Tesina Finale di

Programmazione di Interfacce Grafiche e Dispositivi Mobili

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2020-2021

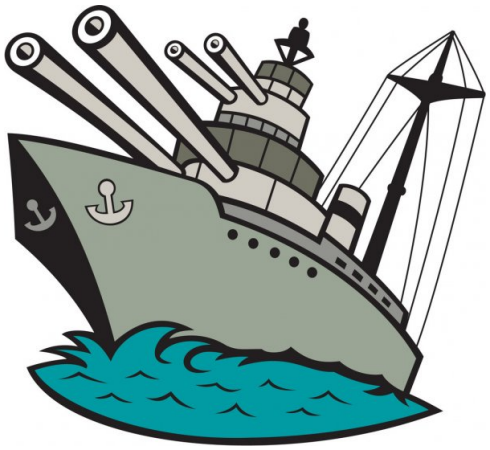
Dipartimento di Ingegneria

docente

Prof. Luca Grilli

Battleship

app nativa per Android sviluppata con AndroidStudio



Studenti

312879 **Edoardo Marchetti** edoardo.marchetti1@studenti.unipg.it

Data ultimo aggiornamento: 28 giugno 2021

0. Indice

1. Descrizione del Problema

L'obiettivo di questo lavoro è lo sviluppo di un'applicazione per Android, denominata *Battleship* (di seguito progetto o app), che prende spunto dall'applicazione Fleet Battle già disponibile sul Play Store.

L'applicazione sarà implementata utilizzando la IDE Android Studio e il codice prodotto sarà testato e ottimizzato per la versione 6.0 (Marshmallow) del sistema operativo Android. I dispositivi che verranno utilizzati per il debug dell'applicazione saranno un Samsung Galaxy A8 (2220 x 1080px) e gli emulatori Nexus 6 (1920x1080px) con API 23 e Pixel 2 (1920 x 1080px) con API 29.

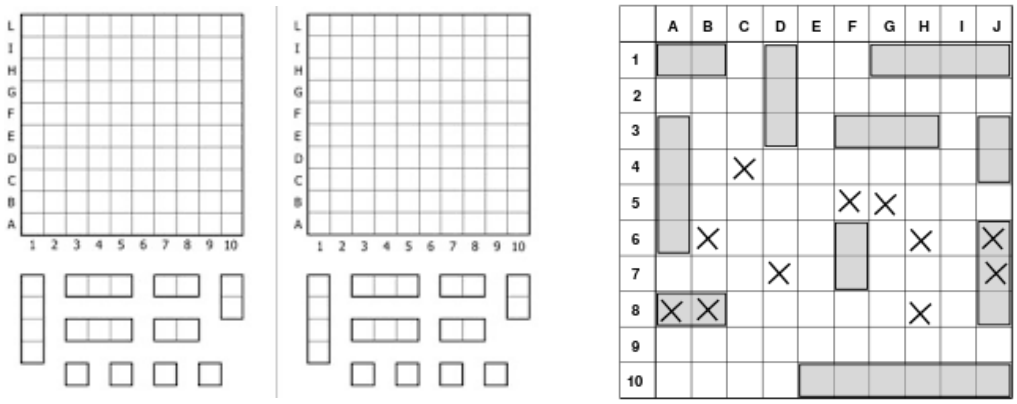
Di seguito sarà data una breve descrizione del gioco originale *Battaglia Navale*, dopodiché si fornirà una descrizione della versione che si intende realizzare.

1.1 Il Gioco Battaglia Navale

La *Battaglia Navale* è un gioco di strategia per due giocatori il cui scopo è affondare tutte le navi del giocatore avversario.

Per giocare sono necessarie quattro tabelle quadrate (due per giocatore), solitamente composte da 10 righe e 10 colonne. Ogni cella della tabella è identificata da una coppia di coordinate, corrispondenti a colonna e riga; tradizionalmente si usano le lettere per indicare le colonne e i numeri per indicare le righe (es: "A-5", "F-8"). I giocatori prima della partita posizionano le proprie navi su una delle due tabelle a loro disposizione senza farle vedere all'avversario. Ogni nave andrà ad occupare un numero di quadretti adiacenti in linea retta (orizzontale o verticale) sulla tabella. I giocatori si accordano preliminarmente su quante navi dovranno posizionare e di quali dimensioni. A turno i giocatori chiameranno la coppia di coordinate corrispondente al quadretto che intendono colpire. Nel caso in cui una parte di nave venga colpita, l'avversario dirà: "colpita!", altrimenti verrà dichiarato: "acqua" o "mancato". Quando l'ultima parte di una nave non ancora affondata

viene colpita, il giocatore che subisce il colpo dirà: "colpita e affondata!". Ogni volta che un giocatore chiamerà una coppia di coordinate, dovrà segnare l'esito del colpo sulla sua seconda tabella.



(a) (b)

Figura 1.1: Un esempio delle due tabelle a disposizione di ogni giocatore (a) e di una possibile configurazione della flotta (b). Le X nella seconda immagine indicano i colpi messi a segno da parte dell'avversario

1.2 L'applicazione Battleship

Il progetto proposto intende fornire una versione mobile del gioco originale. Tramite una rappresentazione grafica l'app permetterà all'utente di elaborare una propria strategia da allenare nella versione offline per poi andare ad affrontare nemici online alla conquista dei 7 mari in tre diversi scenari di gioco: classico, standard, russo.

2. Specifica dei Requisiti

L'applicazione dovrà soddisfare due categorie di requisiti: alcuni richiesti durante il gameplay e altri che appartengono all'interfaccia generale del gioco. Verranno quindi elencati alcuni requisiti facoltativi.

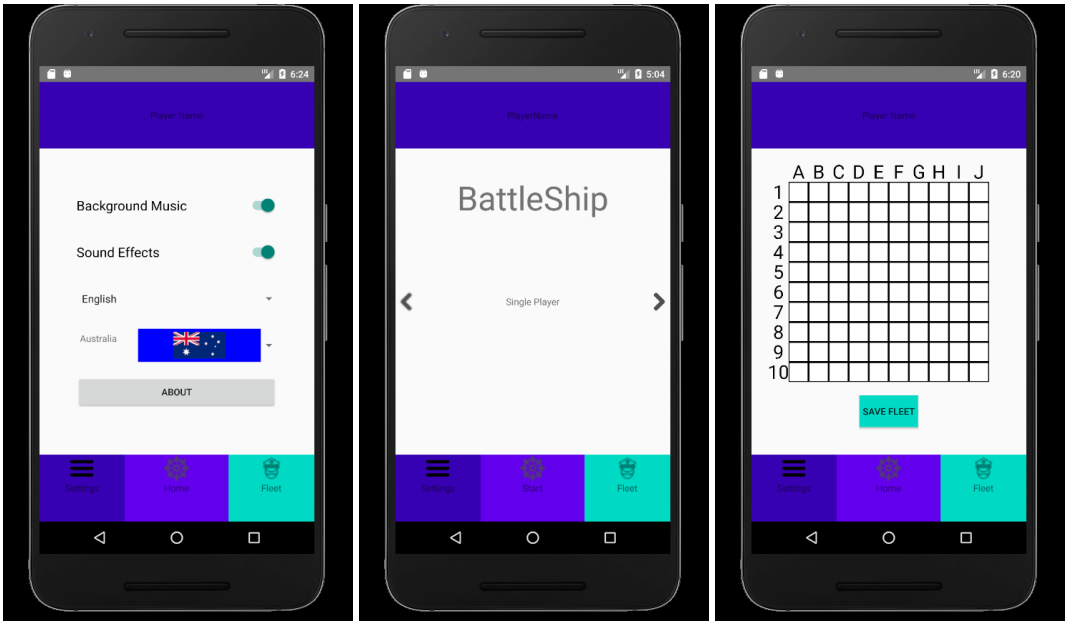
2.1 Requisiti generali

1. Dall'activity principale si dovrà poter accedere a tre diversi menù: home, settings, flotta.

2. Dal menù ***home*** si dovrà poter scegliere la modalità di gioco (riportate al punto 1 dei requisiti di gameplay) desiderata. La schermata inoltre dovrà mostrare il titolo dell'app.

3. Dal menù ***settings*** si potranno attivare o disattivare la musica di sottofondo e gli effetti sonori delle animazioni in maniera indipendente l'una dall'altra. Si potrà, inoltre, selezionare la lingua tra italiano e inglese e consultare tramite un click sul pulsante "about" le regole del gioco e altre informazioni.

4. Dal menù ***flotta*** l'utente potrà decidere una formazione predefinita per le sue navi.



(a) (b) (c)

Figura 2.1: da sinistra a destra un esempio della struttura di base dei menù settings, home e flotta. (Colori e immagini non sono definitivi)

2.2 Requisiti di gameplay

1. L'utente dovrà poter scegliere tra due diverse modalità di gioco: single player (vs CPU), online (multiplayer su diversi dispositivi).
2. Dovranno essere disponibili tre diversi scenari (o livelli) di gioco, ognuno contraddistinto da un set di navi diverso: classico, standard e russo.
3. Prima dell'inizio della partita l'utente dovrà avere la possibilità di poter disporre le navi sulla propria griglia (di dimensione 10x10) e potrà ruotare la nave di 90° gradi con un semplice tocco sulla nave interessata.
4. La casella che si vuol colpire potrà essere indicata attraverso l'uso del touch screen e, al momento del tocco, la riga e la colonna corrispondente dovranno essere evidenziate.
5. Una volta dichiarate le coordinate, un'animazione di un missile dovrà andare a colpire il quadrato corrispondente. Al colpo andato a segno dovrà corrispondere un'animazione che simuli il fuoco, mentre un'animazione che simuli l'acqua verrà attivata qualora il colpo sia andato a vuoto.

6. Dovranno essere presenti effetti sonori che corrispondano all'esplosione della nave o allo "splash" dell'acqua.

7. Le navi avversarie affondate verranno visualizzate completamente nella tabella.

8. Al termine della partita verranno visualizzate le tabelle di entrambi i giocatori mostrando la posizione di tutte le navi (colpite e non) e il nome del giocatore vincitore.

2.3 Requisiti facoltativi

1. Ogni schermata dovrà presentare un header (o badge) composto dal nome del giocatore e un'immagine. Tramite il menù settings l'utente potrà personalizzare l'immagine tra alcune messe a disposizione.

2. L'app dovrà disporre di una terza modalità di gioco 1vs1, selezionabile sempre dal menù start, che permette di affrontare un altro giocatore sullo stesso dispositivo.

3. Battleship dovrà presentare una schermata di splash al lancio dell'app.

3. Progetto

Di seguito vine data una descrizione dell'architettura del porgetto realizzato, illustrando prima quali moduli compongono l'app in generale per poi entrare nella descrizione più dettagliata nelle successive sezioni.

3.1 Architettura del Sistema Software

Per la struttra del software si è preso spunto dalla realizzazione di alcuni videogiochi dal libro *Learning Java by Building Android Games*. In particolare sono stati sviluppati tre moduli: **Activities**, il quale contiene tutte le classi relative alle activity e ai fragment presenti nell'app, **Engines**, contiene le classi adibite alla gestione delle entità necessarie durante la visualizzazione di un campo da gioco, **Entities**, che comprende le classi che definiscono le entità come le navi, i missili, la griglia e i livelli. Inoltre è stato realizzato un modulo **Utils** all'interno del quale sono riportate delle classi di supporto.

Nella pagina seguente è riportata un'immagine del diagramma UML completo. Per una migliore visualizzazione si consiglia di accedere al seguente link.

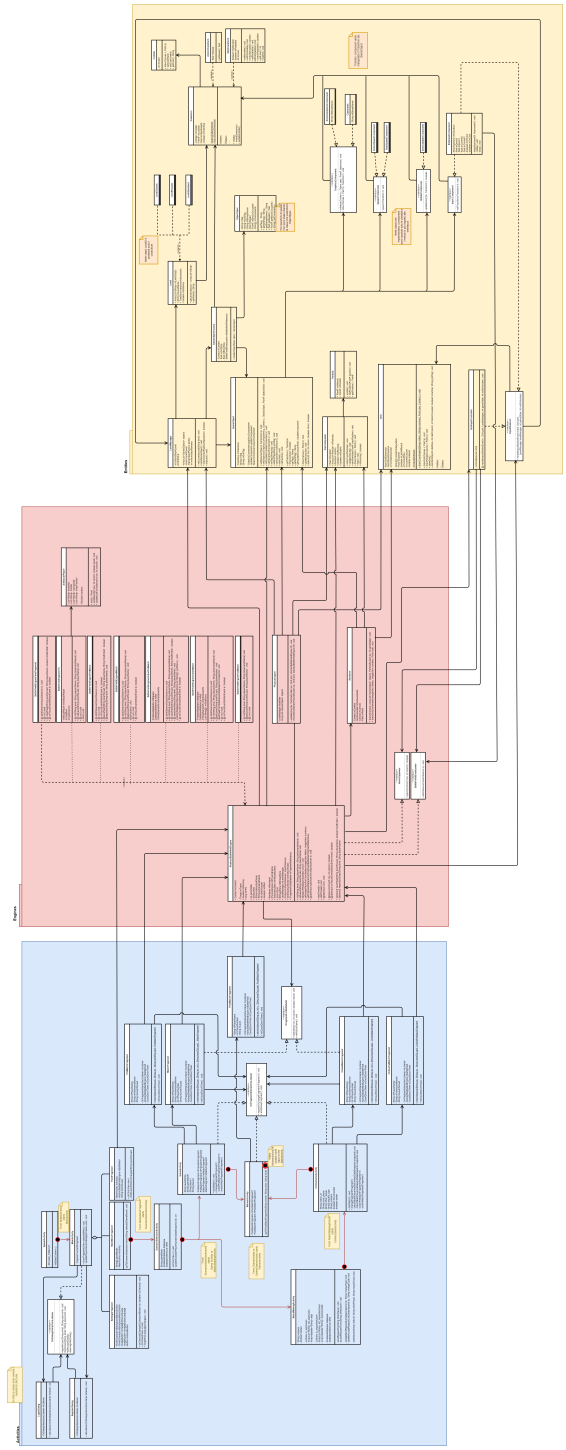


Figura 3.1: Diagramma UML Completo

3.2 Activities

Come anticipato precedentemente, nel modulo ***Activities*** sono presenti tutti i file per la gestione delle activity. Nel diagramma UML viene mostrato anche l'ordine con cui l'utente può visitare le sezioni dell'applicazione.

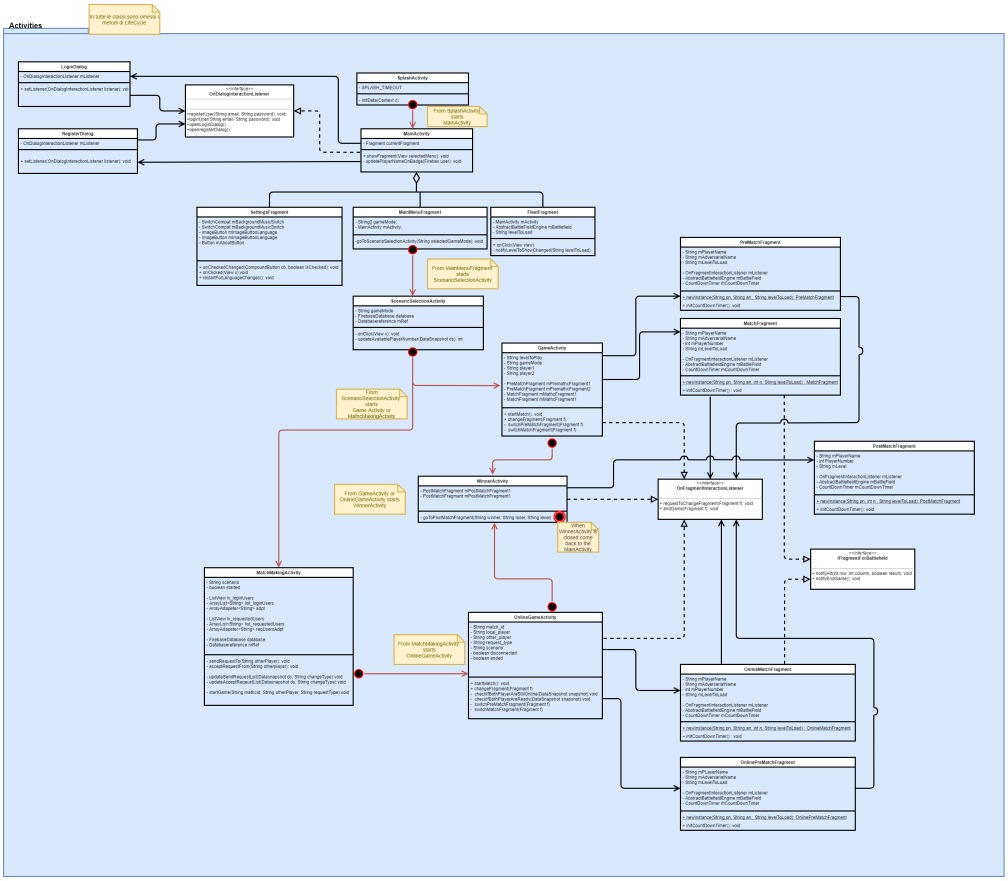


Figura 3.2: Diagramma UML del modulo Activities

Tra le classi principali di questo modulo possiamo trovare la **MainActivity**, la quale viene lanciata dopo un tempo predeterminato e se il metodo *initData(Context c)* della SplashActivity viene eseguito correttamente. Al momento dell'avvio viene mostrato il **MainMenuFragment** dal quale il giocatore può selezionare la modalità di gioco. Tramite un semplice menù l'utente può navigare nel textbfSettingsFragment, da dove può modificare le impostazioni, o nel **FleetFragment**, dal quale può salvare le configurazioni di flotta preferite per ogni scenario disponibile. Se l'app è avviata per la prima volta apparirà anche una finestra di Login tramite la quale il giocatore può accedere oppure andare in una seconda finestra che

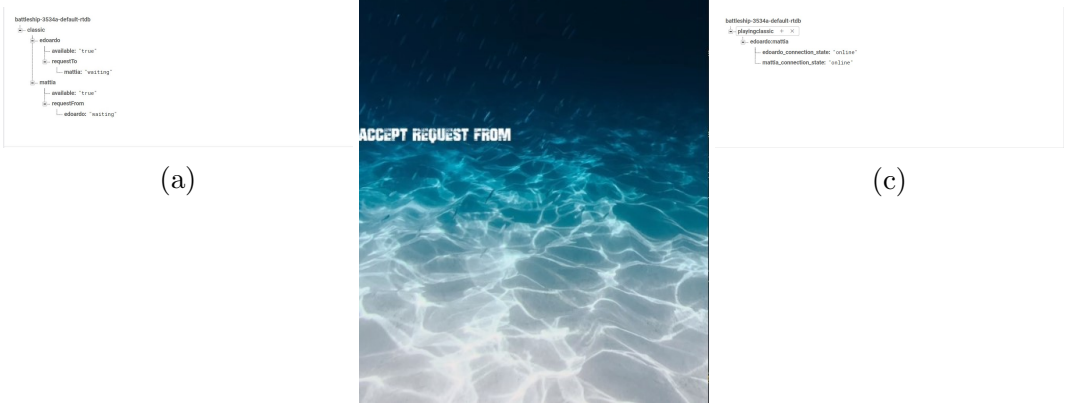
permette la registrazione di un nuovo utente. Il tutto viene gestito tramite i metodi *registerUser(String email, String password)* e *loginUser(String email, String password)*.

(a) (b) (c)

Figura 3.3: Screen dei menu della MainActivity

Nell'activity **ScenarioSelectionActivity** il giocatore potrà quindi scegliere lo scenario con cui giocare, in base al quale cambierà il set di navi e le regole della partita.

Da qui, a seconda alla modalità di gioco scelta si verrà indirizzati in **MatchMakingActivity**, nella quale si possono accettare o inviare richieste di gioco ad altre persone, oppure nella **GameActivity** all'interno della quale si svolgono le partite in modlità offline. La MatchMakingActivity, grazie ai metodi indicati nel diagramma, va ad acquisire tutti gli utenti disponibili dalla piattaforma *Firebase*, la quale mette a disposizione un database in real-time online. Dunque, ogni volta che un giocatore entra o esce da una delle tre stanze (russo, classico, standard) ciò viene registratato nel database e, tramite degli *eventListener*, notificato all'applicazione, la quale aggiorna le liste di invio e accettazione delle richieste di gioco. Una volta che una richiesta di partita viene accettata viene creato un oggetto partita nel database, i due giocatori diventano non disponibili e viene fatto partire il match che verrà gestito dall'activity **OnlineGameActivity**.



(a)

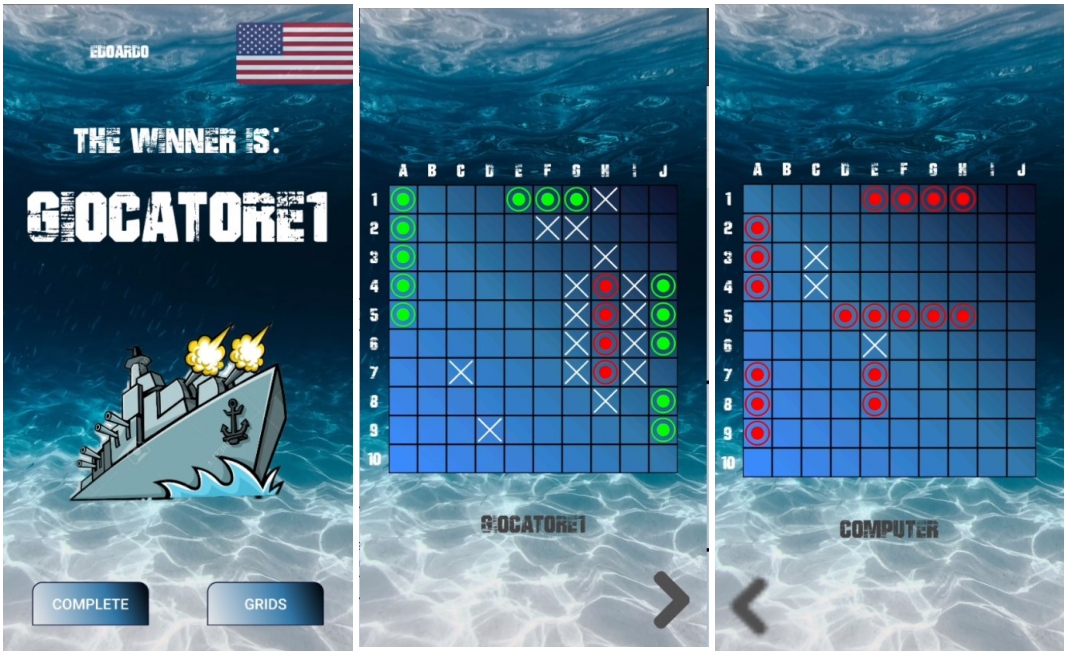
(b)

(c)

Figura 3.4: Screen del database prima (a) e dopo aver accettato la richiesta di gioco (c) e screen della MatchMakingActivity.

Le activity OnlineGameActivity e GameActivity funzionano di base allo stesso modo (con la differenza che la prima riceve notifiche da Firebase quando entrambi i giocatori sono pronti oppure quando uno dei due si disconnette). Inizialmente viene visualizzato un fragment di pre match nel quale il giocatore può posizionare le navi. Una volta che la partita viene inziata l'activity alterna i due match fragment attraverso *changeFragment(Fragment f)* e *switchMatchFragment(Fragment f)* quando viene richiesto dai fragment stessi.

Una volta completata la partita viene lanciata la ***WinnerActivity*** dove verrà visualizzato il nome del vincitore e, se la partita è stata completata, le configurazioni delle griglie in modo da vedere dove sono stati effettuati i colpi e dove erano posizionate le navi.



(a) (b) (c)

Figura 3.5: Screen della WinnerActivity (a) e delle configurazioni delle flotte dei due giocatori(b)(c): X = acqua colpita, Verde = nave non colpita, Rosso = nave colpita.

All'interno di ogni fragment in cui è necessario visualizzare una griglia è presente un oggetto *AbstractBattlefieldEngine* al quale viene passata un'istanza di una delle implementazioni tramite il file di layout del fragment e il metodo *findViewById(int id)*. Il BattleFieldEngine associato comunicherà, se necessario, con il fragment tramite l'interfaccia *IFragmentForBattlefield*, così da poter notificare quando è avvenuto un colpo, e quindi è necessario fare lo switch con l'altro MatchFragmet, o la partita è finita, e dunque lanciare la WinnerActivity.

3.3 Engines

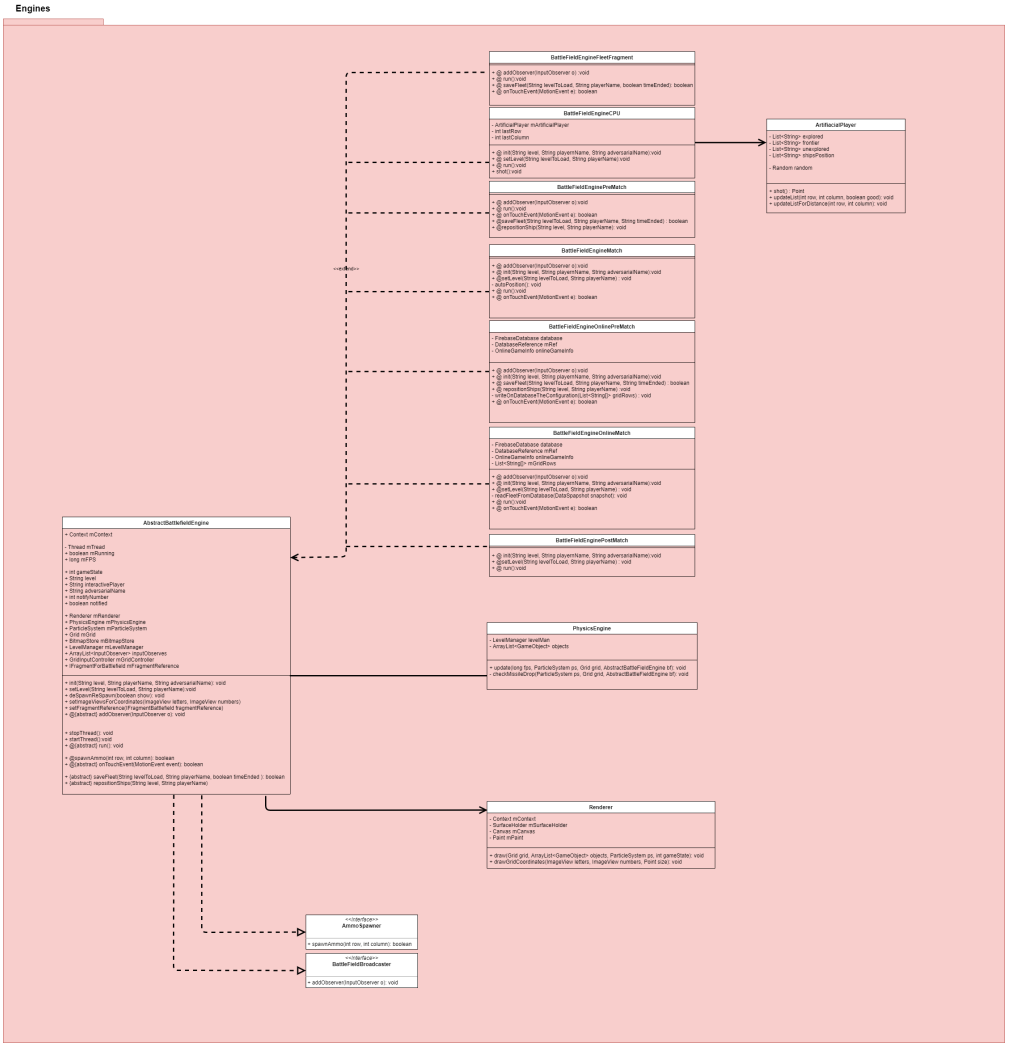


Figura 3.6: Diagramma UML del modulo Engines

All'interno del modulo ***Engines*** si trovano i motori di gestione dei vari campi da gioco, il **PhysicsEngine**, per la gestione dell'aggiornamento degli oggetti dinamici come il ParticleSystem e i missili, il **Renderer**, per il coordinamento dei metodi di disegno delle entità, e una classe **ArtificialPlayer**, che gestisce le scelte del giocatore artificiale nella modalità single player.

3.3.1 AbstractBattleFieldEngine e le sue estensioni

Di seguito vengono descritti brevemente i metodi che tutte le classi BattleFieldEngine implementano, ma che non per forza utilizzano. Successivamente verranno introdotti anche i metodi appartenenti a specifiche implementazioni del motore di gestione del campo da gioco.
I metodi principali sono i seguenti:

- *init(String level, String playerName, String adversarialName)*, permette di inizializzare tutte le istanze degli oggetti necessari e di impostare le variabili gameState, level, interactivePlayer e adversarialName. Nel caso dell'OnlineMatchBattleField è necessaria una completa sovrascrittura in quanto devono essere inseriti gli eventListener per l'invio e la ricezione delle notifiche di avvenuto colpo tramite Firebase.

- *setLevel(String levelToLoad, String playerName)*, metodo che permette di inizializzare il LevelManager e leggere la configurazione della griglia dal file csv corretto.

- *deSpawnReSpawn(boolean show)*, invocato da setLevel. Permette di leggere la configurazione della griglia e assegnare agli oggetti nave creati precedentemente dal LevelManager le giuste caratteristiche, come posizione iniziale e orientamento. Infine invoca il metodo spawn su i singoli oggetti indicando con show se devono apparire da subito sullo shermo.

- *run()*, metodo dell'interfaccia Runnable. Contiene un ciclo while che viene ripetuto fino a quando il *mThread* (istanza della classe Thread) è attivo. Invoca il metodo update del PhysicsEngine e il metodo draw del Renderer. Al termine di ogni cilo fa un check per verificare se è necessario notificare l'esecuzione di un colpo al fragment.

- *spawnAmmo(int row, int column)*, metodo dell'interfaccia AmmoSpawner, permette di far apparire il missile a seguito di un evento touch quando necessario.

- *saveFleet(String levelToLoad, String playerName, boolean TimeEnded)*, permette di salvare in un file csv la configurazione della flotta mostrata a video al momento dell'invocazione se tale disposizione è valida. La variabile timeEnded indica se il metodo è stato chiamato a seguito dello scadere di un timer. In questo caso se la flotta non ripsetta le regole viene riposizionata randomicamente tramite *repositionShips(String level, String playerName)*

Per la gestione degli eventi touch è stato utilizzato l'***Observer pattern***.

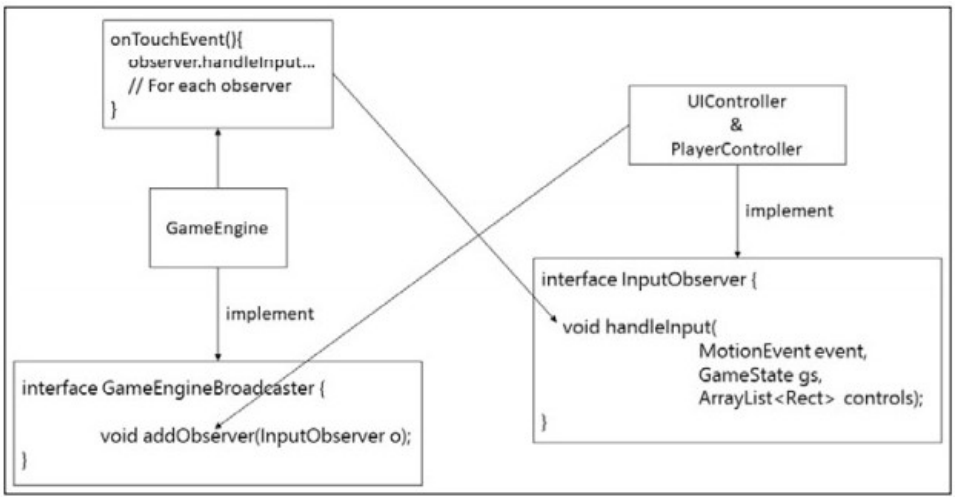


Figura 3.7: Esempio di ObserverPattern preso da Learning Java by Building Android Games

Come si può vedere in figura il motore di gioco implementa l'interfaccia GameEngineBroadcaster e dunque ha un metodo addObserver. Le classi UiController e PlayerController, le quali implementano InputObserver, passano un riferimento a se stessi tramite addObserver al GameEngine. Ad ogni evento touch il motore di gioco dovrà semplicemente passare l'evento a tutti gli observer, che sapranno poi come gestirlo. In questo modo se sarà necessario aggiungere una nuova classe per gestire gli input sarà sufficiente che implementino InputObserver e che chiamino addObserver nel costruttore.

Altri metodi che vale la pane andare a sottolineare sono:

- *shot()*, metodo di BattleFieldEngineCPU, il quale invoca a sua volta il metodo shot di ArtificialPlayer.
- *autoPosition()*, metodo di BattleFiedlEngineMatch. Permette una disposizione randomica delle navi all'interno dela griglia. Usato in single player per posizionare le navi del computer.
- writeOnDatabaseTheConfiguration(List <String[]>gridRows) e readFleetFromDatabase(dataSnapshot snapshot), metodi di BattleFieldEngineOnlinePreMatch e BattleFieldEngineOnlineMatch i quali permettono di andare a scrivere e leggere la configurazione della flotta su e da Firebase.

3.3.2 PhysicsEngine e Renderer

Nella classe ***PhysicsEngine*** è presente il metodo *update(long fps, ParticleSystem ps, Grid grid, AbstarctBattlefield bf)* tramite il quale vengono aggiornate le posizioni dei game object che hanno un UpdateComponent (spiegato nel modulo Entities) e il particle system se attivo. Inoltre viene invocato il metodo *checkMissileDrop(ParticleSystem ps, Grid grid, AbstarctBattlefield bf)* che verifica se il missile è arrivato alle coordinate indicate desiderate, gestisce un eventuale collisione con la nave e imposta il notifyNumber in base all'esito del colpo.

La classe ***Renderer*** invece ha il compito di dover invocare i metodi draw degli oggetti e della griglia.

3.4 Entities

All'interno di ***Entities*** si trovano le classi che definiscono le singole entità.

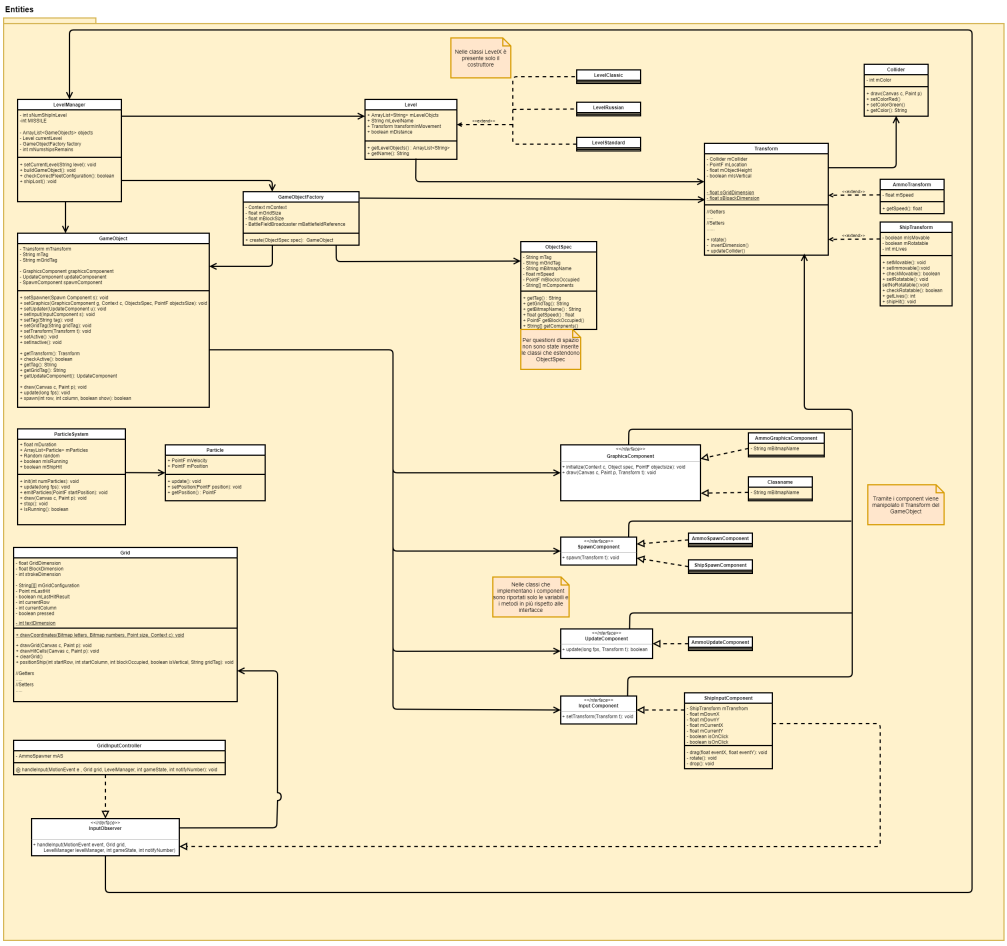


Figura 3.8: Diagramma UML del modulo Entities

3.4.1 ParticleSystem

La classe **ParticleSystem** crea un sistema di oggetti **Particle**. Viene utilizzato per simulare lo splash del razzo o l'esplosione della nave. Tramite *init(int numParticles)* viene indicato il numero di oggetti Particle che devono comporre il sistema che verranno poi emessi con l'invocazione di *emitParticle(PointF startPosition)*. I meetodi *update(long fps)*, *draw(Canvas c, Paint p)* e *isRunning()* sono utilizzati da PhysicsEngine e Renderer per la rappresentazione a video.

3.4.2 Grid

La classe **Grid** contiene tutta la logica e i metodi draw necessari per la rappresentazione della griglia. Oltre ad una serie di getters e setters, sono presenti le funzioni:

- *positionShip(int startRow, int StartColumn, boolean isVertical, String gridTag)*, utilizzata da saveFleet e da autoPosition del motore di gestione per "scrivere" un'oggetto nave sulla griglia,
- *clearGrid()*, imposta tutte le celle a zero.
- *setDispositionInGrid(List<String[]>gridRows)*, usato in setLevel per passare il risultato di lettura del file csv all'oggetto grid.

Gli eventi touch che coinvolgono la griglia sono gestiti dall'observer **GridInputController**. Tramite il metodo *handleInput(MotionEvent e, Grid grid, LevelManager, int gameState, int notifyNumber)* dell'interfaccia InputObserver, questa classe verifica se si è in fase di match. Se ciò è vero allora si calcola la coppia riga-colonna corrispondente alle coordinate dell'evento Action-DOWN e, se i valori sono validi, viene passata all'oggetto Grid, aggiornandola fino al momento di un evento Action-UP. A questo punto si prendono le ultime coordinate indicate e se corrispondono ad una cella non ancora selezionata si verifica se è stata colpita una nave. Alla fine del check viene invocato *setLastHit(int row, int column, boolean hit)* su grid e viene azionata l'animazione del razzo tramite *spawnAmmo(int row, int column)* dell'interfaccia AmmoSpawner. Tramite le informazione sull'ultimo colpo il PhisycsEngine ha a disposizione tutto il necessario per gestire l'eventuale collisione razzo-nave.

3.4.3 LevelManager e le classi Level

Tramite la classe **LevelManager** vengono creati tutti i GameObject necessari. Al costruttore viene indicato il livello da dover caricare con *setCurrentLevel(String level)*. All'interno di questo metodo infatti è presente uno switch che crea un'istanza del livello richiesto (LevelClassic, LevelRussian, LevelStandard). Successivamente viene invocato il metodo *buildGameObject()*, il quale va ad ottenere prima di tutto la lista degli oggetti tramite la procedura *getLevelObjects()* della classe astratta Level e poi per ogni elemento crea l'oggetto corrispondente tramite un'istanza della classe GameObjectFactory. Grazie *checkCorrectFleetConfiguration()* il BattleFieldEngine può verificare se le navi sono disposte con una configurazione valida.

Grazie a questa implementazione aggiungere un nuovo livello è molto semplice. Infatti è sufficiente aggiungere una classe LevelX che estenda la classe Level e

indicare nel costruttore il nome del livello, se è necessaria la distanza tra le navi, la lista delle navi e l'arma da utilizzare.

```
public class LevelClassic extends Level {

    public LevelClassic(){
        mLevelName = "classic";
        mDistance = false; //indica se la distanza è necessaria
        mLevelObjects = new ArrayList<String>();

        //aggiungo le navi
        mLevelObjects.add("battleship"); //5 blocchi
        mLevelObjects.add("cruiser"); //4 blocchi
        mLevelObjects.add("destroyer"); //3 blocchi
        mLevelObjects.add("rescue"); //3 blocchi
        mLevelObjects.add("patrol"); //2 blocchi


        //aggiungo il missile
        mLevelObjects.add("missile");
    }


}
```

Figura 3.9: esempio della classe LevelClassic

3.4.4 GameObject e GameOcjectFactory

Per quanto rigaurda la rappresetnazione degli oggetti nave e arma sono stati utilizzati i pattern Entity-Component e Simple Factory. In questo modo si facilita la loro gestione da parte del BattleFieldEngine di turno e l'inserimento di nuovi oggetti.

Come primo passo è stata implementata una classe **GameObject** che permette di rappresentare un qualsiasi oggetto, indipendentemente che sia un oggetto nave o arma. Successivamente è stata scritta la super classe **ObjectSpec** la quale contiene tutti i getters così che la classe di factory possa ottenere tutte le informazioni necessarie. Tutte le classi che rappresentano i game object reali dovranno semplicemente inizializzare le variabili di istanza e chiamare il costruttore della super classe. Tra le varibili è presente la lista *mComponents* che indica i nomi dei component richiesti per la costruzione dell'oggetto stesso.

```
public class BattleshipSpec extends ObjectSpec {

    private static final String tag = "Ship";
    private static final String gridTag = "B";
    private static final String bitmapName = "ship_battleship";
    private static final float speed = 0;
    private static final PointF blocksOccupied= new PointF( x: 1, y: 5);
    private static final  String[] components = new String[]{"ShipGraphicsComponent",
            "ShipSpawnComponent",
            "ShipInputComponent",
    };

    public BattleshipSpec() { super(tag, gridTag, bitmapName, speed,blocksOccupied, components); }
}
```

Figura 3.10: Esempio di una sotto classe ObjectSpec

A questo punto sono state implementate le interfacce dei component:

- **GraphicsComponent**, è il componente che contiene il metodo draw dell'oggetto. Inoltre è presente il metodo *initialize(Context c , ObjectSpec spec, PointF objectSize)* che permette di caricare la bitmap nel BitmapStore.
- **InputComponent**, component per modificare i valori del Transform associato al game object a seguito di eventi touch (ad esempio ShipInputCompoenent permette di gestire il drag&drop delle navi o la loro rotazione).
- **UpdateComponent**, componente per aggiornare la posizione di oggetti che si muovono senza il bisogno di eventi touch (ad esempio AmmoUpdateComponent aggiorna la posizione del razzo in base alla velocità indicata nel transform e il valore fps).
- **SpawnComponent**, gestisce l'apparizione e il posizionamento inziale degli oggetti.

Successivamente sono state implementate le classi di tutti i componenti necessari.

Sono state dunque realizzate la super classe **Transform** e le sotto classi **ShipTransform** e **AmmoTransform** le quali contengono informazioni sull'oggetto come posizione, velocità e dimensione oltre ad un oggetto **Collider** utilizzato per rilevare delle collisioni tra oggetti.

Alla fine è stata implementata la **GameObjectFactory** la quale assembla l'oggetto nel metodo *create(Objectspec spec)* in base alla specificazione passata.

Per aggiungere un nuovo oggetto al gioco è dunque sufficiente creare una classe che estenda ObjectSpec, definire nuovi components e un nuovo transform se necessario (in questo caso serve modificare anche gli switch della GameObjectFactory) e aggiungerlo ad un livello nella rispettiva classe. Se invece si vuole modificare un oggetto già esistente basta andare a cambiare i valori della classe spec corrispondente.

3.5 Problemi Riscontrati

Il primo problema che si è affrontato nell'implementazione dell'app è stato la gestione dei metodi di lifecycle delle activity e la comunicazione tra activity e fragment. Per apprendere meglio il funzionamento di base di questi oggetti è stato utilizzato il libro *Android 6: guida per lo sviluppatore* e il sito internet *Android Developoers*. La fase centrale del progetto, ovvero la realizzazione delle classi dei moduli Engines e Entities, non ha presentato problematiche rilevanti anche grazie all'utilizzo del libro *Learning Java by Building Android Games* già accennato precedentemente. Senza dubbio però la parte che ha necessitato di più tempo per una corretta implementazione è stata la gestione della piattaforma Firebase. In particolare è risultato complesso riuscire a coordinare i due schermi a seguito della disconnessione di uno dei due giocatori e per l'avvio della partita.

4. Appendice

I metodi più complessi e importanti del codice sono stati riportati della sezione del modulo di appartenenza. Inoltre all'interno del progetto sono stati inseriti dei commenti nei passaggi più delicati per una maggiore chiarezza sul loro comportamento.

5. Bibliografia

- Battaglia navale (gioco), Wikipedia.
- Android 6: guida per lo sviluppatore. Autore: Massimo Carli. Casa Editrice: Apogeo.
- Learning Java by Building Android Games, seconda edizione. Autore: John Horton. Casa Editrice: Packt.
- Android Developers
- Firebase Docs