni importanti. La sezione dependencies è utilizzata per specificare le dipendenze. In questa sezione deve essere necessariamente presente il framework Flutter. La sezione dev-dependencies è utilizzata per specificare le dipendenze di sviluppo, come le librerie di test. Infine, la sezione flutter viene utilizzata per specificare le impostazioni specifiche del framework. Per esempio, se "uses-material-design" è

impostato su true, ciò indica che il progetto utilizza un design ispirato a quello Android nativo. Il file pubspec.yaml viene utilizzato anche per specificare altre informazioni importanti, come l'autore, la licenza dell'app e le risorse come immagini, font e altro ancora.

```yaml
dependencies:
  flutter:
    sdk: flutter


  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
  indexed: ^0.0.8
  intl: any
  provider: ^5.0.0
  firebase_core: ^2.3.0
  firebase_database: 10.0.6
  qr_flutter: ^4.0.0
  qr_code_scanner: ^1.0.1
  firebase_auth: ^4.2.0
  go_router: ^5.2.3
  collection: 1.16.0
  google_fonts: ^2.1.0
  fluttericon: 2.0.0
  lottie: ^1.2.1
  page_transition: "^2.0.9"
  flutter_shake_animated: ^0.0.5
```

Figura 4.6: pubspec.yaml del caso di studio

A titolo esemplificativo esporrò di seguito il pubspec.yaml del caso di studio oggetto di questa tesi, descrivendo brevemente le dependencies utilizzate:

- cupertino-icons: dependencie che permette di utilizzare la libreria di icone native delle app iOS;

- indexed: un pacchetto garantito da PubDev, ossia un archivio verificato dal team di sviluppo Flutter di librerie di terze parti. Esso permette di ottenere una versione del Widget Stack, già precedentemente presentato, con la particolarità che l'ordine con cui i Widget figli vengono disposti gli uni sopra gli altri può essere modificato tramite un indice. L'implementazione di questo Widget verrà sviscerata in seguito;

- Provider: questa dependencie permette di sfruttare un Data Provider che costituisce l'architrave della programmazione reattiva in-app. Il funzionamento del provider verrà esposto descrivendo il GameModel in seguito;

- firebase-core, firebase-database, firebase-auth: la combinazione di questi tre pacchetti permette di accedere alle funzionalità di Firebase, oggetto del capitolo successivo;

- qr-flutter: un pacchetto che permette di andare a generare un QR code a partire da qualsiasi tipo di dato. Esso è stato utilizzato lato game master per andare a generare il QR code contenente i dati che identificano la partita sul database;
- qr-code-scanner: questa dependencie permette di effettuare lo scan di barcode e qrcode tramite la camera del device. Il pacchetto permette di estrapolare i dati necessari, ma consente anche di andare a definire un'interfaccia basilare, comprendente un box contente quanto sta inquadrando la camera, ma anche altri elementi di UI, come delle guide visive per indicare l'area in cui far ricadere il QR code per facilitare lo scan. Tale libreria verrà utilizzata dai giocatori per accedere alla partita;
- go-router: qui si implementa la libreria che permette di utilizzare il Widget GoRouter, già descritto in precedenza;
- google-fonts: permette di utilizzare in app i font proprietari di Google;
- fluttericon: da accesso al set di icone standard per le Flutter app;
- lottie: libreria sviluppsta per Flutter dal team di LottieFiles, ossia un provider di grafiche 2D, animate o meno, pubblicate con licenza cc-by. Questa libreria è stata utilizzata per poter inserire elementi di motion graphics all' interno dell'applicazione, previo scaricamento degli stessi dal sito di LottieFiles;
- page-transition e flutter-shake-animated: librerie che danno accesso a Widget deputati alla realizzazione di specifiche animazioni.

Capitolo 5

Firebase

Firebase [15] è una piattaforma Mobile backend as a service (MBaaS) che permette di sfruttare un cloud backend per la creazione di app web e mobile. Firebase è stata creata da Firebase, Inc. nel 2011, ma nel 2014 la società è stata acquisita da Google e da allora è stata sviluppata e gestita da Google. La piattaforma Firebase è stata originariamente lanciata come un'API di sincronizzazione e archiviazione dei dati in tempo reale, nota come Firebase Realtime Database. Nel corso degli anni, la piattaforma è stata espansa per includere una vasta gamma di funzionalità, tra cui l'autenticazione degli utenti, il cloud messaging, l'hosting web, l'analisi degli utenti e molto altro ancora.

5.1 Servizi Firebase

Uno dei fattori chiave del successo di Firebase è la sua facilità d'uso. La piattaforma è stata progettata per semplificare il processo di sviluppo e fornire agli sviluppatori gli strumenti necessari per creare app di alta qualità in modo efficiente. Grazie alla sua scalabilità, i servizi offerti da Firebase sono adatti sia a piccoli progetti che a grandi applicazioni con un alto traffico di utenti. Firebase è diventata una delle piattaforme di sviluppo più popolari per le app mobile e web grazie alla sua vasta gamma di funzionalità e alla sua facilità d'uso. Google continua a sviluppare e aggiornare la piattaforma Firebase per fornire agli sviluppatori le migliori soluzioni per la creazione di app. Firebase è una piattaforma per lo sviluppo di applicazioni web e mobili che non ha bisogno di un linguaggio di programmazione server side. Inoltre, la piattaforma è pensata per garantire la sincronizzazione automatica dei dati, questo vuol dire che le modifiche apportate al database mentre un client non sta utilizzando l'app saranno visibili al successivo accesso attraverso un aggiornamento istantaneo. Firebase è particolarmente vantaggioso anche in virtù dell'esistenza di librerie dedicate per integrare la piattaforma in app Web,

Android native, iOS native ed anche cross-piattaforma. Per il progetto oggetto di questa tesi è stata utilizzata una libreria dedicata per Flutter. Infine, Firebase fornisce piani con costi differenti sulla base della mole del traffico, con un limite minimo di quest'ultimo al di sotto del quale la piattaforma è gratuita. Firebase offre una vasta gamma di servizi differenti, ognuno dei quali mira a svolgere un compito che i servizi backend tipicamente includono. Questi servizi possono essere utilizzati singolarmente, il che rende molto facile sfruttare Firebase solo per quelle che sono le effettive necessità, potendo embeddare il servizio della piattaforma nei propri progetti front-end. L'idea alla base dei vari servizi Firebase è quella di minimizzare il lavoro backend per permettere agli sviluppatori di concentrarsi sul lavoro client-side. I servizi offerti dalla piattaforma sono più di 20, tra cui:

- Firebase Auth: consente agli utenti di autenticarsi tramite email, password, social media o account Google;
- Realtime Database: un database cloud in tempo reale che consente di sincronizzare i dati in tempo reale tra le applicazioni;
- Firebase Storage: uno spazio di archiviazione cloud per immagini, video e altri file multimediali;
- Firebase Hosting: una piattaforma di hosting web per l'hosting di applicazioni web;
- Firebase Cloud Messaging: una soluzione di messaggistica push per inviare messaggi ai dispositivi mobili;
- Firebase Crashlytics: un servizio di monitoraggio delle prestazioni dell'applicazione per risolvere i problemi di stabilità;
- Firebase Analytics: una soluzione di analisi delle prestazioni dell'applicazione che fornisce dati sulla base degli utenti, la loro interazione con l'applicazione e le metriche di marketing;
- Firebase Test Lab: uno strumento di test automatico che aiuta gli sviluppatori a verificare la funzionalità delle loro applicazioni su una vasta gamma di dispositivi e configurazioni;
- Cloud Functions: un servizio che permette di eseguire script JavaScript dedicati quando avvengono modifiche sul Database;
- Cloud Firestore Database: Database NoSQL basato su documenti con il supporto della sincronizzazione in tempo reale.

Questi sono solo alcuni dei servizi offerti da Firebase. La piattaforma è in costante evoluzione e nuovi servizi vengono regolarmente aggiunti per soddisfare le esigenze degli sviluppatori.

5.2 Firebase Auth

Il servizio Firebase Auth permette di effettuare l'autenticazione degli utenti che accedono al database. Tale autenticazione può essere effettuate tramite credenziali quali password e username, oppure tramite l'appoggio ad alcuni degli identity provider più comuni, come Facebook, Google, Twitter e GitHub. Va detto che Firebase fornisce anche la possibilità di effettuare un'autenticazione anonima. Nel caso in cui si scelga questa opzione Firebase si occuperà di generare un codice univoco che identifica il device da cui si effettua il collegamento. Lato client, attraverso le librerie Firebase dedicate, è possibile accedere al token di autenticazione, il quale comprende tutti i dati relativi al processo di autenticazione, comprese le credenziali utilizzate per l'accesso. Nel caso in cui si utilizzi un'autenticazione anonima allora attraverso l'oggetto "current user" del token di autenticazione è possibile accedere all'id generato da Firebase. Nel caso di studio si è scelto di sfruttare un'autenticazione anonima, che ben si prestava all'utilizzo nel serious game, poiché in questo prodotto non si è pensato di inserire dati relativi al profilo dell'utente, ma esclusivamente alla partita corrente. Per queste motivazioni non si è percepita, dunque, la necessità di implementare un sistema di protezione dei dati con password. RealTime Database e Firestore

RealTime Database è un servizio cloud che fornisce l'accesso ad un database NoSQL. In esso i dati sono memorizzati come un unico file JSON. Il database si compone attraverso una struttura gerarchica con un'unica radice rappresentata da un URL, questo rende molto semplice effettuare le chiamate lato client poiché facendo riferimento all' URL si accede alla root del database. Lo schema dei dati immagazzinati all'interno di questo unico file JSON può modificarsi nel tempo, permettendo al client di scrivere dati sul cloud. ll database in tempo reale di Firebase sincronizza i dati tra le applicazioni client e il database in tempo reale in modo efficiente e affidabile. Ciò significa che quando si aggiungono, modificano o eliminano dati, gli utenti dell'applicazione vedono immediatamente il cambiamento, senza la necessità di aggiornare manualmente la pagina o l'app. Cloud Firestore è un cloud database NoSQL basato sul raggruppamento di dati all'interno di documenti differenti, i quali possono essere raggruppati in collezioni. Anche la struttura di Firestore può essere modificata nel tempo. I singoli documenti in Firestore possono contenere dati di differente natura, come stringhe, numeri e date, ma anche oggetti più complessi come array oppure coordinate di geolocalizzazione. Questi documenti sono archiviati in raccolte, chiamate collezioni che contengono i documenti, ma e anche possibile creare sub-collezioni all'interno dei documenti e creare strutture gerarchiche di dati che scalano man mano che il database cresce. Gli unici limiti imposti da Firestore sono: la dimensione di un singolo documento che e di un megabyte e un massimo di cento collezioni annidate. Inoltre, Firestore Database offre funzionalità avanzate di query, che consentono di recuperare

dati in modo efficiente e flessibile. Le query supportano operatori di confronto, ordinamento, raggruppamento e filtraggio, consentendo agli sviluppatori di eseguire query complesse sui dati. Firestore Database offre anche funzionalità avanzate di indicizzazione, che consentono di creare indici personalizzati per migliorare le prestazioni delle query. Inoltre, il servizio offre funzionalità di sincronizzazione dei dati in tempo reale, che consentono di aggiornare automaticamente i dati quando cambiano. Per il caso di studio si è scelto di utilizzare Firebase RealTime Database per via della sua facilità implementativa; inoltre, la mole e la complessità della struttura dati era adatta ad essere immagazzinata in un unico documento JSON.

5.3 Il JSON

JSON [16], acronimo di JavaScript Object Notation, è un formato di scambio dati leggero, utilizzato per rappresentare oggetti e dati strutturati in modo semplice e leggibile. Un documento JSON consiste di un insieme di coppie chiave-valore, dove le chiavi sono stringhe e i valori possono essere di diversi tipi, tra cui stringhe, numeri, booleani, array e altri oggetti. La sintassi di un documento JSON è molto simile a quella degli oggetti JavaScript, il che lo rende facile da leggere e scrivere per gli sviluppatori. JSON è un formato di dati molto popolare e ampiamente utilizzato per lo scambio di dati tra applicazioni web e mobili e i server. Molte API web espongono i dati in formato JSON, rendendoli facilmente consumabili da applicazioni client. Esso rappresenta una valida alternativa al formato XML-XSLT, tanto che oggi numerosi Web Services mettono a disposizione entrambe le soluzioni. Uno dei fattori di successo del JSON è la facilità con cui è possibile interpretare uno stream di documenti in questo formato, non unicamente in JavaScript, linguaggio da cui deriva, ma anche con gli altri linguaggi server-side più comuni come Java e Kotlin.

5.4 Differenze fra database relazionali e non relazionali

Un database relazionale e un database non relazionale (o NoSQL) sono due tipi di sistemi di gestione dei dati con caratteristiche e funzionalità diverse. Un database relazionale organizza i dati in tabelle con righe e colonne, dove ogni riga rappresenta un record e ogni colonna rappresenta un campo di dati. Un database non relazionale, invece, organizza i dati in vari modi, come documenti, grafi o coppie chiave-valore. Nei database NoSQL, non essendo presenti le relazioni, i collegamenti fra informazioni devono essere strutturati in altro modo. L'opzione più facile è inserire un documento, ad esempio un oggetto JSON, all'interno di un altro. Un'altra possibilità è quella di inserire il riferimento ad un documento all'interno

di un secondo. Questa seconda opzione rende molto facile costruire relazioni uno a molti. Nel caso degli oggetti JSON, essi sono identificati da un campo id, il quale si presta ad essere l'informazione con cui creare il riferimento in un altro documento JSON. I database relazionali sono ottimizzati per la lettura di grandi quantità di dati e supportano una vasta gamma di operazioni di query complesse. I database non relazionali, d'altra parte, sono progettati per il recupero rapido dei dati e supportano operazioni di query più semplici. I database relazionali sono stati in passato il principale modello di database utilizzato per lo sviluppo di applicazioni. Tuttavia, esistono alcuni limiti dei database relazionali che hanno spinto allo sviluppo di sistemi di database alternativi. Alcuni di questi limiti sono:

- Scalabilità: i database relazionali sono limitati in termini di scalabilità orizzontale, ovvero la capacità di aumentare le prestazioni aggiungendo hardware aggiuntivo. Ciò significa che a lungo andare, i database relazionali possono diventare troppo grandi per essere gestiti in modo efficiente da un singolo server;

- Complessità della gestione del database: i database relazionali richiedono uno schema rigido per la struttura dei dati, che deve essere definito in anticipo. Ciò può rendere la gestione del database più complessa e limitare la flessibilità nell'aggiunta o modifica dei dati;

- Prestazioni delle query: le operazioni di join tra tabelle in un database relazionale possono richiedere molto tempo e risorse di elaborazione, soprattutto quando il database diventa grande. Ciò può influire sulle prestazioni del database e rallentare le query;

- Gestione di dati non strutturati: i database relazionali sono progettati per gestire dati strutturati in modo uniforme, ma non sono ottimizzati per gestire dati non strutturati come immagini, audio o video;

- Costi: i database relazionali possono essere costosi da implementare e gestire, soprattutto se si utilizzano soluzioni proprietarie.

In generale, i database relazionali sono ancora molto utilizzati e sono adatti per molte applicazioni aziendali. Tuttavia, per alcune applicazioni in cui la scalabilità e la flessibilità sono particolarmente importanti, possono essere preferibili soluzioni alternative come i database non relazionali. Questi ultimi si configurano come la soluzione ideale in situazioni in cui:

- La struttura dei dati è difficilmente predicibile a priori. In particolare, questo il caso in cui i client contribuiscono in modo importante al popolamento del database;

- I client devono effettuare molto spesso accessi al database per estrarne ogni volta moli di dati limitate;
- Si vuole prediligere la performance dell'interazione client-server alla robustezza di un sistema ACID.

Essendo il caso di studio un applicazioni in cui i giocatori sarebbero stati chiamati a popolare il database sulla base delle loro azioni, e dovendo l'app modificarsi rapidamente in seguito alle azioni svolte dagli altri giocatori, i vantaggi di un database NoSQL si sposavano perfettamente con le necessità progettuali.

5.5 Progettazione del database per il caso di studio

Di seguito andrò ad esporre la struttura ideata per le specifiche esigenze del caso di studio. In aggiunta, indicherò brevemente come i dati sono stati utilizzati in app. La root fondamentale del database possiederà tre figlii: masters, password e matches. Concentriamoci dapprima sui primi due. Quando un nuovo device accede all'app può dichiarare di essere un giocatore oppure il game master. Nel secondo caso verrà richiesto l'inserimento di una password, ond'evitare che i giocatori possano accedere all'area game master, deputata agli operatori dell'azienda committente. Inserita la password, la logica in app si occuperà di controllare che corrisponda con quella salvata sul database, nel qual caso si procederà ad effettuare l'autenticazione anonima del device tramite Firebase, salvando successivamente l'UID generato nella sezione "masters" del database. Attraverso questo processo alla successiva riconnessione del device, se questo si dovesse dichiarare come game master, l'app si occuperebbe di controllare che effettivamente l'UID figuri fra quelli salvati in database nella sezione relativa, senza richiedere ulteriormente la password. Passiamo ora alla sezione "matches", ossia il vero core dei dati salvati in cloud. Quando un game master crea una nuova partita per la prima volta verrà generato un figlio della sezione "masters", il quale sarà l'UID affidato da Firebase al device sotto forma di stringa. Le singole partite create dallo stesso device verranno inserite come figlie dell'UID e identificate dall'epoch timestamp d'inizio partita. Quando il game master genera il QR code, in modo tale da permettere l'accesso alla partita, esso conterrà di fatto un array contenente due stringe: l'UID del game master e il timestamp identificativo della partita. Il timestamp si rivela utile anche nel caso in cui un giocatore erroneamente esca dall'app nel corso di una partita, poiché in questo caso riaprendo l'applicazione questa reindirizzerà il giocatore verso l'ultima partita generata e non ancora terminata, qualora vi fosse.
I figli della partita costituiscono tutti i dati che contribuiscono a definirne lo stato, quali:

- level: qui troviamo il numero del livello corrente, il contesto casuale definito per il livello ed infine lo stato del livello, con quest'ultimo che può essere "preparing", ossia relativo alla fase in cui si stanno dando le carte ai giocatori oppure si sta dando tempo per completare il tutorial iniziale, "play", che indica la fase di gioco, ed "ended" che definisce il termine della partita, e per tanto è uno stato accessibile solo all'ultimo livello.

- masterTutorialDone: questo boolean serve come flag per indicare il completamento del tutorial lato game master, che si occuperà di illustrare l'interfaccia di gioco proiettata sulle lavagne interattive delle classi. Così come per i giocatori, qualora il master uscisse per errore dall'applicazione, riaprendola avrà accesso all'ultima partita generata dal master stesso e non terminata. Il flag si rende necessario per evitare il ripetersi del tutorial alla riapertura se questo è già stato mostrato.

- players: all'interno di questo campo verranno inseriti i giocatori partecipanti alla partita. Quando un utente inquadra il qr code generato dal game master l'app si occupa di inserire il suo Firebase UID come figlio di questo campo. Quando il master darà inizio alla partita per ogni player si creeranno 3 campi: "ownedCards", contenente un codice univoco per ogni carta posseduta dal giocatore; "team" contenente il codice identificativo della squadra in cui il giocatore è inserito; "tutorialDone", ossia un ulteriore flag che indica il completamento o meno del tutorial che presenta l'interfaccia dell'app lato giocatore. Quest'ultimo booleano è necessario per le stesse ragioni del campo "masterTutorialDone".

- teams: all'interno di questo campo si inseriscono tutti i dati relativi alle singole squadre. Per ogni squadra si popolano i seguenti campi: "ableToPlay", che va ad indicare il giocatore corrente attraverso il suo UID. Quando un giocatore termina il suo turno cancella il proprio UID, lasciando questo campo vuoto. Il game master viene notificato da Firebase quando il campo è senza valore, e si occupa di inserire l'UID del giocatore successivo per la squadra. "drawableCards" che contiene i codici identificativi delle carte che possono essere pescate. Il meccanismo di pescata casuale è già stato esposto nel capitolo sul game design. "objective", un campo utilizzato per indicare l'obbiettivo affidato alla squadra per il livello corrente. "playedCards" rappresenta il campo fondamentale, poiché qui i giocatori potranno inserire il frutto delle loro giocate. I campi figli di "playedCards" si compongono di coppie chiave-valore costituite dal mese in cui è stata giocata una carta, ed il codice della carta stessa. "points" determina i punteggi parziali o finali della squadra. Al termine di ogni livello, infatti, il game master calcola il punteggio ottenuto

dalle singole squadre per stilare una classifica parziale, e, al contempo, inserisce i punti accumulati nel database.

Figura 5.1: Struttura del database per il caso di studio

5.6 Regole Firebase

In Firebase, le "regole" si riferiscono alle regole di sicurezza che determinano chi può accedere ai dati memorizzati nel database Firebase, e con quali modalità e possibile accedere agli stessi. Le regole sono definite utilizzando il linguaggio di sintassi Firebase Realtime Database Security Rules, che consente di specificare le condizioni di accesso ai dati in base alle richieste provenienti dai client che utilizzano l'applicazione. Le regole di sicurezza sono fondamentali per proteggere i dati dell'applicazione e garantire che solo gli utenti autorizzati possano accedere e modificare i dati. Le regole di Firebase possono essere personalizzate in base alle esigenze dell'applicazione e possono essere utilizzate per controllare l'accesso ai dati in modo granulare, ad esempio in base all'identità dell'utente o al percorso dei dati nel database. Le regole in Firebase consentono di definire condizioni di accesso ai dati del database in base a vari fattori, tra cui l'identità dell'utente, il percorso dei dati e le autorizzazioni relative all'utente che effettua la richiesta. Di seguito sono riportate le regole che è possibile definire in Firebase RealTime Database:

- “read" e "write": queste regole consentono di controllare l'accesso in lettura e scrittura ai dati del database. È possibile definire queste regole in base all'identità dell'utente, al percorso dei dati e alle autorizzazioni dell'utente;
- "validate": questa regola consente di controllare la validità dei dati che vengono scritti nel database. Ad esempio, è possibile definire una regola "validate" che controlli se il valore di un campo è un intero o una stringa;
- "indexOn": questa regola consente di indicizzare i dati in base a un determinato campo. L'indicizzazione dei dati può migliorare le prestazioni delle query sui dati del database;
- "allow": questa regola consente di definire autorizzazioni personalizzate per l'accesso ai dati del database. Ad esempio, è possibile definire una regola "allow" che consente l'accesso ai dati solo agli utenti che hanno un determinato ruolo o permesso.

Firestore consente di definire regole più articolate rispetto a RealTime Database, aggiungendo altre parole chiave con cui comporle. Queste sono solo alcune delle regole che è possibile definire in Firebase. Le regole possono essere personalizzate in base alle esigenze dell'applicazione per garantire la sicurezza dei dati e il controllo dell'accesso. A titolo esemplificativo presenterò una sezione delle regole definite per il caso di studio.

La sezione qui rappresentata fa riferimento alle regole definite per la sezione “teams” del database. Le regole in Firebase funzionano in modo tale che i figli possano restringere i permessi garantiti dai rami padri, ma non possano in alcun

```
  ''
"teams": {
          ".read" : "auth.uid!== null",
          ".write" : "auth.uid !==null",
         "$teamCode": {
                        "ableToPlay": {
                                    "$playerCode": {
                                        ".read" :  "auth.uid!== null",
                                        ".write" : "auth.uid === $playerCode || auth.uid === $masterUid",
                                        ".validate" : "!data.exists()",
                                    },
                        },
                         "playedCards": {
                                      "$cardCode": {
                                        ".read" :  "auth.uid!== null",
                                        ".write" : "auth.uid !== null"
                                      },
                           "drawableCards": {
                             "$cardCode": {
                           ".read" :  "auth.uid!== null",
                           ".write" : "auth.uid !== null"
                         }},
                           "objective": {

                                        ".read" :  "auth.uid!== null",
                                        ".write" : "auth.uid === $masterUid"
                                      },
                         "points" : {
                            ".read" :  "auth.uid!== null",
                            ".write" : "auth.uid === $masterUid"
                         }
                        },

         }
  },
```

Figura 5.2: Struttura delle regole per il caso di studio

modo allargarli. Come sui può vedere in foto per il ramo “teams”, i dati possono essere letti e scritti, purché il device che richiede di effettuare l’operazione abbia effettuato l’autenticazione anonima tramite FIrebase. Guardando invece al ramo figlio “ableToPlay” possiamo notare l’utilizzo delle variabili dinamiche, identificate dal simbolo "$". Le Firebase rules permetto di andare a definire un nome identificativo per un campo del quale non si conosca preventivamente il nome effettivo. L’utilità di queste variabili dinamiche sta nel fatto che si potranno riutilizzare gli identificativi arbitrari per definire regole che, una volta popolato il database, si applicheranno ai dati effettivi. Va sottolineato che le variabili dinamiche fanno si che il valore di “$playerCode” vari ogni qual volta il valore di “ableToPlay” sia modificato. Come si può notare il player UID, il cui valore è impredicibile a priori, viene identificato come “$playerCode”, il quale sarà leggibile da chiunque, purchè identificato, ma scrivibile unicamente dal player stesso o dal game master (altra variabile dinamica precedentemente definita), inoltre Firebase accetterà la scrittura del dato solo se il campo sarà vuoto nell’istante in cui si richiede la scrittura.

Capitolo 6

Caso studio

6.1 Modello MVVM

Il Model-View-ViewModel (MVVM) [17] è un pattern architetturale utilizzato nel contesto dello sviluppo di applicazioni software. In questo modello, l'interfaccia utente dell'applicazione è suddivisa in tre parti principali:

- Model: Il Model rappresenta la parte dell'applicazione che gestisce i dati e la logica di business dell'applicazione. In altre parole, il Model va a rappresentare il modello dei dati dell'applicazione. Il Model è responsabile della gestione dei dati dell'applicazione, nonché del loro salvataggio e della loro elaborazione. Inoltre, il Model può contenere la logica dell'applicazione che non è specifica dell'interfaccia utente, come ad esempio la logica che determina il comportamento dell'applicazione in risposta a determinati eventi, come un'interazione da parte dell'utente oppure la risposta del server in seguito ad una request. Il Model non ha alcuna dipendenza dalla View o dal ViewModel. Ciò significa che è possibile accedere al Model in diverse punti dell'applicazione, indipendentemente dall'interfaccia utente specificatamente visualizzata in quell'istante;

- View: La View nel MVVM rappresenta la parte dell'applicazione che si occupa di andare a costituire l'interfaccia utente. In altre parole, la View definisce la struttura, il layout e il comportamento della UI. Questo componente del MVVM è responsabile della visualizzazione dei dati forniti dal ViewModel, ma anche della decodifica degli input da parte dell'utente, nonché della definizione della reazione del sistema in seguito all'interazione. Il fulcro del MVVM è definito dal principio architetturale per cui la View non ha alcuna conoscenza del Model sottostante, infatti essa può ricevere dati unicamente dal ViewModel, oppure richiederli a quest'ultimo qualora fosse necessario. Dunque, è

fondamentale che la View sia progettata in modo indipendente dal ViewModel e dal Model, in modo da rendere più facile la manutenzione e l'aggiornamento dell'UI;

- ViewModel: Il ViewModel nel MVVM è il componente che fa da intermediario tra la View e il Model dell'applicazione. Il ViewModel fornisce alla View i dati necessari per visualizzare l'interfaccia utente e gestisce la logica dell'applicazione in risposta alle azioni dell'utente sulla View. Il ViewModel rappresenta la parte dell'applicazione che gestisce la logica di presentazione e la gestione degli eventi dell'UI. Inoltre, il ViewModel fornisce una rappresentazione delle informazioni contenute nel Model in un formato che può essere facilmente visualizzato e gestito dalla View. Il ViewModel si occupa anche di mantenere lo stato dell'applicazione, come ad esempio lo stato dell'UI o lo stato dei dati dell'applicazione. In questo modo, il ViewModel garantisce la consistenza dell'UI e delle informazioni visualizzate dalla View. Il ViewModel non ha alcuna conoscenza diretta dell'interfaccia utente e del Model sottostante, ma si interfaccia con entrambi attraverso i meccanismi di binding forniti dal framework di sviluppo utilizzato.

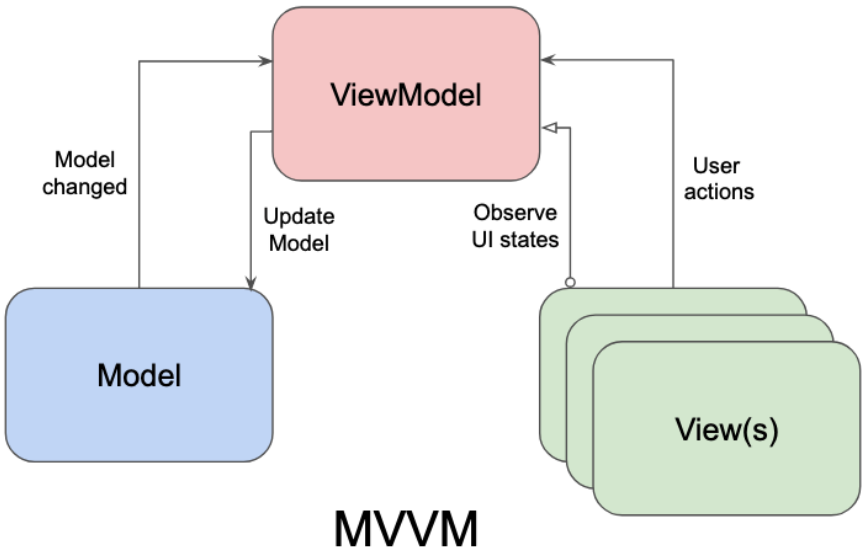

Figura 6.1: Struttura modello MVVM

In questo modo, il MVVM consente di separare la logica dell'interfaccia utente dalla logica dell'applicazione, migliorando la modularità e la manutenibilità del

codice. Inoltre, il ViewModel può essere testato in modo indipendente dalla View, semplificando e velocizzando la fase di testing dell'applicazione. In Flutter è possibile andare a sviluppare questo pattern architetturale attraverso l'utilizzo del package "provider", già precedentemente citato. Esso da accesso accesso alla classe "ChangeNotifier", la quale permette di fare in modo che altre classi possano essere informate del cambiamento dei dati contenuti in essa. La particolarità del "ChangeNotifier" risiede nel fatto che l'intenzione di notificare il cambiamento dello stato della classe deve essere esplicita, infatti lo sviluppatore deve utilizzare il metodo "notifyListeners". Va sottolineato che questo, quando invocato, notificherà il cambiamento a tutti i listeners del "ChangeNotifier" istanziati. Il pacchetto "provider" permette inoltre di utilizzare la classe "Consumer": essa deve essere ritornata internamente al metodo build() di un Widget, ed a sua volta ritornerà l'interfaccia desiderata per il Widget stesso. Quando il metodo "notifyListeners()" viene chiamato, tutti i Consumer attivi imporrano una nuova esecuzione del metodo build(), di fatto costringendo la UI ad aggiornarsi tenendo conto del nuovo stato del "ChangeNotifier". Per poter sfruttare questi componenti è necessario ricorrere alla classe "ChangeNotifierProvider", la quale richiede che venga passato come parametro del costruttore un metodo che istanzierà il "ChangeNotifier", ma anche un componente figlio. Quest'ultimo, così come tutti i componenti che da lui deriva nell'albero delle viste, avranno la possibilità di implementare il Consumer, il quale farà riferimento al "ChangeNotifier" così istanziato. Il pattern MVVM può essere completato istanziando la classe che costituirà il model, contenente la logica di business, direttamente all'interno della classe che estende il "ChangeNotifier". Nel caso di applicazioni molto grandi è bene sfruttare differenti ViewModel, nel numero ideale di uno per ogni View, che per semplicità potremmo considerare una schermata dell'app. Tuttavia, volendo ricorrere a soluzioni meno complesse, è possibile passare l'intera app al costruttore del ChangeNotifierProvider come componente figlio, facendo sì che ogni View dell'app possa accedere al ChangeNotifier. La seconda soluzione rende facilmente comprensibile il percorso dei dati in app, e per questo ben si adatta a progetti più modesti. Per questa ragione nel caso di studio si è deciso di utilizzare questa strategia. Di seguito presenterò la due classi utilizzate per costituire Model e ViewModel nel caso di studio.

6.2 GameLogic

La classe GameLogic costituisce il Model contenente la logica di business inserita in app. All'interno di questa classe verranno innanzitutto istanziate le classi contenenti i dati degli oggetti di gioco, quali:

- Zone: la classe zone identifica il singolo livello, essa richiede che al suo costruttore venga passato il codice identificativo del livello, nonché tutti i

valori relativi ai dati iniziali del livello ed agli obbiettivi raggiungibili in ognuno di essi. La classe Zone richiede anche che venga passato un intero rappresentante la mole dell'edificio, poiché questa influenzerà il costo di alcune operazioni, come ad esempio l'implementazione di illuminazione a LED per sostituire quella ad incandescenza. Il costruttore è poi completato da due liste di stringhe: la prima indicherà la miglior combinazione di carte possibile per il livello, mentre la seconda indicherà le carte inizialmente presenti sul "tabellone di gioco" virtuale;

- Context: questa classe andrà a rappresentare il contesto casuale in cui verranno inseriti gli edifici su cui effettuare i lavori nei vari livelli. Questa classe accetta nel suo costruttore un codice identificativo del contesto, una lista dei codici delle carte le cui performance verranno aumentate se giocate in questo contesto, ed allo stesso modo una lista di quelle che invece avranno performance peggiori. Inoltre, è possibile passare al costruttore una mappa contente dei moltiplicatori per ogni statistica iniziale del singolo livello: impatto ambientale, impatto energetico, comfort degli abitanti e budget disponibile;

- CardData: la classe CardData è stata implementata per andare a contenere tutti i dati relativi ad una singola carta, come ad esempio la variazione delle statistiche dell'edificio qualora la carta venisse giocata, ma anche eventuali variazioni nelle performance della carta stessa qualora altre carte fossero giocate. Infatti, poiché in ambito operativo alcuni interventi di riqualificazione energetica degli edifici ne richiedono a loro volta altri, si è deciso di ricreare questa interazione migliorando le prestazioni di alcune carte se giocate in coppia con altre, come ad esempio i pannelli fotovoltaici e la batteria di accumulo.

La GameLogic implementerà una lista di ognuna delle classi appena presentate, in modo tale da contenere tutti i dati che complessivamente descrivono gli asset del gioco. La logica di business che regola il gioco troverà spazio nei metodi della classe. Qui sarà possibile trovare la logica che permette di inserire i giocatori all'interno delle squadre, di dare loro alcune carte iniziali per ogni livello e di definire l'obbiettivo di ogni squadra per ognuno dei livelli. Al contempo la GameLogic si occuperà di ricevere i dati estratti dal database dal ViewModel al fine elaborarli, costruendo nuove ed aggiornate strutture dati da ritornare al ViewModel, in modo che quest'ultimo si occupi unicamente di scriverle sul database. I metodi più importanti contenuti in questa classe sono:

- findCard: questo metodo accetta un codice identificativo di una carta sotto forma di stringa, il codice identificativo di un contesto ed un intero rappresentante il livello corrente. Questo metodo si occupa di andare a ricalcolare le

performance di una carta sulla base del contesto estratto e della grandezza dell'edificio corrispondente al livello corrente. Si farà dunque ricorso a questo metodo per ottenere i dati delle singole carte prima che vengano giocate;

- obtainPlayedCardsStatsMap: come detto, alcune carte possono migliorare le proprie prestazioni se altre carte vengono giocate, per cui questo metodo prende in input una Map rappresentante gli interventi progettati, ossia la carta giocata per ogni mese dell'anno, e restituirà le performance di ogni singola carta alla luce della presenza o meno dei suddetti bonus sulle performance;
- evaluatePoints: questo metodo si occupa di andare a calcolare il punteggio corrente di ogni squadra. All'interno di questo metodo si andrà a definire un coefficiente che indicherà la prossimità del set di carte giocate alla combinazione perfetta per il livello corrente. Questo coefficiente verrà utilizzato come moltiplicatore per il massimo punteggio ottenibile in un livello, restituendo così i punti accumulati della squadra.

6.3 GameModel

La classe GameModel costituisce il ViewModel dell'applicazione. Essa, come preannunciato precedentemente, estende la classe "ChangeNotifier", di conseguenza ha la possibilità di notificare ai propri listeners l'aggiornamento del suo stato. Questa classe viene istanziata direttamente attraverso l'uso di una lambda nel costruttore del "ChangeNotifierProvider", ed a sua volta istanzierà la classe GameLogic. Attraverso questo processo otterremo di fatto il Model, facendo sì che unicamente il ViewModel vi si possa effettivamente interfacciare. Il GameModel si occuperà anche delle richieste di lettura e scrittura al database, dopodiché immagazzinerà i dati recuperati dal cloud in strutture dati locali. In questo modo il ViewModel diviene l'unico punto di smistamento dei dati interni all'app, siano questi provenienti da elaborazioni locali del Model, oppure remote eseguite in cloud tramite Firebase. La libreria Firebase per Flutter permette di andare a salvare una variabile reference di base che punterà all'url che costituisce la root del repository. Su questa variabile è possibile concatenare una serie di metodi forniti dalla stessa libreria al fine di navigare il database, oppure richiedere operazioni di lettura e scrittura. I principali metodi sono:

- child: questo metodo accetta una stringa e permette di spostarsi più in profondità nella struttura dati unica tipica di RealTime Database. Questo metodo è usualmente concatenato più volte, poiché esso permette di scendere di un solo livello gerarchico per volta, per cui per andare a definire un vero e proprio path per giungere al dato desiderato serve chiamare il metodo più volte. Nel caso in cui si voglia effettuare una lettura di un dato, avendo specificato

un path errato tramite il metodo child(), l'operazione restituirà null. Se lo stesso processo viene però effettuato in un'operazione di scrittura verrà creato il path specificato nel database;

- • get(): il metodo get() permette di ottenere una struttura dati contenente tutti i dati a livello gerarchico inferiore rispetto al child() su cui è chiamato il metodo. Questi dati verranno forniti sotto forma di DataSnapshot, ossia una classe specifica definita dalla libreria Firebase, anch'essa navigabile attraverso l'utilizzo del metodo child(). Inoltre, è molto semplice effettuare elaborazioni su di un DataSnapshot grazie al metodo children, che restituisce tutti i figli di più alto livello della struttura dati sotto forma di iterabile. Il get() è un metodo asincrono, per cui è necessario specificare la parola chiave async nella signature della chiamata. Ad un get() è generalmente consigliabile concatenare un whenComplete() oppure un then(), ossia metodi che attendono il termine del get() per eseguire della logica specificata in una lambda, con il primo metodo che non accetta parametri in input, ed il secondo che richiede che venga passato il DataSnapshot ottenuto dalla get();

- onValue.listen(): questi metodi concatenati permettono di effettuare due operazioni complementari: onValue, chiamato su di uno specifico child(), permette di notificare al client quando il valore del dato puntato varia; mentre listen() apre uno Stream di Dart che si occuperà di chiamare una callback ad ogni variazione del dato. Listen() accetta quindi una lambda, avente come unico input un DataSnapshot che immagazzina i dati aggiornati, contenente la logica che si occuperà di gestire i nuovi dati;

- set(): il metodo set() permette di scrivere uno specifico valore per il child() sul quale è chiamato.

Lavorando con i DataSnapshot è bene tenere a mente che i singoli dati possono essere ottenuti attraverso l'attributo "value", chiamato su di uno specifico percorso definito tramite concatenazione di metodi child(), il quale restituirà sempre un dynamic, il che vuol dire che, qualora l'informazione ottenuta fosse un null il compilatore non darebbe errore. Per questa ragione è buona pratica imporre un casting sul valore ottenuto dal "value", in modo da impedire la compilazione in casi di errore. Infine, la classe GameModel è stata sfruttata per chiamare delle callback che si sarebbero occupate specificatamente di modificare alcuni punti della UI in seguito a chiamate al database. Come già descritto, il "Consumer" permette di modificare l'interfaccia in seguito alla modifica dello stato del "ChangeNotifier", ossia, nel caso di studio, del GameModel. Tuttavia, questo può essere ottenuto solo ricorrendo al metodo "notifyfListeners", il quale invia un messaggio broadcast a tutti i listeners attivi, costringendo ognuno di questi a ricostruire la propria interfaccia. Considerando che la ricostruzione della UI richiede risorse computazionali

```
void listenToLevelChange() {
  db.child("matches").child(firebasePath![0]).child(firebasePath![1]).child("level").onValue.listen((event) {
    if(!(event.snapshot.value == "")){
      if(playerLevelCounter != event.snapshot.child("count").value as int){
        playerLevelCounter = event.snapshot.child("count").value as int;
        playerContextCode = event.snapshot.child("context").value as String;
      }

      if(playerLevelStatus != event.snapshot.child("status").value as String){
        playerLevelStatus = event.snapshot.child("status").value as String;

        switch(event.snapshot.child("status").value.toString()){
          case "preparing" : {
            splash = true;
          }
          break;
          case"play" : {
            setDialogData(DialogData("level $playerLevelCounter", null, true));
          }
          break;
        }
    }

      if (event.snapshot.child("status").value.toString() == "preparing" && playerTimerCountdown != null){
        playerTimer!.cancel();
        playerTimer = null;
        playerTimerCountdown= null;
        setTimeOutTrue();

      }
      notifyListeners();
    }
  });
}
```

Figura 6.2: Esempio di chiamata Firebase

consistenti, nasce la necessità di imporre ricostruzioni mirate a singoli Widget, che in alcuni casi però devono necessariamente seguire ad una chiamata al database. Per questo motivo di è sviluppato un pattern ad hoc per imporre rebuild della UI mirati. La classe GameModel quando istanziata possiede una serie di variabili del tipo Function, che in Dart fa riferimento ad una funzione anonima. Queste variabili sono nullable, ed inizialmente a loro si fa corrispondere il valore null. Quando i componenti della UI vengono istanziati questi ottengono il riferimento all'istanza del GameModel attraverso il "Consumer", per cui è possibile effettuare un null check su di una delle Function registrate, e qualora questa fosse nulla, è possibile passarle un metodo che verrà definito internamente al Widget. Quest'ultimo metodo farà uso di setState() per imporre una ricostruzione della UI. Attraverso questo pattern è possibile chiamare metodi relativi ai singoli componenti dell'interfaccia direttamente dal GameModel, impedendo così la ricostruzione di componenti che in quel momento di fatto non necessitano di aggiornamento, ottimizzando così l'uso delle risorse computazionali del device.

6.4 MainScreenContent

Come già descritto in precedenza nel caso di studio in analisi si è fatto uso del package GoRouter, che mette a disposizione il componente ShellRoute, il quale permette di embeddare la navigazione all'interno di un altro componente UI. Il Widget MainScreenContent è stato pensato per essere il punto di appoggio che definisce l'interfaccia dell'applicazione, infatti esso costuisce uno Scaffold, costituito a sua volta da una statusBar superiore ed una navBar inferiore fisse, ed un body centrale che varia in base alla navigazione effettuata dall'utente. Inoltre, questa classe si occuperà di generare le dialog che dovranno essere mostrate al di sopra dell'interfaccia standard quando necessario. In Flutter è possibile utilizzare il metodo showDialog per aprire una dialog a tutta pagina, il cui layout deve essere ritornato dal metodo stesso. Il GameModel a sua volta possiede una variabile showDialog nullable di tipo DialogData, ossia una classe personalizzata che accetta nel suo costruttore una stringa "title", un Widget "body" ed un boolean "autoExpire". In questo modo è possibile modificare il valore della variabile showDialog all'interno del GameModel per poi notificare la variazione al MainScreenContent, che si occuperà di aprire una dialog avente un titolo superiore con stile predefinito pari alla stringa "title" ed un layout centrale definito dal "body" della DialogData. Il boolean "autoExpire" è stato inserito invece per definire due diversi comportamenti per le dialog: se posto a true, la dialog verrà chiusa automaticamente in seguito ad un tempo predefinito tramite l'utilizzo di un Future, altrimenti il layout della dialog presenterà un bottone inferiore che se premuto chiuderà la finestra. La classe MainScreenContent inserisce lo Scaffold all'interno di uno Stack, così che alla prima costruzione dell'interfaccia, nel caso in cui l'utente non abbia ancora svolto il tutorial, questo appaia come overlay al di sopra della UI standard.

6.5 InfoRow

L' InfoRow rappresenta la statusBar dell'applicazione. Essa dispone di un layout standard che mostra il team in cui il giocatore è inserito, il budget della squadra e, se il giocatore è di turno, il tempo restante per eseguire la propria mossa. La particolarità di questa statusBar risiede nel fatto che è in grado di modificare il proprio layout per fornire ulteriori informazioni all'utente, come ad esempio l'inizio del proprio turno e la mancanza di budget per eseguire la mossa desiderata. In particolare, tali informazioni secondarie verranno mostrate facendo "uscire" il layout standard a destra e facendo "entrare" il layout con le informazioni aggiuntive da sinistra, per poi tornare alla visualizzazione standard nello stesso modo. Questo meccanismo è reso possibile tramite il ricorso al già citato Widget SlideTransition governato a turno da due AnimationController, uno per l'uscita di scena di un layout

ed uno per l'entrata contemporanea del successivo, governati tramite una sequenza di Future, che, essendo metodi asincroni non bloccanti, ben si prestano a governare coreografie composte da più animazioni. All'interno del Widget SlideTransition si pone un InfoRowDynamicContent, ossia un componente personalizzato che richiede nel suo costruttore un enumeratore indicante il tipo di layout da mostrare. All'uscita di un layout si istanzia una nuova versione di questo componente che andrà a generare il layout in entrata.

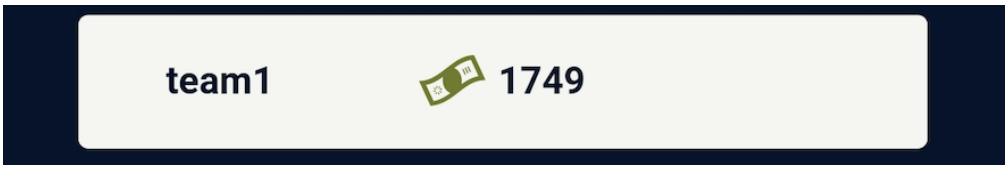

Figura 6.3: InfoRow

6.6 CardSelectionScreen

Il CardSelectionScreen rappresenta la pagina nella quale gli utenti possono navigare le carte in loro possesso. Esso si compone di tre elementi principali, che procederò a descrivere brevemente muovendo dall'alto verso il basso. Al di sotto della statusBar possiamo trovare un classico pager con annessa tab. Questo componente visivo serve per poter visualizzare quali interventi sono stati decisi per ogni mese dell'anno, ma allo stesso tempo permette di poter giocare una carta se la posizione è libera. Il pager possiede un listener onTap() per l'area della Card centrale che si occuperà di chiamare una callback, la quale, effettuati i dovuti controlli, permetterà di giocare la carta imponendo al GameModel di aggiornare il database nella sezione "playedCards" del team corrispondente. Allo stesso tempo questo componente modificherà la sua interfaccia in seguito alle modifiche del database (che vengono notificate automaticamente da Firebase), per cui la scrittura sul server implicherà l'aggiornamento lato client. Al di sotto, nella zona centrale, a destra, troviamo un button costituito da una grafica 2D animata disponibile grazie all'utilizzo del package lottie già presentato in precedenza. Se cliccato questo bottone permette di accedere ad una schermata che presenterà informazioni più esaustive circa la carta centrale nel componente sottostante. Quest'ultimo, ossia la "rosa" di carte inferiore, nasce dal desiderio di restituire agli utenti la sensazione di star effettivamente giocando a carte, ricordando dunque il modo in cui queste vengono fisicamente tenute in mano. Tale Widget personalizzato permette di scorrere le carte attraverso uno swipe nella direzione desiderata. Il meccanismo sottostante fa uso di sei differenti carte, delle quali solo cinque per volta mostrate a schermo, poiché l'ultima viene sfruttata come componente "fantasma":

quando l'utente effettua lo swipe, la carta nascosta, posta al di "sotto" dello schermo, e dunque non renderizzata, verrà mostrata, venendo popolata con i dati della carta successiva posseduta dall'utente. Contemporaneamente tutte le altre carte faranno spazio alla nuova entrata con un'animazione, facendo sì che l'ultima carta nella posizione opposta a quella appena aggiunta alla "mano" esca dallo schermo divenendo la nuova carta "fantasma". Da un punto di vista realizzativo il comportamento desiderato è stato ottenuto sfruttando due differenti Map: la prima costituita da coppie chiave valore in cui ad una stringa identificante la carta si faceva corrispondere la trasformazione che definiva rotazione e traslazione della stessa; mentre la seconda alle stesse identiche stringhe identificative accoppiava delle interpolazioni fra una matrice iniziale e finale, la cui progressione dipendeva da altrettanti controller. La UI si costituiva di una serie di AnimatedBuilder accoppiati a delle Transform, che, come già illustrato, permettono di ottenere animazioni complesse. In questo modo ad ogni swipe serviva unicamente calcolare le matrici che definivano le nuove trasformazioni delle carte, per poi far procedere i controller ad attivare le animazioni. Nel corso dell'animazione sorge però la necessità che le carte cambino anche il loro posizionamento sull'asse z, poiché quella centrale deve sempre sovrastare le altre. Questa caratteristica può essere ottenuta sfruttando il package indexer, che dando accesso al Widget Indexer permette di ottenere uno Stack i cui figli siano rappresentati in un ordine personalizzato, che, all'occorrenza, può essere modificato ricorrendo ad una setState(), come è stato effettuato nel caso di studio. Infine, gli swipe potevano essere decodificati grazie all'utilizzo di un oggetto Listener applicato unicamente sulla carta centrale, attraverso il componente custom UndetailedCardLayout; che, fra le altre cose, accetta nel suo costruttore anche i dati relativi alle singole carte mostrate.

Figura 6.4: CardSelectionScreen

6.7 RetrieveCardScreen e OtherTeamsScreen

Completano l'interfaccia lato giocatore due ulteriori pagine. La prima serve a poter ritirare una carta che è stata giocata in una data posizione sul tabellone di gioco. Il Layout delle carte in questo caso diviene molto più ricco, fornendo informazioni più dettagliate, come un campo di testo che descrive l'intervento e le condizioni ottimali in cui eseguirlo. Inoltre, qualora la carta giocata stia performando meglio,

oppure peggio, del normale sarà possibile cliccare sulle Card che contengono le singole statistiche per ampliarle ed ottenere una descrizione delle ragioni di tale comportamento. La seconda schermata si occupa di mostrare l'andamento di tutte le squadre nella partita corrente, fornendo informazioni circa i risultati ottenuti dagli interventi progettati finora, ma anche il punteggio corrente.

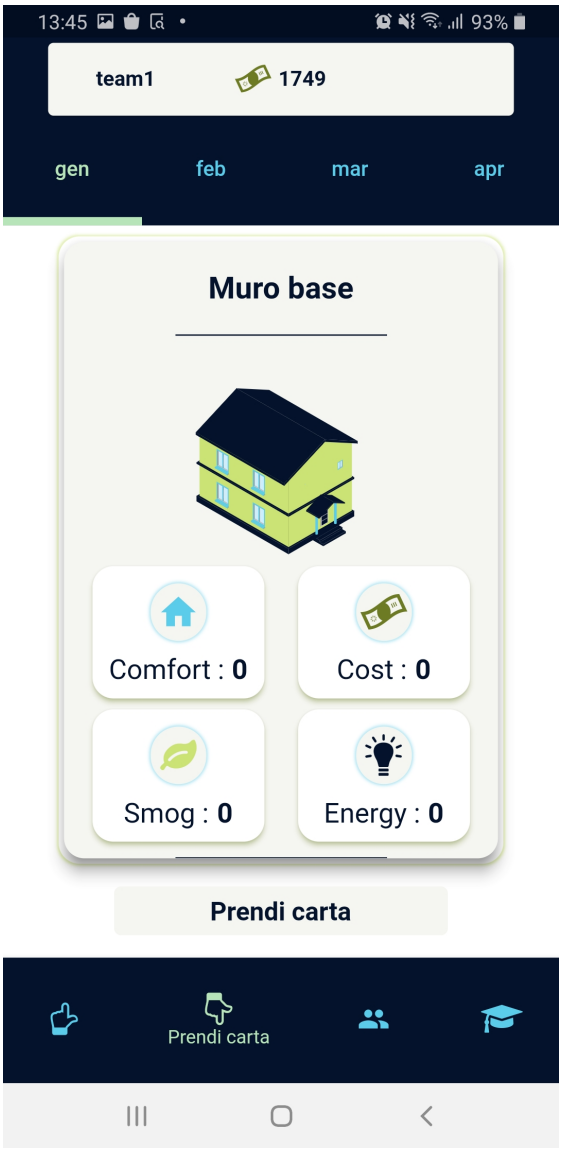

Figura 6.5: RetrieveCardScreeen

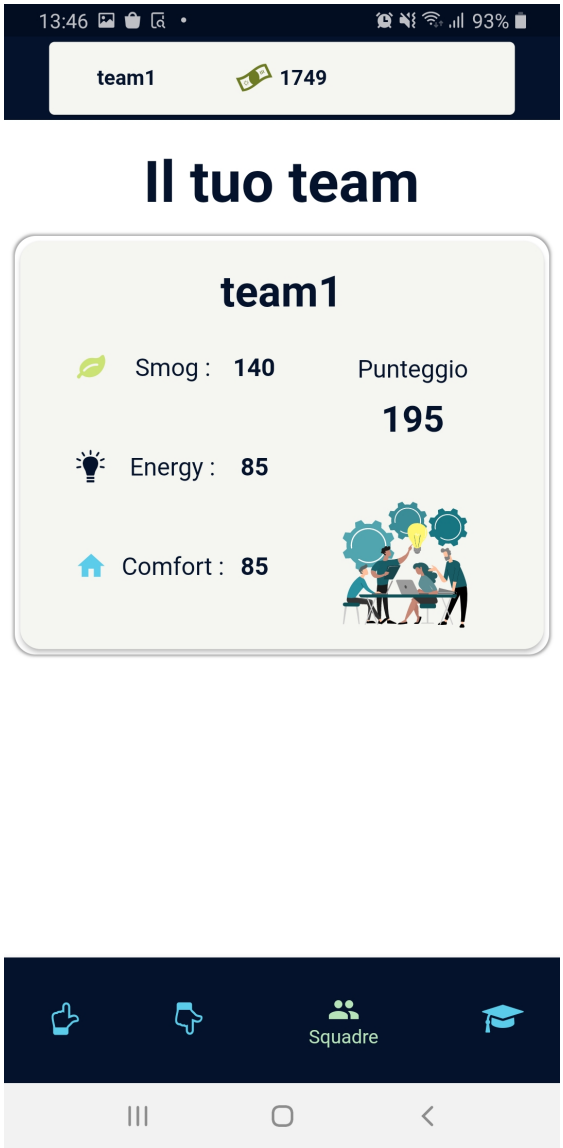

Figura 6.6: OtherTeamsScreen

6.8 GameBoard

Il GameBoard costituisce la principale schermata dell'applicazione lato game master, poiché questa, venendo proiettata sulle lavagne interattive delle classi, rappresenterà il "tabellone di gioco", e si modificherà sulla base delle azioni dei singoli giocatori. Essa si compone di una serie di Card, da un minimo di uno per le partite a giocatore

singolo ad un massimo di 4, ossia il numero massimo delle squadre. Ogni card a sua volta possiede 4 aree distinte che ne compongono la resa visiva:

- il nome della squadra figura in alto al centro della carta;
- a sinistra è presente un indicatore circolare che definisce la vicinanza della squadra al suo obbiettivo;
- a destra alla fine di ogni livello compare il punteggio accumulato, seguito da un'icona rappresentante la posizione provvisoria in classifica;
- la parte centrale è occupata da uno Stack di png che si modifica al modificarsi degli interventi previsti da una squadra. Per ogni carta giocata o sottratta dal "tabellone" questo stack si modificherà mostrando visivamente il risultato provvisorio degli interventi previsti in quell'istante.

Capitolo 7

Conclusioni

Questa tesi si è sviluppata attraverso differenti argomenti, come la definizione di cosa intendiamo quando facciamo riferimento ai serious game, come questi possano essere progettati e quali obbiettivi debbano raggiungere. Questa sezione della tesi si è conclusa con la descrizione delle decisioni prese in fase di design del caso di studio, tenendo conto delle caratteristiche particolari in cui questo prodotto sarebbe stato utilizzato. il mondo delle applicazioni mobile, diviso fra le prestazioni dei linguaggi nativi e l'efficienza dei framework cross- piattaforma. Successivamente, avvicinandoci al caso di studio, si è presentato Flutter, con le potenzialità che lo caratterizzato, fornite in primis dal suo linguaggio dedicato, Dart, ma anche dalla struttura stratificata che ne garantisce le prestazioni elevate. Tramite l'utilizzo dei Widget è possibile andare a definire componenti visivi che definiranno l'interfaccia dell'applicazione, e, al contempo, permetteranno di interfacciarsi con i componenti hardware del device. Sono stati poi presentati i principali Widget che sarebbero stati utilizzati nel caso di studio, indicando le forti sinergie che si possono andare a creare utilizzandoli in modo complementare. Successivamente si è passati a definire il funzionamento della piattaforma cloud Google Firebase, il quale permette di ottenere una serie di servizi backend differenti, a partire da un database, sia esso ottenuto tramite l'utilizzo di RealTime Database oppure Firestore, fino a giungere allo storage in cloud di immagini e filmati, distribuibili poi tramite una CDN preposta di proprietà del colosso californiano. In seguito, è stata presentata la struttura del database utilizzato per mantenere lo stato dell'applicazione sviluppata, descrivendo nel dettaglio lo scopo di ognuno dei dati conservati. L'insieme di questi strumenti tecnologici è stato utilizzato nel caso di studio, ossia un serious game per conto dell'azienda Edilclima di Borgomanero (NO), che si occupa della creazione di software per la valutazione della dispersione energetica nelle abitazioni. L'azienda, riconoscendo l'importanza della prevenzione nell'ambito, da anni si propone di effettuare interventi informativi nelle scuole medie inferiori e superiori del territorio. Dal desiderio di sfruttare le caratteristiche peculiari dei serious game nel corso di

queste lezioni l'azienda si è in passato cimentata nella creazione di un gioco da tavolo. L'esperienza del Covid ha, tuttavia, fatto sorgere la necessità di creare un nuovo gioco educativo sotto forma di mobile app. Si è dunque presentato il modello MVVM utilizzato per la realizzazione della mobile app, nonché la sua implementazione nel caso di studio specifico attraverso l'utilizzo di "ChangeNotifier" e "Consumer", oggetti messi a disposizione dal package "provider", specifico per Flutter. Infine, si sono descritte le principali schermate costituenti l'applicazione, ponendo il focus sull'implementazione delle funzionalità più complesse. L'insieme di queste tecnologie ha dunque permesso di sviluppare un serious game che potrà essere utilizzato in ambito scolastico, al fine di sensibilizzare gli studenti circa l'importanza e le implicazioni degli interventi di riqualificazione energetica degli edifici.

7.1 Sviluppi futuri

Il caso di studio oggetto di questa tesi è ora giunto al termine della propria fase di progettazione e sviluppo. Dovrebbe dunque seguire una fase di testing della mobile app, al fine di scovare errori non ancora corretti, nonché analizzare tutti i possibili worst case, come ad esempio situazioni di connessione internet assente per periodi brevi o lunghi, che necessiterebbero di soluzioni ad hoc da implementare al fine di aggiungere robustezza al prodotto. In seguito, si potrebbe passare alla fase di performance optimization, che dovrebbe necessariamente passare attraverso un tracking del numero di rebuild dell'interfaccia, in modo tale da minimizzare le ricostruzioni della UI non necessarie. Questo processo garantirebbe migliori prestazioni in app, ma allo stesso tempo permetterebbe di minimizzare l'utilizzo delle risorse computazionali, efficientando il software dal punto di vista del consumo energetico. Infine, richiedendo la collaborazione dell'azienda committente, si potrebbe pensare di ampliare il numero di livelli o delle carte di gioco, così da presentare agli utilizzatori situazioni molto differenti le une dalle altre in diverse partite, massimizzando quindi la ri-giocabilità del prodotto anche al di fuori del contesto scolastico per cui è specificatamente pensato.

Bibliografia

[1] Vindice Deplano. «La simulazione come gioco e come modello di apprendimento». In: *Apprendimento e nuove tecnologie. Modelli e strumenti, Franco Angeli, Milano* (2010) (cit. a p. 1).

[2] Jon Ord. «John Dewey and Experiential Learning: Developing the theory of youth work». In: *Youth & Policy* 108.1 (2012), pp. 55–72 (cit. a p. 3).

[3] R Caillois. «I giochi e gli uomini, ed. it». In: *Milano, Bompiani* (1967) (cit. a p. 5).

[4] Luigi Maria Anolli e Fabrizia Mantovani. *Come funziona la nostra mente. Apprendimento, simulazione e Serious Games*. Il Mulino, 2011 (cit. a p. 10).

[5] Roberto Vardisio. *Serious game*. Raffaello Cortina Editore, 2020 (cit. a p. 13).

[6] Mihály Csıkszentmihályi. *Flow: Psicologia dell'esperienza ottimale*. ROI Edizioni, 2022 (cit. a p. 18).

[7] Statista. *cross-platform framework usage*. URL: `https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/` (cit. a p. 26).

[8] React. *React Native*. URL: `https://reactnative.dev/docs/getting-started` (cit. a p. 27).

[9] Microsoft. *Xamarin*. URL: `https://learn.microsoft.com/en-us/xamarin/` (cit. a p. 28).

[10] Ionic. *Ionic framework*. URL: `https://ionicframework.com/docs` (cit. a p. 28).

[11] Google. *Dart documentation*. URL: `https://dart.dev/guides` (cit. a p. 30).

[12] HTML mobile. *Come funziona Flutter*. URL: `https://www.html.it/pag/377313/il-framework-dettagli-tecnici/` (cit. a p. 33).

[13] Google. *Widget class*. URL: `https://api.flutter.dev/flutter/widgets/Widget-class.html` (cit. a p. 34).

[14] Google. *GoRouter*. URL: `https://pub.dev/packages/go-router` (cit. a p. 36).

[15] Google. *Firebase*. URL: `https://firebase.google.com/docs?hl=it` (cit. a p. 46).

[16] JSON.org. *What is JSON?* URL: `https://www.json.org/json-it.html` (cit. a p. 49).

[17] geekandjob. *Cos'è MVVM e a cosa serve?* URL: `https://www.geekandjob.com/wiki/mvvm` (cit. a p. 56).